

**Reproduction prohibited without permission of the author**



Object-oriented Software in Ada 95 Second Edition

Object-oriented Software in Ada 95 Second Edition

Michael A. Smith  
School of Computing  
University of Brighton

# Contents

<b>CONTENTS .....</b>	<b>IV</b>
1.1 GLOSSARY OF TERMS USED .....	XV
<b>1 INTRODUCTION TO PROGRAMMING.....</b>	<b>1</b>
1.1 COMPUTER PROGRAMMING .....	1
1.2 PROGRAMMING LANGUAGES .....	1
1.3 RANGE OF PROGRAMMING LANGUAGES .....	2
1.3.1 Computer programming languages .....	2
1.3.2 The role of a compiler .....	2
1.4 A SMALL PROBLEM .....	3
1.5 SOLVING THE PROBLEM USING A CALCULATOR .....	3
1.5.1 Making the solution more general .....	4
1.6 SOLVING THE PROBLEM USING THE ADA 95 LANGUAGE .....	4
1.6.1 Running the program .....	6
1.7 THE DECLARE BLOCK .....	6
1.8 THE ROLE OF COMMENTS .....	7
1.9 SUMMARY .....	8
1.10 A MORE DESCRIPTIVE PROGRAM .....	8
1.10.1 Running the new program .....	9
1.11 TYPES OF MEMORY LOCATION .....	9
1.11.1 Warning .....	10
1.12 REPETITION .....	10
1.13 INTRODUCTION TO THE WHILE STATEMENT .....	11
1.13.1 Conditions .....	12
1.13.2 A while statement in Ada 95 .....	12
1.13.3 Using the while statement .....	13
1.14 SELECTION .....	14
1.14.1 Using the if statement .....	15
1.15 SELF-ASSESSMENT .....	16
1.16 PAPER EXERCISES .....	17
<b>2 SOFTWARE DESIGN .....</b>	<b>18</b>
2.1 THE SOFTWARE CRISIS .....	18
2.2 A PROBLEM, THE MODEL AND THE SOLUTION .....	18
2.2.1 Responsibilities .....	19
2.3 OBJECTS .....	19
2.3.1 The car as an object .....	20
2.4 THE CLASS .....	21
2.5 METHODS AND MESSAGES .....	21
2.6 CLASS OBJECTS .....	21
2.7 INHERITANCE .....	22
2.8 POLYMORPHISM .....	23
2.9 SELF-ASSESSMENT .....	23
<b>3 ADA INTRODUCTION: PART 1 .....</b>	<b>25</b>
3.1 A FIRST ADA PROGRAM .....	25
3.2 THE CASE OF IDENTIFIERS IN A PROGRAM .....	26
3.3 FORMAT OF AN ADA PROGRAM .....	26
3.3.1 Variable names .....	26
3.3.2 Comments .....	27
3.4 A LARGER ADA PROGRAM .....	27
3.5 REPETITION: WHILE .....	28
3.6 SELECTION: IF .....	28
3.7 OTHER REPETITION CONSTRUCTS .....	29

3.7.1	<i>for</i> .....	29
3.7.2	<i>loop</i> .....	30
3.8	OTHER SELECTION CONSTRUCTS .....	31
3.8.1	<i>case</i> .....	31
3.9	INPUT AND OUTPUT .....	32
3.10	ACCESS TO COMMAND LINE ARGUMENTS .....	33
3.10.1	<i>Putting it all together</i> .....	34
3.11	A BETTER CAT PROGRAM .....	34
3.11.1	<i>Putting it all together</i> .....	35
3.12	CHARACTERS IN ADA .....	35
3.13	SELF-ASSESSMENT .....	36
3.14	EXERCISES.....	37
<b>4</b>	<b>ADA INTRODUCTION: PART 2.....</b>	<b>38</b>
4.1	INTRODUCTION .....	38
4.2	THE TYPE FLOAT .....	38
4.2.1	<i>Other Integer and Float data types</i> .....	39
4.3	NEW DATA TYPES .....	39
4.3.1	<i>Type conversions</i> .....	40
4.3.2	<i>Universal integer</i> .....	40
4.3.3	<i>Constant declarations</i> .....	40
4.4	MODIFIED COUNTDOWN PROGRAM.....	41
4.5	INPUT AND OUTPUT IN ADA .....	41
4.6	THE PACKAGE <code>ADA.FLOAT_TEXT_IO</code> .....	41
4.6.1	<i>Output of floating point numbers</i> .....	42
4.6.2	<i>Input of floating point numbers</i> .....	42
4.7	THE PACKAGE <code>ADA.INTEGER_TEXT_IO</code> .....	42
4.7.1	<i>Output of integer numbers</i> .....	42
4.7.2	<i>Input of integer numbers</i> .....	43
4.8	CONVERSION BETWEEN FLOAT AND INTEGER TYPES.....	43
4.9	TYPE SAFETY IN A PROGRAM .....	44
4.10	SUBTYPES .....	44
4.10.1	<i>Types vs. subtypes</i> .....	45
4.11	MORE ON TYPES AND SUBTYPES .....	45
4.11.1	<i>Root_Integer and Root_Real</i> .....	46
4.11.2	<i>Type declarations: root type of type</i> .....	46
4.11.3	<i>Arithmetic with types and subtypes</i> .....	47
4.11.4	<i>Warning</i> .....	47
4.11.5	<i>Constrained and unconstrained types</i> .....	48
4.11.6	<i>Implementation optimizations</i> .....	48
4.12	COMPILE-TIME AND RUN-TIME CHECKS .....	49
4.12.1	<i>Subtypes Natural and Positive</i> .....	50
4.13	ENUMERATIONS.....	50
4.13.1	<i>Enumeration values</i> .....	51
4.13.2	<i>The attributes 'Val and 'Pos</i> .....	51
4.14	THE SCALAR TYPE HIERARCHY.....	52
4.14.1	<i>The inbuilt types</i> .....	53
4.15	ARITHMETIC OPERATORS .....	53
4.15.1	<i>Exponentiation</i> .....	54
4.15.2	<i>Monadic arithmetic operators</i> .....	54
4.16	MEMBERSHIP OPERATORS .....	54
4.17	USE OF TYPES AND SUBTYPES WITH MEMBERSHIP OPERATOR.....	55
4.18	RELATIONAL OPERATORS.....	56
4.18.1	<i>Boolean operators</i> .....	56
4.18.2	<i>Monadic Boolean operators</i> .....	57
4.19	BITWISE OPERATORS.....	57
4.20	SELF-ASSESSMENT .....	58

4.21	EXERCISES .....	59
<b>5</b>	<b>PROCEDURES AND FUNCTIONS .....</b>	<b>60</b>
5.1	INTRODUCTION .....	60
5.2	FUNCTIONS .....	60
5.2.1	<i>Local variables</i> .....	61
5.2.2	<i>Separate compilation of functions</i> .....	61
5.3	PROCEDURES .....	62
5.3.1	<i>Separate compilation of procedures</i> .....	63
5.4	FORMAL AND ACTUAL PARAMETERS .....	64
5.5	MODES OF A PARAMETER TO A FUNCTION OR PROCEDURE .....	64
5.5.1	<i>Example of mode in out</i> .....	65
5.5.2	<i>Putting it all together</i> .....	65
5.5.3	<i>Summary of access to formal parameters</i> .....	66
5.6	RECURSION .....	66
5.6.1	<i>The procedure Write_Natural</i> .....	67
5.6.2	<i>Putting it all together</i> .....	67
5.7	OVERLOADING OF FUNCTIONS .....	67
5.8	DIFFERENT NUMBER OF PARAMETERS .....	69
5.9	DEFAULT VALUES AND NAMED PARAMETERS .....	69
5.9.1	<i>Putting it all together</i> .....	70
5.10	SELF-ASSESSMENT .....	71
5.11	EXERCISES .....	71
<b>6</b>	<b>PACKAGES AS CLASSES .....</b>	<b>73</b>
6.1	INTRODUCTION .....	73
6.2	OBJECTS, MESSAGES AND METHODS .....	74
6.3	OBJECTS, MESSAGES AND METHODS IN ADA .....	74
6.3.1	<i>An object for a bank account</i> .....	75
6.3.2	<i>The procedure Statement</i> .....	76
6.3.3	<i>Putting it all together</i> .....	76
6.3.4	<i>Components of a package</i> .....	76
6.3.5	<i>Specification of the package</i> .....	76
6.3.6	<i>A class diagram showing a class</i> .....	78
6.3.7	<i>Representation of the balance of the account</i> .....	78
6.3.8	<i>Implementation of the package</i> .....	78
6.3.9	<i>Terminology</i> .....	79
6.4	THE PACKAGE AS SEEN BY A USER .....	79
6.5	THE PACKAGE AS SEEN BY AN IMPLEMENTOR .....	79
6.6	THE CLASS .....	80
6.7	CLAUSES WITH AND USE .....	80
6.7.1	<i>To use or not to use the use clause</i> .....	81
6.7.2	<i>The package Standard</i> .....	81
6.7.3	<i>Positioning of with and use in a package declaration</i> .....	81
6.7.4	<i>Conflict in names in a package</i> .....	82
6.8	MUTATORS AND INSPECTORS .....	82
6.9	TYPE PRIVATE .....	83
6.9.1	<i>Type limited private</i> .....	83
6.10	INITIALIZING AN OBJECT AT DECLARATION TIME .....	84
6.10.1	<i>By discriminant</i> .....	84
6.10.2	<i>Restrictions</i> .....	85
6.10.3	<i>By assignment</i> .....	85
6.10.4	<i>Restrictions</i> .....	86
6.11	A PERSONAL ACCOUNT MANAGER .....	86
6.12	CLASS TUI .....	89
6.13	SELF-ASSESSMENT .....	91
6.14	EXERCISES .....	92

<b>7</b>	<b>DATA STRUCTURES .....</b>	<b>94</b>
7.1	THE RECORD STRUCTURE.....	94
7.2	OPERATIONS ON A DATA STRUCTURE .....	94
7.2.1	<i>Other operations allowed on data structures.....</i>	95
7.3	NESTED RECORD STRUCTURES .....	96
7.4	DISCRIMINANTS TO RECORDS .....	96
7.5	DEFAULT VALUES TO A DISCRIMINANT .....	97
7.5.1	<i>Constrained vs. unconstrained discriminants.....</i>	98
7.5.2	<i>Restrictions on a discriminant.....</i>	98
7.6	VARIANT RECORDS .....	98
7.7	LIMITED RECORDS .....	100
7.8	DATA STRUCTURE VS. CLASS .....	100
7.9	SELF-ASSESSMENT .....	100
7.10	EXERCISES.....	101
<b>8</b>	<b>ARRAYS .....</b>	<b>102</b>
8.1	ARRAYS AS CONTAINER OBJECTS.....	102
8.2	ATTRIBUTES OF AN ARRAY.....	104
8.3	A HISTOGRAM .....	104
8.3.1	<i>Putting it all together .....</i>	107
8.4	THE GAME OF NOUGHTS AND CROSSES .....	108
8.4.1	<i>The class Board.....</i>	109
8.4.2	<i>Implementation of the game .....</i>	109
8.4.3	<i>Displaying the Board.....</i>	110
8.4.4	<i>The class Board.....</i>	111
8.4.5	<i>Putting it all together .....</i>	113
8.5	MULTIDIMENSIONAL ARRAYS.....	113
8.5.1	<i>An alternative way of declaring multidimensional arrays.....</i>	114
8.5.2	<i>Attributes of multidimensional arrays.....</i>	115
8.6	INITIALIZING AN ARRAY.....	115
8.6.1	<i>Multidimensional initializations.....</i>	116
8.7	UNCONSTRAINED ARRAYS .....	117
8.7.1	<i>Slices of an array .....</i>	117
8.7.2	<i>Putting it all together .....</i>	118
8.8	STRINGS .....	118
8.9	DYNAMIC ARRAYS .....	119
8.9.1	<i>Putting it all together .....</i>	119
8.10	A NAME AND ADDRESS CLASS .....	120
8.10.1	<i>Putting it all together .....</i>	122
8.11	AN ELECTRONIC PIGGY BANK.....	122
8.12	SELF-ASSESSMENT .....	125
8.13	EXERCISES.....	126
<b>9</b>	<b>CASE STUDY: DESIGN OF A GAME.....</b>	<b>127</b>
9.1	REVERSI .....	127
9.1.1	<i>A program to play reversi.....</i>	128
9.2	ANALYSIS AND DESIGN OF THE PROBLEM .....	128
9.3	CLASS DIAGRAM.....	130
9.4	SPECIFICATION OF THE ADA CLASSES .....	130
9.5	IMPLEMENTATION OF THE MAIN CLASS GAME.....	132
9.5.1	<i>Running the program.....</i>	133
9.5.2	<i>Example of a typical game .....</i>	133
9.6	IMPLEMENTATION OF THE OTHER CLASSES .....	135
9.7	SELF-ASSESSMENT .....	146
9.8	EXERCISES.....	146
<b>10</b>	<b>INHERITANCE.....</b>	<b>147</b>

10.1	INTRODUCTION .....	147
10.2	TAGGED TYPES .....	148
10.2.1	<i>Terminology</i> .....	148
10.3	THE CLASS INTEREST_ACCOUNT.....	148
10.3.1	<i>Terminology</i> .....	152
10.4	VISIBILITY RULES (NORMAL INHERITANCE).....	152
10.5	CONVERTING A DERIVED CLASS TO A BASE CLASS .....	152
10.6	ABSTRACT CLASS .....	153
10.6.1	<i>Putting it all together</i> .....	155
10.6.2	<i>Visibility of base class methods</i> .....	156
10.7	MULTIPLE INHERITANCE.....	156
10.7.1	<i>Putting it all together</i> .....	159
10.8	INITIALIZATION AND FINALIZATION .....	159
10.8.1	<i>Implementation</i> .....	161
10.8.2	<i>Putting it all together</i> .....	162
10.8.3	<i>Warning</i> .....	162
10.9	HIDING THE BASE CLASS METHODS .....	163
10.9.1	<i>Visibility rules (Hidden base class)</i> .....	164
10.9.2	<i>Putting it all together</i> .....	164
10.10	SELF-ASSESSMENT .....	164
10.11	EXERCISES .....	165
<b>11</b>	<b>CHILD LIBRARIES .....</b>	<b>166</b>
11.1	INTRODUCTION .....	166
11.1.1	<i>Putting it all together</i> .....	167
11.1.2	<i>Warning</i> .....	167
11.2	VISIBILITY RULES OF A CHILD PACKAGE.....	168
11.3	PRIVATE CHILD .....	169
11.3.1	<i>Visibility rules of a private child package</i> .....	169
11.4	CHILD PACKAGES VS. INHERITANCE.....	169
11.5	SELF-ASSESSMENT .....	170
11.6	EXERCISES.....	170
<b>12</b>	<b>DEFINING NEW OPERATORS.....</b>	<b>171</b>
12.1	DEFINING OPERATORS IN ADA .....	171
12.2	A RATIONAL ARITHMETIC PACKAGE.....	172
12.2.1	<i>Ada specification of the package</i> .....	172
12.2.2	<i>Ada implementation of the package</i> .....	173
12.3	A BOUNDED STRING CLASS .....	176
12.3.1	<i>Overloading = and /=</i> .....	177
12.3.2	<i>Specification of the class Bounded_String</i> .....	177
12.3.3	<i>Putting it all together</i> .....	180
12.3.4	<i>Ada.Strings.Bounded a standard library</i> .....	180
12.3.5	<i>use type</i> .....	181
12.4	SELF-ASSESSMENT .....	181
12.5	EXERCISES.....	181
<b>13</b>	<b>EXCEPTIONS.....</b>	<b>183</b>
13.1	THE EXCEPTION MECHANISM .....	183
13.2	RAISING AN EXCEPTION .....	184
13.3	HANDLING ANY EXCEPTION .....	184
13.4	THE CAT PROGRAM REVISITED .....	186
13.5	A STACK .....	186
13.5.1	<i>Putting it all together</i> .....	188
13.5.2	<i>Implementation of the stack</i> .....	188
13.6	SELF-ASSESSMENT .....	189
13.7	EXERCISES.....	190



<b>14</b>	<b>GENERIC</b>	<b>191</b>
14.1	GENERIC FUNCTIONS AND PROCEDURES	191
14.1.1	<i>Advantages and disadvantages of generic units</i>	193
14.2	SPECIFICATION OF GENERIC COMPONENT	194
14.3	GENERIC STACK	195
14.3.1	<i>Putting it all together</i>	197
14.3.2	<i>Implementation techniques for a generic package</i>	198
14.4	GENERIC FORMAL SUBPROGRAMS	198
14.4.1	<i>Example of the use of the generic procedure G_3Order</i>	200
14.4.2	<i>Summary</i>	200
14.5	SORTING	201
14.5.1	<i>Efficiency</i>	201
14.6	A GENERIC PROCEDURE TO SORT DATA	202
14.6.1	<i>Putting it all together</i>	203
14.6.2	<i>Sorting records</i>	203
14.7	GENERIC CHILD LIBRARY	204
14.7.1	<i>Putting it all together</i>	206
14.8	INHERITING FROM A GENERIC CLASS	206
14.8.1	<i>Putting it all together</i>	207
14.9	SELF-ASSESSMENT	208
14.10	EXERCISES	208
<b>15</b>	<b>DYNAMIC MEMORY ALLOCATION</b>	<b>209</b>
15.1	ACCESS VALUES	209
15.1.1	<i>Access to an object via its access value</i>	210
15.1.2	<i>Lvalues and rvalues</i>	210
15.1.3	<i>Read only access</i>	211
15.2	DYNAMIC ALLOCATION OF STORAGE	212
15.2.1	<i>Problems with dynamically allocated storage</i>	215
15.3	RETURNING DYNAMICALLY ALLOCATED STORAGE	215
15.3.1	<i>Summary: access all, access constant, access</i>	216
15.4	USE OF DYNAMIC STORAGE	216
15.4.1	<i>Putting it all together</i>	219
15.5	HIDING THE STRUCTURE OF AN OBJECT (OPAQUE TYPE)	220
15.5.1	<i>Putting it all together</i>	222
15.5.2	<i>Hidden vs. visible storage in a class</i>	223
15.6	ACCESS VALUE OF A FUNCTION	223
15.6.1	<i>Putting it all together</i>	224
15.7	ATTRIBUTES 'ACCESS AND 'UNCHECKED_ACCESS	225
15.8	SELF-ASSESSMENT	226
15.9	EXERCISES	226
<b>16</b>	<b>POLYMORPHISM</b>	<b>227</b>
16.1	ROOMS IN A BUILDING	227
16.1.1	<i>Dynamic binding</i>	228
16.2	A PROGRAM TO MAINTAIN DETAILS ABOUT A BUILDING	228
16.2.1	<i>Putting it all together</i>	231
16.3	RUN-TIME DISPATCH	232
16.4	HETEROGENEOUS COLLECTIONS OF OBJECTS	232
16.4.1	<i>An array as a heterogeneous collection</i>	233
16.4.2	<i>Additions to the class Office and Room</i>	233
16.5	A BUILDING INFORMATION PROGRAM	235
16.5.1	<i>Putting it all together</i>	236
16.6	FULLY QUALIFIED NAMES AND POLYMORPHISM	237
16.7	PROGRAM MAINTENANCE AND POLYMORPHISM	238
16.8	DOWNCASTING	238

## x Contents

16.8.1	Converting a base class to a derived class.....	239
16.9	THE OBSERVE-OBSERVER PATTERN.....	240
16.9.1	The Observer's responsibilities.....	241
16.9.2	The responsibilities of the observable object.....	241
16.9.3	Putting it all together.....	242
16.10	USING THE OBSERVE-OBSERVER PATTERN.....	244
16.10.1	The observed board object.....	244
16.10.2	An observer for the class <i>Board</i> .....	246
16.10.3	The driver code for the program of nought and crosses.....	246
16.11	SELF-ASSESSMENT.....	247
16.12	EXERCISES.....	248
<b>17</b>	<b>CONTAINERS.....</b>	<b>249</b>
17.1	LIST OBJECT.....	249
17.1.1	List vs. array.....	251
17.2	METHODS IMPLEMENTED IN A LIST.....	251
17.2.1	Example of use.....	251
17.3	SPECIFICATION AND IMPLEMENTATION OF THE LIST CONTAINER.....	253
17.3.1	The list iterator.....	256
17.3.2	Relationship between a list and its iterator.....	260
17.4	LIMITATIONS OF THE LIST IMPLEMENTATION.....	260
17.5	REFERENCE COUNTING.....	262
17.6	IMPLEMENTATION OF A REFERENCE COUNTING SCHEME.....	264
17.6.1	Ada specification.....	265
17.6.2	Ada implementation.....	266
17.6.3	Putting it all together.....	268
17.7	A SET.....	269
17.7.1	Putting it all together.....	271
17.8	SELF-ASSESSMENT.....	271
17.9	EXERCISES.....	272
<b>18</b>	<b>INPUT AND OUTPUT.....</b>	<b>273</b>
18.1	THE INPUT AND OUTPUT MECHANISM.....	273
18.1.1	Putting it all together.....	274
18.2	READING AND WRITING TO FILES.....	275
18.3	READING AND WRITING BINARY DATA.....	276
18.4	SWITCHING THE DEFAULT INPUT AND OUTPUT STREAMS.....	278
18.4.1	Putting it all together.....	278
18.5	SELF-ASSESSMENT.....	279
18.6	EXERCISES.....	279
<b>19</b>	<b>PERSISTENCE.....</b>	<b>280</b>
19.1	A PERSISTENT INDEXED COLLECTION.....	280
19.1.1	Putting it all together.....	282
19.1.2	Setting up the persistent object.....	282
19.2	THE CLASS <i>PIC</i> .....	282
<b>20</b>	<b>TASKS.....</b>	<b>289</b>
20.1	THE TASK MECHANISM.....	289
20.1.1	Putting it all together.....	290
20.1.2	Task rendezvous.....	291
20.1.3	The task's implementation.....	292
20.2	PARAMETERS TO A TASK TYPE.....	293
20.2.1	Putting it all together.....	294
20.3	MUTUAL EXCLUSION AND CRITICAL SECTIONS.....	294
20.4	PROTECTED TYPE.....	295
20.5	IMPLEMENTATION.....	295

20.5.1	Barrier condition entry .....	298
20.5.2	Putting it all together .....	300
20.6	DELAY .....	300
20.7	CHOICE OF ACCEPTS .....	300
20.7.1	Accept alternative .....	301
20.7.2	Accept time-out .....	301
20.8	ALTERNATIVES TO A TASK TYPE .....	302
20.8.1	As part of a package .....	302
20.8.2	As part of a program unit .....	304
20.9	SELF-ASSESSMENT .....	304
20.10	EXERCISES .....	305
<b>21</b>	<b>SYSTEM PROGRAMMING.....</b>	<b>306</b>
21.1	REPRESENTATION CLAUSE.....	306
21.1.1	Putting it all together .....	307
21.2	BINDING AN OBJECT TO A SPECIFIC ADDRESS .....	307
21.2.1	Access to individual bits .....	308
21.3	SELF-ASSESSMENT .....	310
21.4	EXERCISES .....	310
<b>22</b>	<b>A TEXT USER INTERFACE.....</b>	<b>311</b>
22.1	SPECIFICATION .....	311
22.2	API FOR TUI .....	312
22.2.1	To set up and close down the TUI .....	312
22.2.2	Window API calls .....	313
22.2.3	Dialog API calls .....	313
22.2.4	User interaction with the TUI .....	314
22.2.5	Classes used .....	314
22.3	AN EXAMPLE PROGRAM USING THE TUI .....	314
22.3.1	How it all fits together .....	316
22.3.2	Putting it all together .....	317
22.4	THE MENU SYSTEM .....	317
22.5	NOUGHTS AND CROSSES PROGRAM .....	320
22.5.1	The class Board .....	320
22.5.2	Package Pack_Program .....	322
22.5.3	Putting it all together .....	326
22.6	SELF-ASSESSMENT .....	326
22.7	EXERCISES .....	327
<b>23</b>	<b>TUI: THE IMPLEMENTATION .....</b>	<b>328</b>
23.1	OVERVIEW OF THE TUI .....	328
23.1.1	Structure of the TUI .....	329
23.2	IMPLEMENTATION OF THE TUI .....	329
23.2.1	Constants used in the TUI .....	330
23.2.2	Raw input and output .....	330
23.2.3	Machine-dependent I/O .....	331
23.2.4	The class Screen .....	333
23.3	THE CLASS ROOT_WINDOW .....	334
23.4	THE CLASSES INPUT_MANAGER AND WINDOW_CONTROL .....	335
23.4.1	Specification of the class Input_manager .....	335
23.4.2	Specification of the class Window_control .....	335
23.4.3	Implementation of the class Input_manager .....	336
23.4.4	Implementation of the class Window_control .....	337
23.5	OVERLAPPING WINDOWS .....	340
23.6	THE CLASS WINDOW .....	340
23.6.1	Application API .....	340
23.6.2	Window system API .....	341

23.6.3	<i>The specification for the class Window</i> .....	341
23.6.4	<i>Implementation of the class Window</i> .....	343
23.7	THE CLASS DIALOG.....	348
23.7.1	<i>Implementation of the class Dialog</i> .....	349
23.8	THE CLASS MENU .....	350
23.8.1	<i>Implementation of the class Menu</i> .....	352
23.9	THE CLASS MENU_TITLE.....	354
23.9.1	<i>Implementation of the class Menu_title</i> .....	354
23.10	SELF-ASSESSMENT .....	355
23.11	EXERCISES .....	355
<b>24</b>	<b>SCOPE OF DECLARED OBJECTS .....</b>	<b>357</b>
24.1	NESTED PROCEDURES .....	357
24.1.1	<i>Advantages of using nested procedures</i> .....	359
24.1.2	<i>Introducing a new lexical level in a procedure or function</i> .....	359
24.1.3	<i>Holes in visibility</i> .....	359
24.1.4	<i>Consequences of lexical levels</i> .....	359
24.2	SELF-ASSESSMENT .....	360
<b>25</b>	<b>MIXED LANGUAGE PROGRAMMING.....</b>	<b>361</b>
25.1	LINKING TO OTHER CODE.....	361
25.2	SELECTED TYPES AND FUNCTIONS FROM INTERFACES . C.....	361
25.2.1	<i>Integer, character and floating point types</i> .....	361
25.2.2	<i>C String type</i> .....	361
25.2.3	<i>Selected functions</i> .....	361
25.3	AN ADA PROGRAM CALLING A C FUNCTION .....	362
25.3.1	<i>Another example</i> .....	363
25.4	AN ADA PACKAGE IN C.....	363
25.5	LINKING TO FORTRAN AND COBOL CODE .....	365
<b>APPENDIX A:</b>	<b>THE MAIN LANGUAGE FEATURES OF ADA 95.....</b>	<b>366</b>
	SIMPLE OBJECT DECLARATIONS .....	366
	ARRAY DECLARATION .....	366
	TYPE AND SUBTYPE DECLARATIONS .....	366
	ENUMERATION DECLARATION .....	366
	SIMPLE STATEMENTS .....	366
	BLOCK.....	366
	SELECTION STATEMENTS .....	366
	LOOPING STATEMENTS .....	367
	ARITHMETIC OPERATORS.....	367
	CONDITIONAL EXPRESSIONS.....	367
	EXITS FROM LOOPS .....	367
	CLASS DECLARATION AND IMPLEMENTATION .....	368
	INHERITANCE.....	368
	PROGRAM DELAY.....	369
	TASK .....	369
	COMMUNICATION WITH A TASK .....	369
	RENDEZVOUS .....	369
	PROTECTED TYPE.....	370
<b>APPENDIX B:</b>	<b>COMPONENTS OF ADA .....</b>	<b>371</b>
B.1	RESERVED WORDS AND OPERATORS IN ADA 95.....	371
B.1.1	<i>Reserved words</i> .....	371
B.1.2	<i>Operators</i> .....	371
B.2	ATTRIBUTES OF OBJECTS AND TYPES .....	371
B.2.1	<i>Scalar objects</i> .....	371
B.2.2	<i>Array objects and types</i> .....	371

B.2.3	Scalar objects and types .....	372
B.2.4	Discrete objects .....	372
B.2.5	Task objects and types.....	372
B.2.6	Floating point objects and types .....	372
B.3	LITERALS IN ADA .....	373
B.4	OPERATORS IN ADA 95.....	373
B.4.1	Priority of operators from high to low.....	374
B.5	ADA TYPE HIERARCHY .....	374
B.6	IMPLEMENTATION REQUIREMENTS OF STANDARD TYPES .....	375
B.7	EXCEPTIONS .....	375
B.7.1	Pre-defined exceptions.....	375
B.7.2	I/O exceptions.....	376
B.8	ADA 95, THE STRUCTURE.....	376
B.9	SOURCES OF INFORMATION .....	376
B.8.1	Copies of the Ada 95 compiler.....	376
B.8.2	Ada information on the World Wide Web.....	377
B.8.3	News groups.....	377
B.8.4	CD ROMs.....	377
B.8.5	Additional information on this book .....	377
<b>APPENDIX C: LIBRARY FUNCTIONS AND PACKAGES .....</b>		<b>378</b>
C.1	GENERIC FUNCTION UNCHECKED_CONVERSION.....	378
C.2	GENERIC FUNCTION UNCHECKED_DEALLOCATION.....	378
C.4	THE PACKAGE STANDARD .....	378
C.5	THE PACKAGE ADA.TEXT_IO .....	382
C.6	THE PACKAGE ADA.SEQUENTIAL_IO.....	389
C.7	THE PACKAGE ADA.CHARACTERS.HANDLING.....	390
C.8	THE PACKAGE ADA.STRINGS.BOUNDED.....	391
C.9	THE PACKAGE INTERFACES.C .....	397
C.10	THE PACKAGE ADA.NUMERICS .....	399
C.11	THE PACKAGE ADA.NUMERICS.GENERIC_ELEMENTARY_FUNCTIONS.....	399
C.12	THE PACKAGE ADA.COMMAND_LINE .....	400
C.13	THE PACKAGE ADA.FINALIZATION.....	400
C.14	THE PACKAGE ADA.TAGS.....	401
C.15	THE PACKAGE ADA.CALENDAR .....	401
C.16	THE PACKAGE SYSTEM.....	402
<b>APPENDIX D: ANSWERS TO SELECTED EXERCISES .....</b>		<b>404</b>
FROM CHAPTER 2 .....		404
FROM CHAPTER 3 .....		405
FROM CHAPTER 4 .....		407
FROM CHAPTER 5 .....		408
FROM CHAPTER 6 .....		410
FROM CHAPTER 9 .....		411
FROM CHAPTER 13.....		413
FROM CHAPTER 14.....		415
FROM CHAPTER 19.....		417
<b>REFERENCES .....</b>		<b>419</b>
<b>26</b>	<b>INDEX.....</b>	<b>420</b>

# Preface

This book is aimed at students and programmers who wish to learn the object-oriented language Ada 95. The book illustrates the language by showing how programs can be written using an object-oriented approach. The book treats Ada 95 as a language in its own right and not just as an extension to Ada 83.

The first chapter provides an introduction to problem solving using an object-oriented design methodology. The methodology illustrated in this introductory chapter is based on Fusion.

The next three chapters concentrate on the basic constructs in the Ada 95 language. In particular the use of types and subtypes is encouraged. By using types and subtypes in a program the compiler can help spot many errors and inconsistencies at compile-time rather than run-time.

The book then moves on to discuss the object-oriented features of the language, using numerous examples to illustrate the ideas of encapsulation, inheritance and polymorphism. A detailed case study of the design and implementation of a program using an object-oriented design methodology is included.

An introduction to the tasking features of Ada is included. Finally a text user interface API is developed to illustrate in a practical way the use of object-oriented components in a program. Several programs that use this interface are shown to illustrate the processes involved.

Exercises and self assessment questions are suggested for the reader at the end of each chapter to allow the reader to practise the use of the Ada components illustrated and to help reinforce, the reader's understanding of the material in the chapter. Answers to many of the practical exercises are given at the end of the book.

I would in particular like to thank Corinna for putting up with my many long hours in the 'computer room' and her many useful suggestions on the presentation and style used for the material in this book.

## Website

Support material for the book can be found on the Authors website :

<http://www.it.brighton.ac.uk/~mas>. The material consists of further solutions, source code, artwork and general information about Ada 95.

Michael A. Smith  
Brighton, May 2001

M.A.Smith@brighton.ac.uk

The example programs shown in this book use the following conventions:

Item in program	Example	Convention used
Attribute of an object or type	Integer'Last	Starts with an upper-case letter.
Class	<b>package</b> Class_cell <b>is</b> <b>type</b> Cell <b>is</b> <b>private</b>  <b>private</b>  <b>end</b> Class_cell;	Is declared as a package prefixed with the name 'Class_'. The class name is given to the private type that is then used to elaborate instances of the class.
Instance method: function or procedure	Display(The: <b>in</b> Cell)	The function or procedure is in lowercase and the first parameter passed to it is an instance of the class which is named the.
Instance attribute: a data item contained in an object.	Balance: Float;	Starts with an upper-case letter in the private part of the package.
Class attribute: a global data item that is shared between all instances of the class	The_Count: Integer;	Starts with The_ and is declared in the private part of the package.
Constant or enumeration	Max	Starts with an upper-case letter.
Function or procedure	Deposit	Starts with an upper-case letter..
Package	Pack_Account	Starts with 'Pack_'.
Formal parameter	Amount	Starts with an upper-case letter.
Protected type	<b>protected type</b> PT_Ex <b>is</b> <b>entry</b> Put(i: <b>in</b> T); <b>entry</b> get(i: <b>out</b> T); <b>end</b> PT_ex;	Starts with 'PT_'
Reserved word	<b>procedure</b>	Is in bold lower-case.
Task type	<b>task type</b> Task_Ex <b>is</b> <b>entry</b> Start; <b>end</b> Task_Ex;	Starts with 'Task_'.
Type or subtype	Colour	Starts with an upper-case letter.
Variable name	Mine P_Ch	Starts with an upper-case letter.. An access value for an item will start with 'P_'.

## 1.1 Glossary of terms used

Access type      A type used to elaborate an access value

Access value      The address of an object.

**Actual parameter** The physical object passed to a function, procedure, entry or generic unit. For example, in the following statement the actual parameter to the procedure Put is Number.

```
Print( Number );
```

**Ada 83** The version of the language that conforms to ANSI/MIL-STD 1815A ISO/IEC 8652:1983, 1983. Ada 83 is superseded by Ada 95. The language is named after Ada Augusta the Countess of Lovelace, daughter of the poet Lord Byron and Babbage's 'programmer'.

**Ada 95** The version of the language that conforms to ANSI/ISO/IEC 8652:1995, January 1995. The ISO standard was published on 15th February 1995. Ada 95 is now often referred to as Ada

**Ada class** In Ada the terminology class is used to describe a set of types. To avoid confusion this will be termed an Ada class.

**Allocator** An allocator is used to claim storage dynamically from a storage pool. For example, storage for an Integer is allocated dynamically with:

```
P_Int := new Integer;
```

**Base class** A class from which other classes are derived.

**Class** The specification of a type and the operations that are performed on an instance of the type. A class is used to create objects that share a common structure and behaviour.

The specification of a class Account is as follows:

```
package Class_Account is
  type Account is private;
  subtype Money is Float;

  function Balance ( The:in Account ) return Money;
  -- Other methods on an instance of an Account
private
  type Account is record
    Balance_Of : Money := 0.00; --Amount in account
  end record;
end Class_Account;
```



**Class attribute** A data component that is shared between all objects in the class. In effect it is a global variable which can only be accessed by methods in the class. A class attribute is declared in the private part of the package representing the class. For example, the class attribute `The_Interest_Rate` in the class `Interest_Account` is declared in the private part of the package as follows:

```
package Class_Interest_Account is
  type Interest_Account is private;

  procedure Set_Rate( Rate:in Float );

private
  type Interest_Account is new Account with record
    Balance_Of          : Money := 0.00;
    Accumulated_Interest : Money := 0.00;
  end record;

  The_Interest_Rate : Float := 0.00026116;
end Class_Interest_Account;
```

**Class method** A procedure or function in a class that only accesses class attributes. For example, the method `Set_Rate` in the class `Interest_Account` which sets the class attribute `The_Interest_Rate` is as follows:

```
procedure Set_Rate( Rate:in Float ) is
begin
  The_Interest_Rate := Rate;
end Set_Rate;
```

*Note: As `Set_Rate` is a class method an instance of the class is not passed to the procedure.*

**Controlled object** An object which has initialization, finalization and adjust actions defined. A limited controlled object only has initialization and finalization defined as assignment is prohibited.

**Discriminant** The declaration of an object may be parameterized with a value. The value is a discriminant to the type. For example, the declaration of `corinna` is parameterized with the length of her name.

```
type Person( Chs:Str_Range := 0 ) is record
  Name   : String( 1 .. Chs );
  Height : Height_Cm   := 0;
  Sex    : Gender;
end record;

Corinna : Person(7);
```

**Dynamic-binding** The binding between an object and the message that is sent to it is not known at compile-time.

**Elaboration** At run-time the elaboration of a declaration creates the storage for an object. For example:

```
Mike : Account;
```

creates storage at run-time for the object Mike.

**Encapsulation** The provision of a public interface to a hidden (private) collection of data procedures and functions that provide a coherent function

**Formal parameter** In a procedure, function, entry or generic unit the name of the item that has been passed. For example, in the procedure print shown below the formal parameter is Value.

```
procedure Print( Value:in Integer ) is
begin
  -- body
end print;
```

**Generic** A procedure, function or package which is parameterized with a type or types that are used in the body of the unit. The generic unit must first be instantiated as a specific instance before it can be used. For example, the package Integer\_Io in the package Ada.Text\_Io is parameterized with the integer type on which I/O is to be performed. This generic unit must be instantiated with a specific integer type before it can be used in a program.

**Inheritance** The derivation of a class (derived class) from an existing class (base class). The derived class will have the methods and instance/class attributes in the class plus the methods and instance/class attributes defined in the base class. In Ada this is called programming by extension. The class Interest\_Account that is derived from the class Account is specified as follows:

```
with Class_Account;
use Class_Account;
package Class_Interest_Account is

  type Interest_Account is new Account with private;

  procedure Set_Rate( Rate:in Float );
  procedure Calc_Interest( The:in out
                           Interest_Account );
private
  Daily_Interest_Rate: constant Float := 0.00026116;
  type Interest_Account is new Account with record
    Accumulated_Interest : Money := 0.00
  end record;
  The_Interest_Rate      : Float := 0.00026116;
end Class_Interest_Account;
```

**Instance attribute** A data component contained in an object. In Ada the data components are contained in a record structure in the private part of the package.

```
type Account is record
  Balance_Of : Money := 0.00; --Instance attribute
end record;
```

**Instance method** A procedure or function in a class that accesses the instance attributes (data items) contained in an object. For example, the method `Balance` accesses the instance attribute `Balance_Of`.

```
function Balance( The:in Account ) return Money is
begin
  return The.Balance_Of;
end Balance;
```

**Instantiation** The act of creating a specific instance of a generic unit. For example, the generic package `Integer_Io` in the package `Ada.Text_Io` can be instantiated to deliver the package `Pack_Mark_Io` which performs I/O on the integer type `Exam_Mark` as follows:

```
type Exam_Mark is range 0 .. 100;

package Pack_Mark_Io is new
  Ada.Text_Io.Integer_Io(Exam_Mark);
```

Then a programmer can write

```
Miranda : Exam_Mark;

Pack_Mark_Io.Put( Miranda );
```

to write the contents of the `Integer` object `Miranda`.

**Message** The sending of data values to a method that operates on an object. For example, the message 'deposit £30 in account Mike' is written in Ada as:

```
Deposit( Mike, 30 );
```

*Note: The object to which the message is sent is the first parameter.*

**Meta-class** An instance of a meta-class is a class. Meta-classes are not supported in Ada.

**Method** Implements behaviour in an object. A method is implemented as a procedure or function in a class. A method may be either a class method or an instance method.

**Multiple inheritance** A class derived from more than one base class. Multiple inheritance is not directly supported in Ada.

Object	An instance of a class. An object has a state that is interrogated / changed by methods in the class. The object <code>mike</code> that is an instance of <code>Account</code> is declared as follows:
	<pre>Mike: Account;</pre>
Overloading	When an identifier can have several different meanings. For example, the procedure <code>Put</code> in the package <code>Ada.Text_IO</code> has several different meanings. Output an instance of a <code>Character</code> , output an instance of a <code>String</code> .
	<pre>Put("Hello Worl"); Put('d' );</pre>
Overriding Polymorphism	The ability to send a message to an object whose type is not known at compile-time. The method selected depends on the type of the receiving object. For example the message <code>'Display'</code> is sent to different types of picture elements that are held in an array.
	<pre>Display( Picture_Element(I) );</pre>
Rendezvous	The interaction that occurs when two tasks meet to synchronize and possibly exchange information.
Representation clause	Directs the compiler to map a program item onto specific hardware features of a machine. For example, <code>location</code> is defined to be at address <code>16#046C#</code> .
	<pre>Mc_Address : constant Address :=     To_Address( 16#046C# );  Location : Integer; for Location'Address use Mc_Address;</pre>
Static binding	The binding between an object and the message that it is sent to it is known at compile-time.
Type	A type defines a set of values and the operations that may be performed on those values. For example, the type <code>Exam_Mark</code> defines the values that may be given for an exam in English.
	<pre>type Exam_Mark is range 0 .. 100;  English : Exam_Mark</pre>

*To my wife Corinna Lord, daughter Miranda and mother Margaret Smith*

*and guinea pig Delphi*



# 1 Introduction to programming

A computer programming language is used by a programmer to express the solution to a problem in terms that the computer system can understand. This chapter looks at how to solve a small problem using the computer programming language Ada 95.

## 1.1 Computer programming

Solving a problem by implementing the solution using a computer programming language is a meticulous process. In essence the problem is expressed in terms of a very stylized language in which every detail must be correct. However, this is a rewarding process both in the sense of achievement when the program is completed, and usually the eventual financial reward obtained for the effort.

Like the planet on which we live where there are many different natural languages, so the computer world also has many different programming languages. The programming language Ada 95 is just one of the many computer programming languages used today.

## 1.2 Programming languages

In the early days of computing circa 1950s, computer programs had to be written directly in the machine instructions of the computer. Soon assembly languages were introduced that allowed the programmer to write these instructions symbolically. An assembler program would then translate the programmer's symbolic instructions into the real machine code instructions of the computer. For example, to calculate the cost of a quantity of apples using an assembly language the following style of symbolic instructions would be written by a programmer:

```
LDA    AMOUNT_OF_OF_APPLES ; Load into the accumulator # pounds
MLT    PRICE_PER_POUND     ; Multiply by cost per pound of apples
STA    COST_OF_APPLES      ; Save result
```

*Note: Each assembly language instruction corresponds to a machine code instruction.*

In the period 1957—1958 the first versions of the high-level languages FORTRAN & COBOL were developed. In these high-level programming languages programmers could express many ideas in terms of the problem rather than in terms of the machine architecture. A compiler for the appropriate language would translate the programmer's high level statements into the specific machine code instructions of the target machine. Advantages of the use of a compiler include:

- Gains in programmer productivity as the solution is expressed in terms of the problem rather than in terms of the machine.
- If written correctly, programs may be compiled into the machine instructions of many different machines. Hence, the program may be moved between machines without having to be re-written.

For example, the same calculation to calculate the cost of apples is expressed in FORTRAN as:

```
COST = PRICE * AMOUNT
```

## 2 Introduction to programming

### 1.3 Range of programming languages

Since the early days of computer programming languages the number and range of high level languages has multiplied greatly. However, many languages have also effectively died through lack of use. A simplistic classification of the current paradigms in programming languages is shown in the table below:

Type of language	Brief characteristics of the language	Example
Functional	The problem is decomposed into individual functions. To a function is passed read only data values which the function transforms into a new value. A function itself may also be passed as a parameter to a function. As the input data to a function is unchanged individual functions may be executed simultaneously as soon as they have their input data.	ML
Logic	The problem is decomposed into rules specifying constraints about a world view of the problem.	Prolog
Object-oriented	The problem is decomposed into interacting objects. Each object encapsulates and hides methods that manipulate the hidden state of the object. A message sent to an object evokes the encapsulated method that then performs the requested task.	Ada 95 Eiffel Java Smalltalk
Procedural	The problem is decomposed into individual procedures or subroutines. This decomposition is usually done in a top down manner. In a top down approach, once a section of the problem has been identified as being implementable by a procedure, it too is broken down into individual procedures. The data however, is not usually part of this decomposition.	C Pascal

#### 1.3.1 Computer programming languages

A computer programming language is a special language in which a high level description of the solution to a problem is expressed. However, unlike a natural language, there can be no ambiguity or error in the description of the solution to the problem. The computer is unable to work out what was meant from an incorrect description. For example, in the programming language Ada 95, to print the result of multiplying 10 by 5 the following programming language statement is written:

```
Put( 10 * 5 );
```

To the non programmer this is not an immediately obvious way of expressing: print the answer to 10 multiplied by 5.

#### 1.3.2 The role of a compiler

The high-level language used to describe the solution to the problem, must first be converted to a form suitable for execution on the computer system. This conversion process is performed by a compiler. A compiler is a program that converts the high-level language statements into a form that a computer can obey. During the conversion process the compiler will tell the programmer about any syntax or semantic mistakes that have been made when expressing the problem in the high-level language. This process is akin to the work of a human translator who converts a document from English into French so that a French speaker can understand the contents of the document.

Once the computer program has been converted to a form that can be executed, it may then be run. It usually comes as a surprise to many new programmers that the results produced from running their program is not what



they expected. The computer obeys the programming language statements exactly. However, in their formulation the novice programmer has formulated a solution that does not solve the problem correctly.

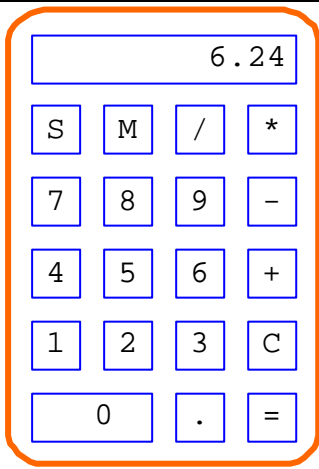
## 1.4 A small problem

A local orchard sells some of its rare variety apples in its local farm shop. However, the farm shop has no electric power and hence uses a set of scales which just give the weight of the purchased product. A customer buying apples, fills a bag full of apples and takes the apples to the shop assistant who weighs the apples to determine their weight in kilograms and then multiplies the weight by the price per kilogram.

If the shop assistant is good at mental arithmetic they can perform the calculation in their head, or if mental arithmetic is not their strong point they can use an alternative means of determining the cost of the apples.

## 1.5 Solving the problem using a calculator

For example, to solve the very simple problem of calculating the cost of 5.2 kilos of apples at £1.20 a kilo using a pocket calculator the following 4 steps are performed:

Pocket calculator	Step	Steps performed
	1	Enter the cost of a kilo of apples: C 1 . 2 0
	2	Enter the operation to be performed: *
	3	Enter the number of kilos to be bought: 5 . 2
	4	Enter calculate =

Note: The keys on the calculator are:

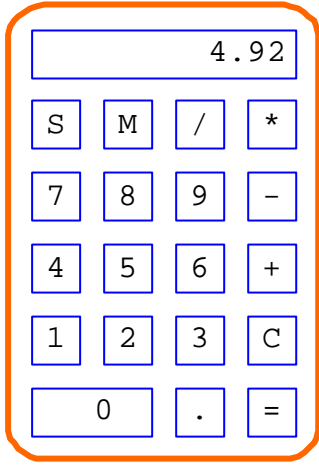
C	Clear the display and turn on the calculator if off		
S	Save the contents of the display into memory		
M	Retrieve the contents of the memory		
+ - */	Arithmetic operations		
*	Multiply	/	Division
+	plus	-	minus
=	Calculate		

When entered, these actions cause the calculation  $1.20 * 5.2$  to be evaluated and displayed. In solving the problem, the problem is broken down into several very simple steps. These steps are in the 'language' that the calculator understands. By obeying these simple instructions the calculator 'solves' the problem of the cost of 5.2 kilos of apples at £1.20 a kilo.

## 4 Introduction to programming

### 1.5.1 Making the solution more general

The calculation using the pocket calculator can be made more general by storing the price of the apples in the calculator's memory. The price of a specific amount of apples can then be calculated by retrieving the stored price of the apples and multiplying this retrieved amount by the quantity required. For example, to setup the price of apples in the calculator's memory and calculate the cost of 4.1 kilos of apples, the process is as follows:

Pocket calculator	Step	Steps performed
	1	Enter the cost of a kilo of apples: C 1 . 2 0
	2	Save this value to the calculator's memory: S
	3	Retrieve the value from memory: M
	4	Enter the operation to be performed: *
	5	Enter the number of kilos to be bought: 4 . 1
	6	Enter calculate =

To calculate the price for each customer's order of apples, only steps 3—6 need be repeated. In essence, a generalized solution to the problem of finding the price of any quantity of apples has been defined and implemented.

## 1.6 Solving the problem using the Ada 95 language

To solve the problem of calculating the cost of a quantity of apples using the programming language Ada 95, a similar process to that used previously when using a pocket calculator is followed. This time, however, the individual steps are as follows:

Step	Description
1	Set the memory location <code>Price_per_kilo</code> to the cost per kilogram of the apples.
2	Set the memory location <code>Kilos_of_apples</code> to the kilograms of apples required.
3	Set the memory location <code>Cost</code> to the result of multiplying the contents of memory location <code>Price_per_kilo</code> by the contents of the memory location <code>Kilos_of_apples</code> .
4	Print the contents of the memory location <code>Cost</code> .

*Note: Although a shorter sequence of steps can be written to calculate 1.2 multiplied by 5.2 the above solution can easily be extended to allow the price of any number of kilograms of apples to be calculated.*

In Ada 95 like most programming languages when a memory location is required to store a value, it must first be declared. This is done for many reasons, some of these reasons are:

- So that the type of items that are to be stored in this memory location can be specified. By specifying the type of the item that can be stored the compiler can allocate the correct amount of memory for the item as well as checking that a programmer does not accidentally try and store an inappropriate item into the memory location.

- The programmer does not accidentally store a value into a memory location `Cost` when they meant `Cost`. The programmer accidentally typed zero (0) when they meant the letter (o).

The sequence of steps written in pseudo English is transformed into the following individual Ada 95 statements which, when obeyed by a computer, will display the cost of 5.2 kilograms of apples at £1.20 a kilogram.

Step	Line	Ada 95 statements
	1	<code>Price_per_kilo : Float;</code>
	2	<code>Kilos_of_apples : Float;</code>
	3	<code>Cost : Float;</code>
1	4	<code>Price_per_kilo := 1.20;</code>
2	5	<code>Kilos_of_apples := 5.2;</code>
3	6	<code>Cost:= Price_per_kilo*Kilos_of_apples;</code>
4	7	<code>Put( Cost );</code>
	8	<code>New_Line;</code>

*Note: Words in bold type are reserved words in the Ada 95 language and cannot be used for the name of a memory location.*

*The name of the memory location contains the character \_ to make the name more readable. Spaces in the name of a memory location are not allowed.*

*Each Ada 95 statement is terminated with a ;.*

*Multiplication is written as \*.*

The individual lines of code of the Ada 95 program are responsible for the following actions:

Line	Description
1	Allocates a memory location called <code>Price_per_kilo</code> that is used to store the price per kilogram of apples. This memory location is of type <code>Float</code> and can hold any number that has decimal places.
2—3	Allocates memory locations: <code>Kilos_of_apples</code> and <code>Cost</code> .
4	Sets the contents of the memory location <code>Price_per_kilo</code> to 1.20. The <code>:=</code> can be read as 'is assigned the value'.
5	Assign 5.2 to memory location <code>Kilos_of_apples</code> .
6	Sets the contents of the memory location <code>Cost</code> to the contents of the memory location <code>Price_per_kilo</code> multiplied by the contents of the memory location <code>Kilos_of_apples</code> .
7	Writes the contents of the memory location <code>Cost</code> onto the computer screen.
8	Starts a new line on the computer screen.

This solution is very similar to the solution using the pocket calculator, except that individually named memory locations are used to hold the stored values, and the calculation is expressed in a more human readable form.

## 6 Introduction to programming

An animation of the above Ada 95 program is shown below. In the animation the contents of the memory locations are shown after each individual Ada 95 statement is executed. When a memory location is declared in Ada 95 inside a function its initial contents are undefined.

Ada 95 statements	price	kilos	Cost
Price_per_kilo : Float; Kilos_of_apples : Float; Cost : Float;	U	U	U
Price_per_kilo := 1.20;	1.20	U	U
Kilos_of_apples := 5.2;	1.20	5.2	U
Cost := Price_per_kilo * Kilos_of_apples;	1.20	5.2	6.24
Put( Cost );	1.20	5.2	6.24

Note: U indicates that the contents of the memory location are undefined.

Due to lack of room in the title column the variable *Price\_per\_kilo* is represented by *price* and *Kilos\_of\_apples* by *kilos*.

### 1.6.1 Running the program

The above lines of code, though not a complete Ada 95 program, form the core code for such a program. When this code is augmented with additional peripheral code, compiled and then run, the output produced will be of the form:

```
6.24
```

A person who knows what the program does, will instantly know that this represents the price of 5.2 kilograms of apples at £1.20 a kilogram. However, this will not be obvious to a casual user of the program.

### 1.7 The declare block

In Ada a declaration is separated from an executable statement. One way of expressing this split is the `declare` block that is specified as follows:

```
declare
  Cost : Float;
begin
  Cost := 5.2 * 1.20;
  Put( Cost );
end;
```

Note: By using this construct our Ada 95 programming statements are almost a complete program.

The section of a declare block are illustrated below in Figure 1.1

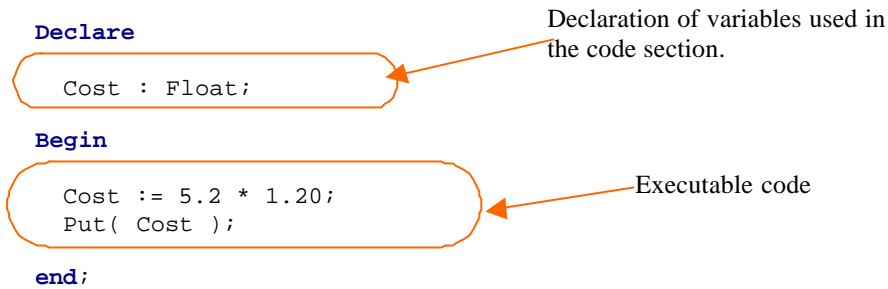


Figure 1.1 The declare block in Ada.

## 1.8 The role of comments

To make an Ada 95 program easier to read, comments may be placed in the program to aid the human reader of the program. A comment starts with `--` and extends to the end of the line. It is important however, to realize that the comments you write in a program are completely ignored by the computer when it comes to run your program. For example, the previous fragment of code could be annotated with comments as follows:

```

declare
    Price_Per_Kilo  : Float;           --Price of apples
    Kilos_Of_Apples : Float;           --Apples required
    Cost            : Float;           --Cost of apples
begin
    Price_Per_Kilo  := 1.20;           --Set cost £1.20
    Kilos_Of_Apples := 5.2;            --Kilos required

    Cost := Price_Per_Kilo * Kilos_Of_Apples; --Evaluate cost
    Put( Cost );                          --print the cost
    New_Line;                             --Print a new-line
end;

```

*Note: This is an example of comments, the more experienced programmer would probably miss out many of the above comments as the effect of the code is easily understandable.*

Comments that do not add to a reader's understanding of the program code should be avoided. In some circumstances the choice of meaning full names for memory locations is all that is required. As a general rule, if the effect of the code is not immediately obvious then a comment should be used to add clarity to the code fragment.

## 8 Introduction to programming

### 1.9 Summary

The statements in the Ada 95 programming language seen so far are illustrated in the table below:

Ada 95 statement/declaration	Description
<code>Cost : Float;</code>	Declare a memory location called <code>cost</code> .
<code>Cost := 1.2 * 5.2;</code>	Assign to the memory location <code>cost</code> the result of evaluating <code>1.2</code> multiplied by <code>5.2</code> .
<code>Put( "Hi!" );</code>	Print the message <code>Hi!</code> .
<code>Put( Cost ); New_Line;</code>	Print the contents of the memory location <code>Cost</code> followed by a newline.

Statements of this form allow a programmer to write many different and useful programs.

### 1.10 A more descriptive program

By adding additional Ada 95 statements, the output from a program can be made clear to all who use the program. For example, the program in Section 1.6 can be modified into the program illustrated below. In this program, a major part of the program's code is concerned with ensuring that the user is made aware of what the results mean.

Line	Ada 95 statements
1	<b>declare</b>
2	<code>Price_Per_Kilo : Float;</code> --Price of apples
3	<code>Kilos_Of_Apples: Float;</code> --Apples required
4	<code>Cost : Float;</code> --Cost of apples
5	<b>begin</b>
6	<code>Price_Per_Kilo := 1.20;</code>
7	<code>Kilos_of_apples := 5.2;</code>
8	<code>Cost := Price_per_kilo * Kilos_of_apples;</code>
9	<code>Put( "Cost of apples per kilo : " );</code>
10	<code>Put( Price_per_kilo );</code>
11	<code>New_Line;</code>
12	<code>Put( "Kilos of apples required K " );</code>
13	<code>Put( Kilos_of_apples );</code>
14	<code>New_Line;</code>
15	<code>Put( "Cost of apples               £ " );</code>
16	<code>Put( Cost );</code>
17	<code>New_Line;</code>
18	<b>end</b>

Line	Description
1	Start a declare block.
2—4	Declare the variables used in this fragment of code.
5	Begin the code section of the block.
6–8	Calculate the cost of 5.2 kilograms of apples at £1.20 per kilogram.
9	Displays the message <code>Cost of apples per kilo £</code> onto the computer screen. The double quotes around the text message are used to signify that this is a text message to be printed rather than the contents of a memory location.
10	Displays the contents of the memory location <code>Cost</code> onto the computer screen after the above message.
11	Starts a new line of output on the computer screen.
12—14	As for lines 7—9 but this time the message is <code>Kilos of apples required K</code> and the memory location printed is <code>Kilos_of_apples</code> .
15—17	As for lines 7—9 but this time the message is <code>Cost of apples £</code> and the memory location printed is <code>Cost</code> .
18	End the declare block

### 1.10.1 Running the new program

With the addition of some extra lines of code, the above program can be compiled and then run on a computer system. Once executed the following results will be displayed:

```
Cost of apples per kilo £ 1.2
Kilos of apples required K 5.2
Cost of apples £ 6.24
```

This makes it easy to see what the program has calculated.

### 1.11 Types of memory location

So far the type of the memory location used has been of type `Float`. A memory location of type `Float` can hold any number that has a fractional part. However, when such a value is held it is only held to a specific number of decimal places. Sometimes it is appropriate to hold numbers that have an exact whole value, e.g. a memory location `people` that represents the number of people in a room. In such a case the memory location should be declared to be of type `Integer`.

For example, the following fragment of code uses an `Integer` memory location to hold the number of people in a room.

```
Room : Integer;    -- Memory location
Room := 7;         -- Assigned the number 7
```

## 10 Introduction to programming

The choice of the type of memory location used, will of course depend on the values the memory location is required to hold. As a general rule, when an exact whole number is required, then a memory location of type Integer should be used and when the value may have a fractional part then a memory location of type Float should be used.

Memory location	Assignment to memory location
People : Integer	People := 2;
Weight : Float	Weight := 7.52;

### 1.11.1 Warning

Ada 95 will not allow assignments or expressions that mix different types of memory locations or numbers. In particular this means that you cannot assign a number with decimal places or implied decimal places to a location that holds an integer value. Likewise, you cannot assign a whole number to a memory location that holds a number with potential decimal places.

For example, the following assignment is invalid:

Memory location	Invalid assignment	Reason
People: Integer	People := 2.1;	You cannot assign a number with a fractional part to a memory location of type Integer.
Weight : Float	Weight := 2;	You cannot assign an whole number to a location that holds a result with implied decimal places.
	People := people + weight;	The right hand side mixes whole numbers and numbers with decimal places.

The reason for this initially rather severe restriction is to help prevent programming errors go undetected. For example, if you accidentally stored a number with decimal places into a location that only contained a whole number then the resultant loss of precision may result in an error in the logic of the program.

## 1.12 Repetition

So far, all the Ada 95 programs used in the examples have used straight line code. In straight line code the program consists of statements that are obeyed one after another from top to bottom. There are no statements that affect the flow of control in the program. This technique has allowed us to produce a solution for the specific case of the cost of 5.2 kilograms of apples at £1.20 per kilogram.

Using this strategy, to produce a program to list the cost of apples for a series of different weights would effectively involve writing out the same code many times. An example of this style of coding is illustrated below:

```
declare
  Price_Per_Kilo : Float;      --Price of apples
  Kilos_Of_Apples: Float;     --Apples required
  Cost           : Float;     --Cost of apples
begin
  Price_Per_Kilo := 1.20;

  Put( "Cost of apples per kilo  : " );
  Put( Price_Per_Kilo );
  New_Line;

  Put( "Kilo's Cost" );
  New_Line;
```

```
Kilos_Of_Apples      := 0.1;
```



```

Cost := Price_Per_Kilo * Kilos_Of_Apples;
Put( Kilos_Of_Apples );
Put( "      " );
Put( Cost );
New_Line;

```

```

Kilos_Of_Apples      := 0.2;

```

```

Cost := Price_Per_Kilo * Kilos_Of_Apples;
Put( Kilos_Of_Apples );
Put( "      " );
Put( Cost );
New_Line;

```

etc.

```

end;

```

Whilst this is a feasible solution, if we want to calculate the cost of 100 different weights this will involve considerable effort and code. Even using copy and paste operations in an editor to lessen the typing effort, will still involve considerable effort! In addition, the resultant program will be large and consume considerable resources.

## 1.13 Introduction to the while statement

In Ada 95 a while statement is used to repeat program statements while a condition holds true. A while statement can be likened to a rail track as illustrated in Figure 1.2. While the condition is true the flow of control is along the true track. Each time around the loop the condition is re-evaluated. Then, when the condition is found to be false, the false track is taken.

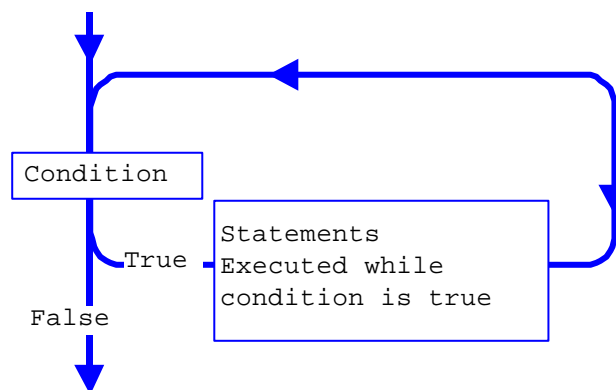


Figure 1.2 The while statement as a rail track.

In a while loop the condition is always tested first. Due to this requirement if the condition initially evaluates to false then the code associated with the while loop will never be executed.

## 12 Introduction to programming

### 1.13.1 Conditions

In the language Ada 95, a condition is expressed in a very concise format which at first sight may seem strange if you are not used to a mathematical notation. For example, the conditional expression: 'the contents of the memory location count is less than or equal to 5' is written as follows:

```
count <= 5
```

*Note:* The memory location named `count` will need to be declared as:

```
count : Integer;
```

The symbols used in a condition are as follows:

Symbol	Means	Symbol	Means
<	Less than	<=	Less than or equal to
=	Equal to	/=	Not equal to
>	Greater than	>=	Greater than or equal to

If the following memory locations contain the following values:

Memory location	Assigned the value
Temperature : Integer;	Temperature := 15;
Weight : Float;	Weight := 50.0;

then the following table shows the truth or otherwise of several conditional expressions written in Ada 95.

In English	In Ada 95	Condition is
The temperature is less than 20	Temperature < 20	true
The temperature is equal to 20	Temperature = 20	false
The weight is greater than or equal to 30	Weight >= 30.0	true
20 is less than the temperature	20 < Temperature	false

*Note:* As a memory location that holds a `Float` value represents a number that is held only to a certain number of digits accuracy, it is not a good idea to compare such a value for equality = or not equality /=.

### 1.13.2 A while statement in Ada 95

Illustrated below is a fragment of code that uses a `while` statement to write out the text message `Hello` five times:

```
declare
  Count : Integer;
begin
  Count := 1;           --Set count to 1

  while Count <= 5 loop  --While count less than or equal 5
    Put( "Hello" );      --Print Hello
    New_Line;
    Count := Count + 1;  --Add 1 to count
  end loop;
end;
```

*Note:* The statement: `Count := Count + 1;` adds 1 to the contents of `Count` and puts the result back into the memory location `Count`.

In this code fragment, the statements between `loop` and `end loop;` are repeatedly executed while the contents of `Count` are less than or equal to 5. The flow of control for the above `while` statement is illustrated in Figure 1.3.

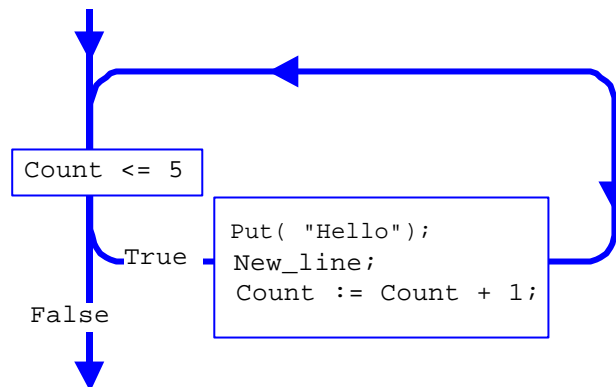


Figure 1.3 Flow of control for a `while` statement in Ada 95.

### 1.13.3 Using the `while` statement

The real advantage of using a computer program accrues when the written code is repeated many times, thus saving the implementor considerable time and effort. For example, if we wished to produce a table representing the cost of different weights of apples, then a computer program is constructed that repeats the lines of Ada 95 code that evaluate the cost of a specific weight of apples. However, for each iteration of the calculation the memory location that contains the weight of the apples is changed. A fragment of Ada 95 code to implement this solution is illustrated below:

```

declare
  Price_Per_Kilo : Float;           --Price of apples
  Kilos_Of_Apples: Float;           --Apples required
  Cost           : Float;           --Cost of apples
begin
  Price_Per_Kilo := 1.20;

  Put( "Cost of apples per kilo : " );
  Put( Price_Per_Kilo ); New_Line;

  Put( "Kilo's Cost" ); New_Line;

  Kilos_Of_Apples := 0.1;

  while Kilos_Of_Apples <= 10.0 loop  --While lines to print
    Cost := Price_Per_Kilo * Kilos_Of_Apples; --Calculate cost
    Put( Kilos_Of_Apples );               --Print results
    Put( "      " );
    Put( Cost );
    New_Line;
    Kilos_Of_Apples := Kilos_Of_Apples + 0.1; --Next value
  end loop;
end;

```

## 14 Introduction to programming

which when compiled with suitable peripheral code produces output of the form:

```
Cost of apples per kilo : 1.20
Kilo's   Cost
0.1      0.12
0.2      0.24
0.3      0.36
0.4      0.48
0.5      0.60
0.6      0.72
0.7      0.84
0.8      0.96
0.9      1.08
1.0      1.12
1.1      1.32
1.2      1.44
1.3      1.56
...
9.9      11.88
10.0     12.00
```

*Note:* Using `Put(Price_Per_Kilo)`, `Put(Kilos_Of_Apples)` and `Put(Price)` will cause the value to be output in scientific notation. To get the effect of the format shown above the `Put` statements would need to be changed to:

```
Put(Price_Per_Kilo)  -> Put(Price_Per_Kilo,Exp=>0,Aft=>2)
Put(Kilos_Of_Apples) -> Put(Kilos_Of_Apples,Exp=>0,Aft=>2)
Put(Cost)            -> Put(Cost,Exp=>0,Aft=>2).
```

*This is fully explained in Section 4.6.1.*

### 1.14 Selection

The `if` construct is used to conditionally execute a statement or statements depending on the truth of a condition. This statement can be likened to the rail track illustrated in Figure 1.4 in which the path taken depends on the truth of a condition. However, unlike the `while` statement there is no loop back to re-execute the condition.

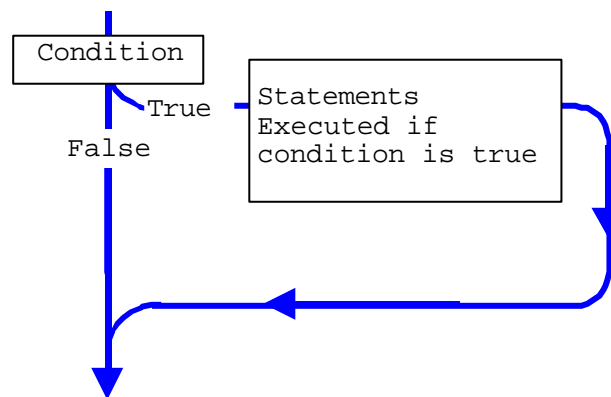


Figure 1.4 The `if` statement represented as a rail track.

For example, the following fragment of an Ada 95 program only prints out Hot! when the contents of the memory location Temperature are greater than 30.

```

declare
  Temperature : Integer;
begin
  Temperature := 30;
  if Temperature > 30 then      --If temperature greater than 30
    Put( "Hot!" );              --Say its hot
    New_Line;
  end if;
end;

```

In this code fragment, the statements between then and end if; are only executed if the condition Temperature > 30 is true. The flow of control for the above fragment of code is illustrated in Figure 1.5.

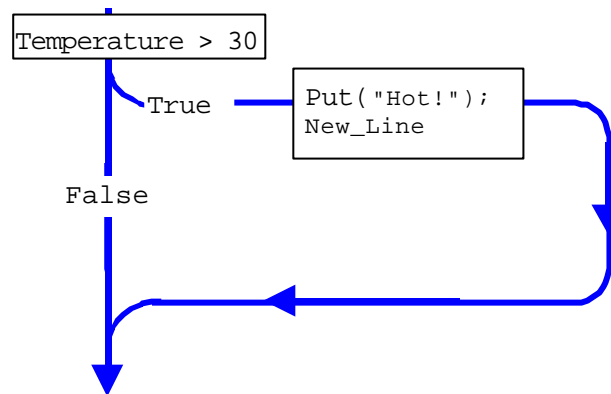


Figure 1.5 The if statement represented as a rail track.

### 1.14.1 Using the if statement

The fragment of program code which was used earlier to tabulate a list of the price of different weights of apples can be made more readable by separating every 5 lines by a blank line. This can be achieved by having a counter count to count the number of lines printed and after the 5th line has been printed to insert a blank line. After a blank line has been printed the counter count is reset to 0. This modified program is shown below:

```

declare
  Price_Per_Kilo   : Float := 1.20;
  Kilos_Of_Apples  : Float := 0.0;
  Cost             : Float;
  Lines_Output     : Integer := 0;
begin
  Put( "Cost of apples per kilo : " );
  Put( Price_Per_Kilo ); New_Line;
  Put( "Kilo's Cost" ); New_Line;
  while Kilos_Of_Apples <= 10.0 loop      --While lines to print
    Cost := Price_Per_Kilo * Kilos_Of_Apples; --Calculate cost
    Put( Kilos_Of_Apples );               --Print results
    Put( "      " );
    Put( Cost );
    New_Line;
    Kilos_Of_Apples := Kilos_Of_Apples + 0.1; --Next value
    Lines_Output := Lines_Output + 1;         --Add 1
    if Lines_Output >= 5 then                --If printed group
      New_Line;                             -- Print line
      Lines_Output := 0;                    -- Reset count
    end if;
  end loop;
end;

```

## 16 Introduction to programming

which when compiled with additional statements would produce output of the form shown below:

```
Cost of apples per kilo : 1.20
Kilo's Cost
0.0      0.00
0.1      0.12
0.2      0.24
0.3      0.36
0.4      0.48

0.5      0.60
0.6      0.72
0.7      0.84
0.8      0.96
0.9      1.08

1.0      1.20
1.1      1.32
1.2      1.44
1.3      1.56
1.4      1.68

etc.
```

*Note:* Using `Put(Price_Per_Kilo)`, `Put(Kilos_Of_Apples)` and `Put(Price)` will cause the value to be output in scientific notation. To get the effect of the format shown above the `Put` statements would need to be changed to:

```
Put(Price_Per_Kilo)  -> Put(Price_Per_Kilo,Exp=>0,Aft=>2)
Put(Kilos_Of_Apples) -> Put(Kilos_Of_Apples,Exp=>0,Aft=>2)
Put(Cost)            -> Put(Cost,Exp=>0,Aft=>2).
```

*This is fully explained in Section 4.6.1.*

### 1.15 Self-assessment

- What is a computer programming language?
- What do the following fragments of Ada 95 code do?

```
declare
  I : Integer;
begin
  I := 10;
  while I > 0 loop
    Put( I );
    I := I - 1;
  end loop;
  New_Line;
end;
```

```
declare
  Temperature : Integer;
begin
  Temperature := 10;
  if Temperature > 20 then
    Put( "It's Hot!" );
  end if;
  if Temperature <= 20 then
    Put( "It's not so Hot!" );
  end if;
  New_Line;
end;
```

- Write an Ada 95 fragment of code for the following conditions. In your answer show how any memory location you have used has been declared.
  - The temperature is less than 15 degrees centigrade.
  - The distance to college is less than 15 kilometres.
  - The distance to college is greater than or equal to the distance to the football ground.
  - The cost of the bike is less than or equal to the cost of the hi-fi system.

## 1.16 Paper exercises

Write down on paper Ada 95 statements to implement the following. You do not need to run these solutions.

- *Name*  
Write out your name and address.
- *Weight*  
Calculate the total weight of 27 boxes of paper. Each box of paper weighs 2.4 kilograms.
- *Name*  
Write out the text message "Happy Birthday" 3 times using a while loop.
- *Times table*  
Print the 7 times table. The output should be of the form:  

$$\begin{array}{rcl} 7 * 1 & = & 7 \\ 7 * 2 & = & 14 \\ \text{etc.} \end{array}$$

*Hint: Write the Ada 95 code to print the line for the 3rd row, use a variable row of type Integer to hold the value 3.*

$$7 * 3 = 21$$

*Enclose these statements in a loop that varies the contents of row from 1 to 12.*

- *Weight table*  
Print a table listing the weights of 1 to 20 boxes of paper, when each box weighs 2.4 kilograms.
- *Times table*  
Print a multiplication table for all values between 1 and 5. The table to look like:

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

*Hint: Write the Ada 95 code to print the line for the 2nd row, use a variable row of type Integer to hold the value 2.*

$$2 \mid 2 \quad 4 \quad 6 \quad 8 \quad 10$$

*Enclose these statements in a loop that varies the contents of row from 1 to 5. Add statements to print the heading:*

	1	2	3	4	5
--	---	---	---	---	---

## 2 Software design

This chapter looks at software production in the large. In particular it looks at problems that occur in the development of large and not so large software systems. The notation used by UML (Unified Modelling Language) is introduced as a mechanism for documenting and describing a solution to a problem that is to be implemented on a computer system.

### 2.1 The software crisis

In the early days of computing, it was the hardware that was very expensive. The programs that ran on these computers were by today's standards incredibly small. In those distant times computers only had a very limited amount of storage; both random access memory and disk storage.

Then it all changed. Advances in technology enabled computers to be built cheaper, with a far greater capacity than previous machines. Software developers thought, "Great! We can build bigger and more comprehensive programs". Software projects were started with an increase in scope and great optimism.

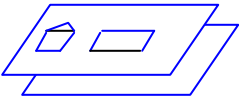
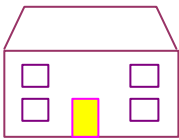
Soon, with projects running over budget and not meeting their client's expectations, the truth dawned: large scale software construction is difficult. The early techniques that had been used in small scale software construction did not scale up successfully for large scale software production.

This can be likened to using a bicycle to travel a short distance. Whilst this is adequate for the purpose, the use of a bicycle is inappropriate if a long distance has to be travelled in a short space of time. You cannot just peddle faster and faster.

### 2.2 A problem, the model and the solution

In implementing any solution to a problem, we must first understand the problem that is to be solved. Then, when we understand the problem fully, a solution can be formulated.

There are many different ways of achieving an understanding of a problem and its solution. Usually, this involves modelling the problem and its solution using either a standard notation or a notation invented by the programmer. The advantage of using a standard notation is that other people may inspect and modify the description of the problem and its proposed solution. For example, in building a house, an architect will draw up a plan of the various components that are to be built. The client can view the plans and give their approval or qualified approval subject to minor modifications. The builders can then use the plan when they erect the house.

Architect's plan (model)	Finished house
	

Writing a computer program involves the same overall process. First, we need to understand the task that the computer program will perform. Then we need to implement a solution using the model that we have created.

An easy pitfall at this point is to believe that the model used for the solution of a small problem can be scaled up to solve a large problem. For example, to cross a small stream we can put a log over the stream or if athletic we can even jump over the stream. This approach to crossing a stream however, will not scale up to crossing a large river. Likewise to build a 100-storey tower block, an architect would not simply take the plans for a 2-storey house and instruct the builders to build some extra floors.



In software the same problems of scale exist; the techniques that we use to implement a small program cannot usually be successfully used on a large programming project. The computer literature is full of examples of software disasters that have occurred when a computer system has been started without a full understanding of the problem that is to be solved.

### 2.2.1 Responsibilities

Since our earliest days we have all been told that we have responsibilities. Initially, these responsibilities are very simple, but as we grow older so they increase. A responsibility is a charge, trust or duty of care to look after something. At an early age this can be as simple as keeping our room neat and tidy. In later life, the range and complexity of items that we have responsibility for, increases dramatically.

A student for example, has the responsibility to follow a course of study. The lecturer has the responsibility of delivering the course to the students in a clear and intelligible manner. The responsibilities of the student and lecturer are summarized in tabular form below:

Responsibilities of a student	Responsibilities of a lecturer
Follow the course of study.	Deliver the course.
Perform to the best of their ability in the exam/assessment for the course.	Set and mark the assessment for the course.
	Attend the exam board for the delivered course.

Software too has responsibilities. For example, a text editor has the responsibility of entering the user's typed text correctly into a document. However, if the text that is entered into the text editor is incorrect or meaningless, then the resultant document will also be incorrect. It is not the role of the text editor to make overall decisions about the validity of the entered text.

In early computing literature, a common saying was "Garbage in, garbage out". Even though the software package implements its responsibilities correctly, the results produced may be at least meaningless, at worse damaging if used.

## 2.3 Objects

The world we live in is composed of many different objects. For example, a person usually has access to at least some of the following objects:

- A telephone.
- A computer.
- A car.

Each object has its own individual responsibilities. For example, some of the responsibilities associated with the above objects are:

Object	Responsibilities
Telephone	<ul style="list-style-type: none"> <li>● Establish contact with another phone point.</li> <li>● Convert sound to/from electrical signals.</li> </ul>
Computer	<ul style="list-style-type: none"> <li>● Execute programs.</li> <li>● Provide a tcp/ip connection to the internet.</li> </ul>
Car	<ul style="list-style-type: none"> <li>● Move</li> <li>● Go faster/slower</li> <li>● Turn left/right</li> <li>● Stop.</li> </ul>

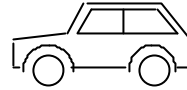
A responsibility here, is a process that the object performs. For example, a car can move forwards or backwards. However, the car has to be instructed by the driver to perform this task. The object is passive, and only performs an action when instructed to do so.

## 20 *Software design*

### 2.3.1 The car as an object

A car is made up of many components or objects. From a user's perspective some of the major objects that make up a car are:

- The shell or body of the car.
- The engine.
- The gearbox.
- The clutch.
- The battery that provides electric power.



We can think of the body or shell of the car as a container for all the other objects, that when combined, form a working car. These other objects are hidden from the driver of the car. The driver can, however, interact with these objects by using the external interfaces that form part of the car shell. This arrangement of objects is expressed diagrammatically using the UML notation in Figure 2.1.

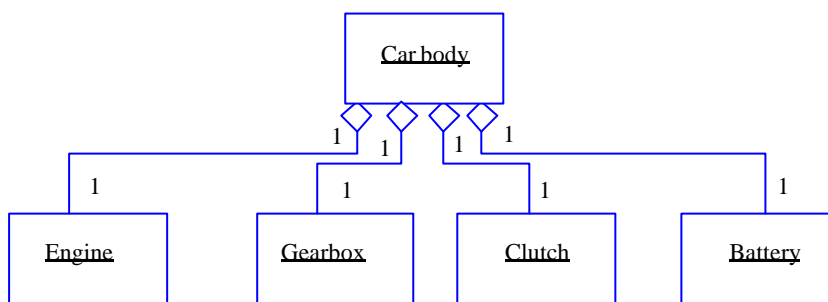


Figure 2.1 Objects that make up a car.

In Figure 2.1 the following style of notation is used:

	Denotes an object. In this specific case the car engine.
	Denotes aggregation. The engine contains 4 pistons. Note:  Denotes aggregation, the component B is contained in the container A.  Denotes composition, in addition the component B is created and destroyed by the container A.


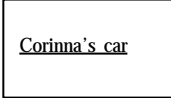
By using this notation, we can express the 'part of' relationship between objects. The engine, gearbox, clutch and battery are 'part of' a car.

## 2.4 The class

In object-oriented terminology a class is used to describe all objects that share the same responsibilities and internal structure. A class is essentially the collective name for a group of like objects. For example, the following objects all belong to the class car:

Corinna's red car	Mike's silver car	Paul's blue car
		

Although the objects differ in detail, they all have the same internal structure and responsibilities. Each object is an instance of the class `Car`. The notation for a class is slightly different from that of an object. The UML notation for a class and an object are illustrated below:

A class	An object (an instance of a class)
	

*Note: The name of the object is underlined.*

It is important to distinguish between a class and an object. A very simple rule is that objects usually have a physical representation, whereas classes are an abstract concept.

## 2.5 Methods and messages

A method implements a responsibility for a class. For example, some of the responsibilities for the class `Car` are as follows.

### Responsibilities of the class `Car`

- Start/stop engine
- Go faster/slower
- Turn left/right
- Stop.



An instance of the class `Car` is an object. By sending a message to the object a hidden method inside the object (a responsibility of the class `Car`) is invoked to process the message. For example, the driver of the car by pressing down on the accelerator, sends the message 'go faster'. The implementation of this is for the engine control system to feed more petrol to the engine. Normally however, the details of this operation are not of concern to the driver of the car.

## 2.6 Class objects

We have looked at a car's shell as a container for objects and can look at a laptop computer as a container for several computing devices or objects. A laptop computer is composed of:

- The shell of the laptop, that has external interfaces of a keyboard, touch pad and display screen.
- The local disk drive.
- The network file system.
- The CPU.
- The sound and graphics chipset.

## 22 Software design

In this analysis, the networked file system is shared between many different laptops, each individual laptop having access to the networked file system. In object-oriented terminology the networked file system is a class object which is shared between all the notebooks.

The concept of a shared object is important as it allows all instances of a class to have access to the same information. Thus, if one instance of a laptop computer creates a file on the network file system, the other notebooks will be able to access the contents of this file.

This arrangement of objects for a laptop computer can be expressed diagrammatically as illustrated using the UML notation in Figure 2.2. Unfortunately in UML there is no way to show diagrammatically that a class item is shared between many classes.

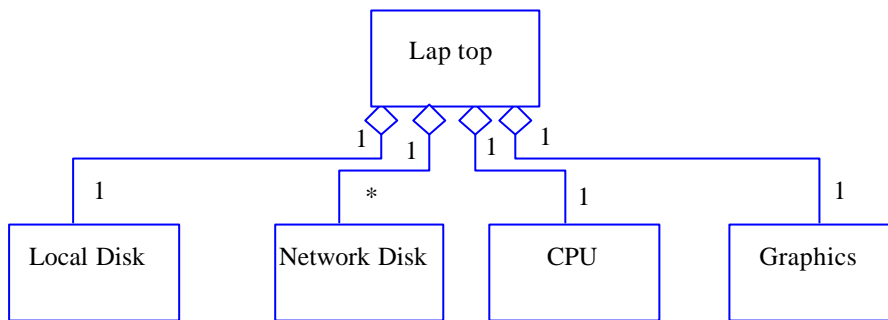


Figure 2.2 Objects that make up a laptop computer from a user's perspective.

Another interesting property of a class object, is that to access it you do not need an instance of the container object. For example, the network file system can be used by devices other than the laptop computers.

## 2.7 Inheritance

A typical office will usually contain at least the following objects:

- A telephone.
- A fax machine with a telephone hand set.
- A computer.

Each of these objects has their own individual responsibilities. For example, some of the responsibilities of these office objects are:

Object	Responsibilities
Telephone	<ul style="list-style-type: none"><li>• Establish contact with another phone point.</li><li>• Convert sound to/from electrical signals.</li></ul>
Fax machine with a telephone hand set	<ul style="list-style-type: none"><li>• Establish contact with another phone point.</li><li>• Convert sound to/from electrical signals.</li><li>• Convert images to/from electrical signals.</li></ul>
Computer	<ul style="list-style-type: none"><li>• Execute programs.</li><li>• Provide a tcp/ip connection to the internet.</li></ul>

Looking at these responsibilities shows that the telephone and fax machine share several responsibilities. The fax machine has two of the responsibilities that the telephone has. We could say that a fax machine is a telephone that can also send and receive images. Another way of thinking about this is that the fax machine can be used as if it were only a telephone. This relationship between classes that represent all telephones and fax machines is shown diagrammatically in Figure 2.3 using the UML notation. In this relationship a fax machine is inherited (or formed from the components) of a telephone.

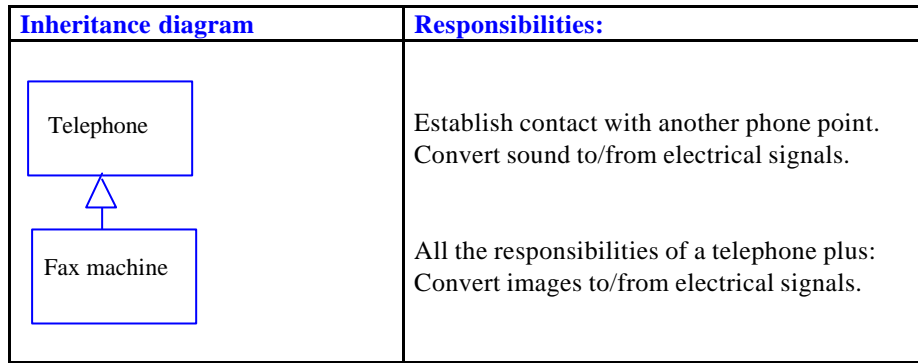


Figure 2.3 Relationship between a telephone and a fax machine.

*Note:* The superclass (telephone) is the class from which a subclass (fax machine) is inherited. Inheritance requires you to take all the responsibilities from the superclass; you cannot selectively choose to take only some. However, even though you inherit the responsibilities you do not need to use them.

The inheritance relationship is an important concept in object-oriented programming as it enables new objects to be created by specializing an existing object. In creating the new object, only the addition, responsibilities have to be constructed. The development time for a new software object is reduced as the task of creating the inherited responsibilities has already been done. This process leads to a dramatic reduction in the time and effort required to create new software objects.

## 2.8 Polymorphism

In a collection of different objects if all the objects are capable of receiving a specific message then this message may be sent to any object in the collection. The method executed when this message is received by an object will depend on the type of the object that receives the message.

For example, in a group of individuals if you ask a person how to take part in their favourite sport, you will probably get many different answers. In effect the message ‘How to take part in your favourite sport’ is polymorphic in that the answer you get depends on the individual person you select to ask. A tennis player for example, would give a different answer than a golfer.

## 2.9 Self-assessment

- Explain why the solution to a small problem may not always scale up to solve a much larger and complex problem.
- What is a “Responsibility”?
- What are the responsibilities of:
  - A video camera.
  - An alarm clock.
  - A traffic light.
  - An actress playing the role of Olga in the *Three sisters* by Chekov.
- What is the relationship between an object, message and a method?

## 24 *Software design*

- What classes do the following objects belong to?

apartment	cat	crayon	crystal	dog
guinea pig	igloo	house	ink pen	library
mansion	office block	pencil	rabbit	sheep

Identify which classes are subclassed from other classes?

- Identify several objects and classes around you at the moment. Can you find responsibilities that any of the objects or classes have in common?

## 3 Ada introduction: Part 1

This chapter looks at some simple Ada programs, and presents the basic control structures of the language. The data types `Integer` and `Character` are used to introduce these structures.

### 3.1 A first Ada program

The first program presented is a simple one that writes the message 'Hello World' onto the user's terminal.

```
with Text_IO;  
use Text_IO;  
procedure Hello is  
begin  
    Put("Hello World"); New_Line;  
end Hello;
```

*Note:* The example programs in this book are shown with reserved words in bold to aid readability. As the name suggests, reserved words can only be used for their intended purpose. Strange error messages can occur when a reserved word is inadvertently used by the programmer as the name of an object in a program. Reserved words are entered as normal text when writing a program. Section B.1, Appendix B lists all the reserved words in the Ada programming language.

When compiled and run, this program will display on a user's terminal the message:

```
Hello World
```

In the above program, the reserved words **begin** and **end** are used to bracket the body of the procedure `Hello`. In Ada, a procedure can be a self-contained program unit that may be independently compiled. In the above example, the single procedure `Hello` forms a complete program that may be compiled and run by an appropriate Ada compiling system.

The statement `Put("Hello World");` is responsible for outputting the greeting to the terminal. Used in conjunction with `New_Line`, which outputs a new line character to the terminal, these procedures are defined and implemented by the library package `Ada.Text_IO`.

*Note:* The **end** keyword is followed by the name of the procedure, in this example `helloworld`. The compiler checks for this to ensure that the procedure's extent agrees with the programmer's view.

One of the important concepts in Ada is the idea of encapsulating items together to form a package which may be re-used in other programs. The library package `Ada.Text_IO` is provided on Ada systems to allow the input and output of textual information to and from the user's program. This library package is introduced to a procedure by means of the statements **with** `Ada.Text_IO;` **use** `Ada.Text_IO;` the details of which will be explained later.

## 26 Ada introduction: Part 1

Figure 3.1 illustrates the components of an Ada program.

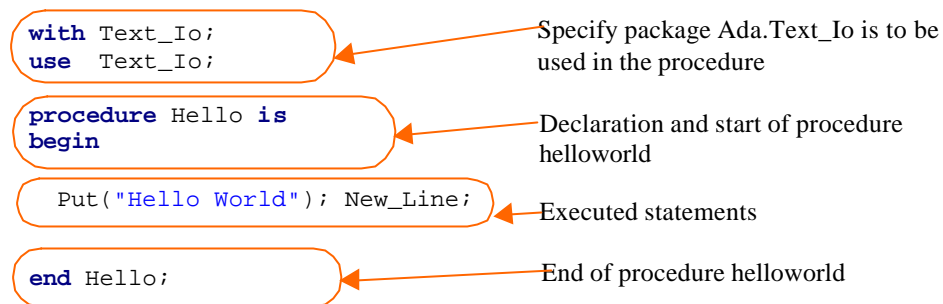


Figure 3.1 Components of an Ada program.

### 3.2 The case of identifiers in a program

In Ada, the case of characters used in reserved words and identifiers is unimportant; the compiler will take `begin`, `Begin` or even `BeGiN` to mean exactly the same thing. The only place where the case of a letter matters is in string and character constants. For example, the above program, could have been written as:

```
With Text_IO;
Use Text_IO;
Procedure HELLO is
Begin
  PUT("Hello World"); NEW_LINE;
End hello;
```

### 3.3 Format of an Ada program

In Ada, like in many other languages, white space is used mainly as a way of neatly laying out a program so that everyone, including the author, may clearly see the structure and purpose of the statements. There are many conventions for the layout of an Ada program and there are even programs which will reformat an Ada program for you.

The program illustrated above could have been written as:

```
with Text_Io; use Text_Io; procedure Hello is begin
Put("Hello World"); New_Line; end Hello;
```

although it is now more difficult to see exactly what the code is supposed to achieve. The only place where white space character(s) or a new line are needed, is between words that contain alphabetic characters. Naturally, any strings that contain white space characters will be output containing these white space characters. However a newline character is not allowed in a string literal.

A line of an Ada program can be up to 200 characters long and implementors may, if they wish, allow longer lines. A consequence of this is that names of items in an Ada program are considered unique if the first 200 characters are different.

#### 3.3.1 Variable names

A variable name must start with a letter and can then be followed by any number of letters and digits and the underscore character. However, two underscore characters cannot occur next to each other and an underscore character must not start or finish a variable name.



### 3.3.2 Comments

In Ada, comments may be introduced into a program by preceding the remainder of the line containing the comment, with two - characters. For example, a possible comment to the above program might be:

```
-- This program writes the message
-- "Hello World" to a users terminal
```

## 3.4 A larger Ada program

A program to produce a countdown is shown below. In this program, various constructs which affect the flow of control are introduced.

```
with Ada.Text_IO;           --With package Text_Io
use  Ada.Text_IO;           --Use  components
procedure Main is
  Count : Integer;          --Declaration of count
begin
  Count := 10;              --Set to 10
  while Count > 0 loop      --loop while greater than 0
    if Count = 3 then      --If 3 print Ignition
      Put("Ignition"); New_Line;
    end if;
    Put( Integer'Image( Count ) ); --Print current count
    New_Line;
    Count := Count - 1;    --Decrement by 1 count
    delay 1.0;             --Wait 1 second
  end loop;
  Put("Blast off"); New_Line; --Print Blast off
end Main;
```

In this program an integer variable `count` is declared which contains the current value of the countdown. This is achieved with the declaration `Count : Integer;`. The Ada statement `delay 1.0;` causes a pause of one second in the program.

*Note: Declarations of items are allowed in any order in Ada 95.  
 Integer'Image( Count ) delivers count as a character string. This is necessary as the package Ada.Text\_IO only implements input and output on a character or a string.*

When run, this program will produce the following output:

```
10
9
8
7
6
5
4
Ignition
3
2
1
Blast off
```

This is similar to the commentary used during the take-off procedures of early space missions.

## 28 Ada introduction: Part 1

### 3.5 Repetition: while

```
while Count > 0 loop                --loop while greater than 0
    -- Repeated statements
end loop;
```

The above construct repeatedly executes the statements between **loop** and **end loop** while the condition `count > 0` is true.

*Note: The mandatory **end loop** terminates the **while loop**. In Ada, most constructs are terminated by a mandatory termination keyword(s). This prevents the kind of errors that can occur in other languages when an extra statement is added in the belief that it forms part of the construct. It also allows the compiler to check that the user has constructed a program correctly by matching the start and end of each construct.*

### 3.6 Selection: if

```
if Count = 3 then                  --If 3 print Ignition
    Put("Ignition"); New_Line;
end if;
```

The **if** statement allows a statement or statements to be executed only if the condition is true. In the above example, the statements `Put("Ignition"); New_Line;` will only be executed when `count` is equal to 3.

*Note: The mandatory **end if** terminates the **if** statement.*

An **else** part may also be included, in which case statement or statements which follow it are only obeyed if the condition is false. For example:

```
if Count = 3 then
    Put("Count is 3"); New_Line;
else
    Put("Count is not 3"); New_Line;
end if;
```

*Note: The else part of an **if** statement is optional. However, if it is included it must be followed by at least one statement.*

The rather inelegant nested **if** structure below:

```
if Count = 3 then
    Put("Count is 3"); New_Line;
else
    if Count = 4 then
        Put("Count is 4"); New_Line;
    else
        Put("Count is not 3 or 4"); New_Line;
    end if;
end if;
```

can be rewritten using the following **elsif** construct:

```
if Count = 3 then
  Put("Count is 3"); New_Line;
elsif Count = 4 then
  Put("Count is 4"); New_Line;
else
  Put("Count is not 3 or 4"); New_Line;
end if;
```

*Note:* For the statements in the **else** part to be obeyed, all the conditions in the **if** and **elsif** parts must be false.  
There may be many **elsif** components in an **if** statement, but only one **else**.

## 3.7 Other repetition constructs

### 3.7.1 for

In Ada, a loop may be constructed in which a variable is varied by one unit between two values. For example, the code to print out the numbers from 1 to 10 can be written using a **for** statement as follows:

```
for Count in 1 .. 10 loop           --count declared here
  Put( Integer'Image( Count ) );
end loop;
New_Line;
```

When run, this would produce:

```
1 2 3 4 5 6 7 8 9 10
```

*Note:* The variable *count* is declared by the **for** statement and is visible only for the extent of the **for** loop. It is a read only item and therefore cannot be written to.

The values may be stepped through in reverse order by inserting the keyword **reverse** after the keyword **in**. For example:

```
for Count in reverse 1 .. 10 loop
  Put( Integer'Image( Count ) );
end loop;
New_Line;
```

When run, this would produce:

```
10 9 8 7 6 5 4 3 2 1
```

## 30 Ada introduction: Part 1

*Note:* The range must evaluate to a possible list of values for the body of the **for loop** to be executed. For example:

```
for Count in reverse 10 .. 1 loop
  Put( Integer'Image( Count ) );
end loop;
```

would not execute the body of the **for loop**.

In the program below the two loops produce identical results:

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Main is
  Count      : Integer;           --count as Integer object
  Count_To : constant Integer := 10; --integer constant
begin
  Count := 1;
  while Count <= Count_To loop    --While loop
    Put( Integer'Image( Count ) );
    Count := Count + 1;
  end loop;
  New_Line;

  for Count in 1 .. Count_To loop  --count declared here
    Put( Integer'Image( Count ) );
  end loop;
  New_Line;
end Main;
```

When run, this would produce:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

*Note:* For the **for** loop a new count is declared which is visible only for the extent of the loop.

### 3.7.2 loop

Another way of writing the above loop is by using the infinite looping construct **loop end loop**. As this construct repeats for ever, an exit mechanism is provided to short-circuit the loop. This escape mechanism is the **exit** statement, which causes an immediate exit from the loop. Older programmers will recognise this as a restricted version of the goto statement. The loops seen earlier in Sections 2.5 and 2.7.1 could have been expressed using a **loop** construct as follows:

```

with Ada.Text_IO;
use   Ada.Text_IO;
procedure Main is
    Count      : Integer;           --count as Integer object
    Count_To : constant Integer := 10; --integer constant
begin
    Count := 1;
    loop
        Put( Integer'Image( Count ) );
        exit when Count = Count_To;    --Exit loop when ...
        Count := Count + 1;
    end loop;
    New_Line;
end Main;

```

When run, this would produce:

```
1 2 3 4 5 6 7 8 9 10
```

The exit from the **loop** is accomplished by adding a condition to the **exit** statement, in this case **when** Count = Count\_To.

*Note: The **loop** construct can be used when it is necessary to execute the code at least one-time. An **exit** statement may be used to exit from a **while loop** and a **for loop**.*

## 3.8 Other selection constructs

### 3.8.1 case

The previous series of **if then else** statements in Section 2.6 can be replaced by the following **case** statement:

```

case Count is
    when 3      => Put("Count is 3"); New_Line;
    when 4      => Put("Count is 4"); New_Line;
    when others => Put("Count is not 3 or 4"); New_Line;
end case;

```

In Ada, a **case** statement must take account of all values that the control variable may take: hence the **when others** component in the above statement. Had it not been present, then a compile-time error would have been generated.

## 32 Ada introduction: Part 1

A character variable may be declared which can hold a character from the Ada character set. The following **case** statement would print out a classification of the character held in the object Ch.

```
Ch := 'a';
case Ch is
  when '0' | '1' | '2' | '3' | '4' |
    '5' | '6' | '7' | '8' | '9' =>
    Put("Character is a digit");
  when 'A' .. 'Z' =>
    Put("Character is upper case English letter");
  when 'a' .. 'z' =>
    Put("Character is lower case English letter");
  when others =>
    Put("Not an English letter or digit");
end case;
New_Line;
```

In this **case** statement two ways of combining case labels are introduced.

Case component	Description	Explanation
	Or	For example, '0'   '1' will match the character '0' or the character '1'
..	Range	For example, 'A' .. 'Z' will match any character in the range Capital A to Capital Z.

The fragment of program code combined with appropriate declarations and compiled would produce when run:

```
Character is lower case English letter
```

## 3.9 Input and output

In Ada, input and output are performed by a variety of standard packages. The full implications of the package construct are discussed fully in Chapters 5 and 17 which describe in detail the I/O packages. For the moment, the discussion about input and output will concentrate solely on character data.

Text is output to the terminal using the `put` procedure. This procedure may take a formal parameter which is either a character or a character string. For example, to output `hello` the user could write either:

```
Put( "Hello" );
```

or

```
Put('h'); Put('e'); Put('l'); Put('l'); Put('o');
```

*Note: A string is use to represent a sequence of characters. A string is enclosed in " " whilst a character is enclosed in ' '. In this way the compiler can distinguish between a character 'A' and a string of a single character "A".*

To input a character into the variable `ch` of type `Character`, the user could write:

```
get(ch);
```

A simple program to copy its input, character-by-character to the output source, could be as follows:

```
with Ada.Text_IO;
use   Ada.Text_IO;
procedure Simple_Cat is
  Ch : Character;           --Current character
begin
  while not End_Of_File loop --For each Line
    while not End_Of_Line loop --For each character
      Get(Ch); Put(Ch);       --Read / Write character
    end loop;
    Skip_Line; New_Line;     --Next line / new line
  end loop;
end Simple_Cat;
```

The above program uses the following input and output functions or procedures.

Function/Procedure	Effect
End_Of_File	Delivers true when the end of the file is reached, otherwise it delivers false.
End_Of_Line	Delivers true when all the characters have been read from the current input line, otherwise it delivers false. NB. This does not include the new line character.
Skip_Line	Positions the input pointer at the start of the next line. Any information on the current line is skipped.
New_Line	Write the new line character to the output stream NB. On some systems new line is represented by two characters when output.

If compiled to the executable file `Simple_Cat`, the same program could be run on a Unix or MSDOS system to implement a simple software tool to print the contents of the file `about_ada`. To list the contents of the file `about_ada` to the terminal using an MSDOS system, a user could type:

```
Simple_Cat < about_ada
```

*Note: On a DOS or Unix system the command `Simple_Cat < about_ada` runs the program `Simple_Cat` taking its input from the file `about_ada`.*

### 3.10 Access to command line arguments

When a program is executed it is possible to access any arguments given on the same line as the program name. For example, the following program `echo` has two command line arguments:

```
echo Hello there!
```

## 34 Ada introduction: Part 1

If the program `echo` is compiled with the package `Ada.Command_Line` then the programmer has available the following function calls:

call of function	Returns
<code>Argument_Count</code>	The number of command line arguments. In this case, two.
<code>Argument(1)</code>	A string representing the first command line argument. In this case "Hello".
<code>Argument(2)</code>	A string representing the second command line argument. In this case "there!".

*Note:* It would be an error detected at run-time to access `argument(3)`.

The code for the program `echo` is as follows:

```
with Ada.Text_IO, Ada.Command_Line;
use  Ada.Text_IO, Ada.Command_Line;
procedure Echo is
begin
  for I in 1 .. Argument_Count loop --For each argument
    Put( Argument(I) );             -- Print it
    if I /= Argument_Count then    -- If not last
      Put( " " );                  -- Print separator
    end if;
  end loop;
  New_Line;
end Echo;
```

*Note:* See how the package `Ada.Command_Line` has been used here.

### 3.10.1 Putting it all together

If compiled to the executable file `echo`, the program could be run on a Unix or MSDOS system to implement the command `echo` as follows:

```
% echo Hello there!
```

Which when run would write:

```
Hello there!
```

## 3.11 A better cat program

By using, the package `Ada.Command_line` a better version of the `cat` program can be written. In this new version the files to be listed to the terminal are specified after the executable program name.

The following procedures and functions are used to control the reading of data from a file.

Function/Procedure	Effect
<code>Open</code>	Opens an existing file. A file descriptor to this file is returned as the result. A file descriptor is of type: <code>File_type</code> in the package <code>Ada.Text_IO</code> .
<code>Close</code>	Close the open file.



Function/Procedure	Effect
End_Of_File	As previously described but this time the effect is not on the normal input stream, but on the input of data from a file. The extra first parameter denotes the file descriptor attached to the file.
End_Of_Line	
Skip_Line	
Get	

This new program is as follows:

```

with Ada.Text_Io, Ada.Command_Line;
use  Ada.Text_Io, Ada.Command_Line;
procedure Cat is
  Fd  : Ada.Text_Io.File_Type;           --File descriptor
  Ch  : Character;                      --Current character
begin
  if Argument_Count >= 1 then
    for I in 1 .. Argument_Count loop  --Repeat for each file
      Open( File=>Fd, Mode=>In_File,    --Open file
            Name=>Argument(I) );
      while not End_Of_File(Fd) loop    --For each Line
        while not End_Of_Line(Fd) loop  --For each character
          Get(Fd,Ch); Put(Ch);          --Read / Write character
        end loop;
        Skip_Line(Fd); New_Line;        --Next line / new line
      end loop;
      Close(Fd);                        --Close file
    end loop;
  else
    Put("Usage: cat file1 ... "); New_Line;
  end if;
end Cat;

```

### 3.11.1 Putting it all together

Which when compiled to the executable file `cat` can be run as follows on an MSDOS or Unix system:

```
% cat file1.txt file2.txt
```

*Note:* If a file does not exist then the program will fail with an uncaught exception condition. Chapter 12 describes how such exceptional conditions may be caught and processed in a program.

## 3.12 Characters in Ada

In Ada there are two distinct types used for holding characters. These are:

Type	An instance of this type
Character	Can hold 256 distinct characters. Characters with internal code 0-127 are from the ASCII character set. The ASCII standard is equivalent to ISO 8859.
Wide_Character	Can hold 65536 different characters. The characters are defined in ISO 10646 BMP

*Note:* Many computer systems use the ASCII character set to represent data held internally or transmitted. In Ada 83 variables of type `Character` are restricted to holding only 128 different character values compared to Ada 95's 256.

## 36 Ada introduction: Part 1

These types are defined as enumeration types in the package standard. A consequence of this is that characters like 'A' are enumerations of both `Character` and `Wide_character`. The effect is that a programmer cannot write:

```
if 'A' = 'A' then    ...    end if;
```

as the character 'A' could belong to the type `Character` or `Wide_Character` which the compiler cannot resolve from the statement.

### 3.13 Self-assessment

- What is the purpose of the package `Ada.Text_IO`?
- What are the disadvantages of the **exit** statement?
- Why did the designers of Ada make the control variable in a **for** loop read only?
- Why might the omission of **when others** in a **case** statement cause a compile-time error?
- Can every **loop end loop** statement be expressed as a **while end loop** statement which does not have an **exit** statement? For example, the following program illustrates a **loop end loop**:

```
with Ada.Text_IO;
use  Ada.Text_IO;
procedure Main is
  Count      : Integer;           -- Variable
  Count_To   : constant Integer := 10; -- Integer constant
begin
  Count := 1;
  loop
    Put( Integer'Image( Count ) );
    exit when Count = Count_To;    -- Exit loop when ...
    Count := Count + 1;
  end loop;
  New_Line;
end Main;
```

Is the converse true? Explain your answer.

- What are the major differences discussed so far between Ada and other programming languages known to you?
- Why might an Ada 95 program using a `Character` variable not compile using an Ada 83 compiler?
- How may command line arguments be accessed from a program?

### 3.14 Exercises

Construct the following programs:

- Numbers  
A program to print the first 20 positive numbers (1, 2, 3, etc.).
- Times table  
A program to print out the 8 times table so that the output is in the following form:
 

```

8 * 1   =      8
8 * 2   =     16
8 * 3   =     24
.   .   .
8 * 12  =     96
      
```
- Series  
A program to print out numbers in the series 1 1 2 3 5 8 13 ... until the last term is greater than 10000.
- Character table  
A program to print the characters represented by the numbers 32 to 126.

Hint:

If a variable `number` is of type `Integer` then `Character 'Val (number )` will deliver the character which is represented internally by the value contained in `number`.

- Table  
A program to print out the square, cube and 4th power of the first 15 positive numbers.

## 4 Ada introduction: Part 2

This chapter looks at declarations and use of scalar data items in Ada. One of Ada's key contributions to programming is the ability to declare data items that can only take a specific range of values. Ada's strong typing ensures that many errors in a program will be detected at compile rather than run-time.

### 4.1 Introduction

So far, only objects of type `Integer` or `Character` have been introduced. An `Integer` object stores a number as a precise amount with no decimal places. The exact range of values that can be stored is implementation defined. A user can find out this range by employing the attributes `'First` and `'Last` on the type `Integer`. In addition the attribute `'Bits` returns the size in bits of an `Integer` object. The following program prints these attributes for an `Integer` type.

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
begin
  Put("Smallest Integer "); Put( Integer'Image(Integer'First)); New_Line;
  Put("Largest Integer "); Put( Integer'Image(Integer'Last)); New_Line;
  Put("Integer (bits) "); Put( Integer'Image(Integer'Size )); New_Line;
end Main;
```

*Note: The attribute 'First is pronounced 'tick first'.*

When compiled and run on two different machines, this would produce:

Machine using a 16 bit word size	Machine using a 32 bit word size
Smallest integer -32768 Largest integer 32767 Integer (bits) 16	Smallest integer -2147483648 Largest integer 2147483647 Integer (bits) 32

### 4.2 The type Float

An instance of the type `Integer` holds numbers to an exact value. In the solution of some problems the numbers manipulated will not be an exact value. For example, a person's weight is 80.23 kilograms. The data type `Float` elaborates an object which can hold a number which has decimal places. Thus in a program a person's weight can be held in the object `weight` which is declared as follows:

```
Weight : Float := 80.23;
```

A `Float` is implemented as a floating point number. A floating point number holds a value to a specific number of decimal digits. This will in many cases be an approximation to the exact value which the programmer wishes to store. For example, a  $1/3$  will be held as 0.333 ... 33. The following table shows how various numbers are effectively stored in floating point form to 6 decimal places:

Number	Scientific notation	Floating point form
80.23	$0.8023 * 10^2$	+802300 +02
0.008023	$0.8023 * 10^{-2}$	+802300 -02
0.333333	$0.333333 * 10^0$	+333333 +00

*Note: In reality the floating point number will be held in binary.*

The main consequence of using a floating point number is that numbers are held to an approximation of their true value. Calculations using floating point numbers will usually only give an approximation to the true answer. However, in many cases this approximation will not cause any problems. An area where this approximation will cause problems is when the value represents a monetary amount.

The attributes 'First, 'Last and 'Size may also be applied to objects of type Float. In addition the attribute 'Digits returns the precision in decimal digits of a number stored in a Float object. For example, the following program:

```
with Ada.Text_IO;
use  Ada.Text_IO;
procedure Main is
begin
  Put("Smallest Float ");
  Put( Float'Image( Float'First ) ); New_Line;
  Put("Largest Float ");
  Put( Float'Image( Float'Last ) ); New_Line;
  Put("Float (bits)    ");
  Put( Integer'Image( Float'Size ) ); New_Line;
  Put("Float (digits)  ");
  Put( Integer'Image( Float'digits ) ); New_Line;
end Main;
```

*Note: Float'Image delivers a string representing a floating point number in scientific notation.*  
when compiled and run on two different machines would produce:

Machine using a 32 bit word size	Machine using a 64 bit word size
Smallest Float -3.40282E+038 Largest Float 3.40282E+038 Float (bits) 32 Float (digits) 6	Smallest Float -1.79769313486232E+308 Largest Float 1.79769313486232E+308 Float (bits) 64 Float (digits) 15

### 4.2.1 Other Integer and Float data types

Some implementations of Ada may provide data types that offer a greater precision than the in-built types of Integer and Float. If these are provided they will be called Long\_Integer, Long\_Float, Long\_Long\_Integer etc.

## 4.3 New data types

Using an object of type Integer to hold numeric values may be a useful approach, but it does not lead to a program that is machine independent. For example, during its execution a program could create values which were not containable in a particular machine's Integer object. If this happened, then the program would fail with a run-time error of 'Constraint\_Error'.

Ada provides an elegant solution to this problem. It allows a user to define a new data type, which has a specific range of values. For example, the following declaration defines a new data type Distance which will hold the distance between two places:

```
type Distance is range 0 .. 250_000;
```

*Note: If the compiler cannot provide an object which can hold such a range, a compile-time error message will be generated.*

## 40 Ada introduction: Part 2

Distance is a new type, instances of which may not be mixed with instances of other types. The following table shows some examples of type declarations in Ada.

Type declaration	An instance of T will Declare
<b>type</b> T <b>is range</b> 0 .. 250_000;	An object which can hold whole numbers in the range 0 .. 250_000.
<b>type</b> T <b>is digits</b> 8;	An object which can hold a floating point number which has a precision of 8 digits.
<b>type</b> T <b>is digits</b> 8 <b>range</b> 0.0 .. 10.0;	An object which can hold a floating point number which has a precision of 8 digits and can store numbers in the range 0.0 .. 10.0.

### 4.3.1 Type conversions

To convert between compatible scalar types the type name of the required type is used to convert an object to the required type. For example, the following program converts an object of type `Apples` into an object of type `French_Apples`.

```
procedure Main is
  type Apples      is range 0 .. 100;
  type French_Apples is range 0 .. 100;
  Number           : Apples;
  Number_From_France : French_Apples;
begin
  Number := 10;
  Number_From_France := French_Apples( Number );
end Main;
```

It is, however, up to the programmer to determine whether the conversion is meaningful. Conversion, however, can only take place between types that are compatible.

### 4.3.2 Universal integer

To avoid tedious type conversion when dealing with constants, Ada has the concept of a universal integer. The compiler will automatically convert a universal integer to an appropriate type when used in an arithmetic expression. In Ada all integer numeric constants are regarded as being of type universal integer. Likewise all floating point constants are regarded as a universal float.

### 4.3.3 Constant declarations

To make a program more readable, all values other than 0 or 1 should normally be given a symbolic name. This helps to improve the readability of a program and allows the programmer to change the value by means of a single textual change. For example, the capacity of a car park could be described as:

```
Max_Parking_Spaces: constant := 100;
```

This describes `Max_Parking_Spaces` as a universal integer. However, if the declaration had been:

```
Max_Parking_Spaces: constant Parking_Spaces := 100;
```

then `Max_Parking_Spaces` would be a constant of type `Parking_spaces`.

*Note: The latter declaration will restrict the places where `Max_Parking_Spaces` can be used to only those places where a value of type `Parking_Spaces` can occur.*

## 4.4 Modified countdown program

The countdown program shown earlier in Section 3.4 can be rewritten, restricting count to the values 1 to 10 as follows:

```
with Ada.Text_IO;
use   Ada.Text_IO;
procedure Main is
  type Count_Range is range 0 .. 10;
  Count : Count_Range := 10;           --Declaration of count
begin
  for Count in reverse Count_Range loop
    if Count = 3 then                  --If 3 print Ignition
      Put("Ignition"); New_Line;
    end if;
    Put( Count_Range'Image( Count ) ); --Print current count
    New_Line;
    Delay 1.0;                         -- Wait 1 second
  end loop;
  Put("Blast off"); New_Line;         --Print Blast off
end Main;
```

*Note:* Even though count is of type Count\_range, it can be compared with the integer constant 3. The use of the type Count\_Range in the **loop** statement. This confines the loop to the range of values that an instance of Count\_Range can take. The use of

`Count_Range'Image( Count )`

to deliver a character representation of the contents of count. Remember count is of type Count\_Range.

## 4.5 Input and output in Ada

One of the obstacles in writing programs in Ada is the complexity involved in outputting integer and floating point numbers. To simplify this process Ada 95 provides the following packages:

- `Ada.Integer_Text_IO`      for input and output of integer numbers.
- `Ada.Float_Text_IO`        for input and output of floating point numbers.
- `Ada.Text_IO`                for input and output of characters and strings.

Chapter 18 describes how specific packages in `Ada.Text_IO` are instantiated to output instances of other integer and floating point types.

## 4.6 The package `Ada.Float_Text_IO`

The package `Ada.Float_Text_IO` is used to input and output floating point numbers.

## 42 Ada introduction: Part 2

### 4.6.1 Output of floating point numbers

A floating point number is output using the overloaded procedure `Put`, though by default this displays the number in scientific notation. Extra parameters to `Put` are used to control the output form of the floating point number. These parameters are used together or individually. The main parameters to control the format are named `Fore`, `Aft` and `Exp`. For example, to output the contents of the `Float` object `Num` that contains 123.456 the following versions of `Put` may be used:

Put statement	Output	Notes
<code>Put( Num );</code>	1.23456E+02	1
<code>Put( Num, Fore=&gt;4, Aft=&gt;2, Exp=&gt;0 );</code>	1.23	2
<code>Put( Num, Fore=&gt;4, Aft=&gt;2, Exp=&gt;3 );</code>	1.23E+02	3

*Note:* Section C.5, Appendix C contains a description of the package `Ada.Text_IO` and shows other forms of the `put` statement. Section 5.9 describes in more detail how parameters to a procedure or function may be named.

- 1 Scientific notation by default.
- 2 `Aft =>2` Number of places after the decimal point.  
`Fore =>4` Number of places before the decimal point.  
This includes any sign character such as `-`.  
`Exp =>0` No exponent hence non scientific notation.
- 3 `Exp =>3` Scientific notation with three places for the exponent.  
This includes any sign character such as `-`.

### 4.6.2 Input of floating point numbers

A floating point number is input using the overloaded procedure `Get` as follows:

Get statement	Notes
<code>Get( Num );</code>	1
<code>Get( Num, Width=&gt;5</code>	2

*Notes:*

- 1 Reads a floating point number from the input source. It is an error to read an integer number. This procedure will skip any leading white space characters before reading the floating point number.
- 2 `Width=> 5`  
Number of characters input when constructing the floating point number. If a line terminator is encountered no more characters are input to form the number

## 4.7 The package `Ada.Integer_Text_IO`

The package `Ada.Integer_Text_IO` is used to input and output floating point numbers.

### 4.7.1 Output of integer numbers

In the output of an integer number the parameters `base` and `width` can be used together or individually to control the format of the output. For example, to output the contents of the `Integer` object `Num` which contains 42 the following versions of `put` may be used:

The put statement	Output	Notes
<code>Put( Num );</code>	42	1
<code>Put( Num, Base=&gt; 8, Width=&gt;5 );</code>	8#52#	2

*Notes:*

- 1 Output in the default field width.
- 2 `Base =>8` Output base: in this case octal.  
`Width =>5` Field width for the number.



### 4.7.2 Input of integer numbers

An integer is input using the overloaded procedure `Get` as follows:

Theget statement	Notes
<code>Get( Num );</code>	1
<code>Get( Num, Width=&gt;5</code>	2

Notes:

- 1 Reads an integer number from the input source. This procedure will skip any leading white space characters before reading the integer number.
- 2 `Width=> 5`  
Number of characters input when constructing the integer number. If a line terminator is encountered no more characters are input to form the number.

## 4.8 Conversion between Float and Integer types

Because `Float` and `Integer` are two separate types, instances of these types may not be mixed. This initially can cause problems as often we informally mix whole numbers and 'floating point numbers' together. For example, the following program prints a conversion table for whole pounds to kilograms:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
procedure Main is
begin
  for I in 1 .. 5 loop
    Put( I ); Put( " Pounds = " );
    Put( Float(I) / 2.2046, Exp=>0, Aft=>2 );
    Put( " Kilograms" ); New_Line;
  end loop;
end Main;
```

Note: The explicit conversion `Float( I )` converts the `Integer` object `I` to an instance of a `Float`.

When run, this will give the following output:

```
1 Pounds = 0.45 Kilograms
2 Pounds = 0.91 Kilograms
3 Pounds = 1.36 Kilograms
4 Pounds = 1.81 Kilograms
5 Pounds = 2.27 Kilograms
```

The conversion process may be used in reverse to convert a floating point number to an integer form. The effect of this conversion is to round away from zero, so that:

Float object <code>f</code> contains	Integer( <code>f</code> ) delivers
1.5	2
1.3	1
-1.5	-2
-1.3	-1

## 44 Ada introduction: Part 2

### 4.9 Type safety in a program

By using the type mechanism, errors in a program can be detected at compile-time. For example, a program which processes distances in miles and kilometres can be made safer by defining separate types for miles and kilometres as follows:

```
type Miles      is digits 8 range 0.0 .. 25_000.0;
type Kilometres is digits 8 range 0.0 .. 50_000.0;
```

*Note:* The range of values is adequate to accommodate any distance between two points on the earth.

A program which processes distances between cities could be defined as follows:

```
with Ada.Text_IO, Ada.Float_Text_IO;
use  Ada.Text_IO, Ada.Float_Text_IO;
procedure Main is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;
  London_Paris   : Miles;
  Paris_Geneva    : Kilometres;
  London_Paris_Geneva : Kilometres;
begin
  London_Paris := 210.0;  --Miles
  Paris_Geneva := 420.0;  --Kilometres
  London_Paris_Geneva :=
    Kilometres( London_Paris * 1.609_344 ) + Paris_Geneva;
  Put("Distance london - paris - geneva (Kms) is " );
  Put( Float( London_Paris_Geneva ), Aft=>2, Exp=>0 );
  New_Line;
end Main;
```

*Note:* There is an explicit conversion of a distance in miles to kilometres using the type conversion `kilometres( london_paris * 1.609_344 )`.  
The contents of `London_Paris_Geneva` has been converted to a `Float` so that it can be printed using the package `Ada.Float_Text_IO`. Chapter 17 describes how user defined types may be output.  
The parameters to `put` when outputting a floating point number control the number of decimal places output and the format of the number. Section C.5, Appendix C lists the parameters used in outputting numbers.

If by accident a programmer wrote:

```
London_Paris_Geneva := London_Paris + Paris_Geneva;
```

then the Ada compiler would detect a type mismatch at compile-time. London to Paris is in miles and Paris to Geneva is in kilometres.

### 4.10 Subtypes

The type mechanism can on occasion, be restricting as a programmer wants the range checking provided by the type mechanism but does not want to have to keep explicitly performing type conversions. A subtype of a type provides the range checking associated with a type, but instances of a type and its subtypes may be freely mixed in expressions.

For example, the speed of various forms of transport can be defined using the type and subtype mechanism as follows:

```
type    Speed_Mph      is range 0 .. 25_000;
subtype Train_Speed    is Speed_Mph range 0 .. 130;
subtype Bus_Speed      is Speed_Mph range 0 .. 75;
subtype Cycling_Speed  is Speed_Mph range 0 .. 30;
subtype Person_Speed   is Speed_Mph range 0 .. 15;
```

A subtype is derived from an existing type and constrains the values that can be assigned to an instance of the subtype. The compiler will enforce this constraint either by performing a compile-time check or by generating code to check the constraint at run-time. Of course, the subtype inherits all the operations that can be performed on an instance of the type.

Instances of a type and its subtypes may be freely mixed in arithmetic, comparison and assignment operations. For example, using the above type and subtypes declarations for the speed of various forms of transport, the following code can be written:

```
with Ada.Text_IO;
use   Ada.Text_IO;
procedure main is
  --      Type and subtype declarations for speeds
  T0715 : Train_Speed;  --07:15 Brighton - London
  B0720 : Bus_Speed;    --07:20 Brighton - London
begin
  T0715 := 55;  --Average speed Brighton - London (Train)
  B0720 := 35;  --Average speed Brighton - London (Bus)
  if T0715 > B0720 then
    Put("The train is faster than the bus");
  else
    Put("The bus is faster than the train");
  end if;
  New_Line;
end Main;
```

*Note: It is of course an error to mix instances of subtypes which are derived from different types.*

#### 4.10.1 Types vs. subtypes

Criteria	Types	Subtype
Instances may be mixed with	only instances of the same type	only instances of a type and subtypes derived from the type
May have a constraint	Yes	Yes

#### 4.11 More on types and subtypes

In Ada only subtypes have names. The consequence of this is that the declaration:

```
type    Speed_Mph      is range 0 .. 25_000;
```

## 46 Ada introduction: Part 2

is effectively treated as:

```
type    Anonymous    is -- implementation defined
subtype Speed_Mph    is Anonymous range 0 .. 25_000;
```

The anonymous type from which `Speed_Mph` is derived can be obtained by using the attribute `'Base`. The attribute `'Base` refers to the anonymous base type from which a type or subtype has been originally derived. For example, the range of the anonymous type from which `Speed_Mph` is derived is printed with the following code:

```
Put("The base range of the type T2 is " );
Put( Integer(T2'Base'First) ); Put( " .." );
Put( Integer(T2'Base'Last) ); New_Line;
```

*Note:* An instance of the base type can be declared by using the type declaration `Speed_mph'Base`.

### 4.11.1 Root\_Integer and Root\_Real

The model of Ada's arithmetic is based on the anonymous types `Root_Integer` and `Root_Real`. These types are in effect used as the base types from which all integer and real types are derived. The following table summarizes the properties of `Root_Integer` and `Root_Real`.

Root type	Range / precision
<code>Root_Integer</code>	<code>System.Min_Int .. System.Max_Int</code>
<code>Root_Real</code>	<code>System.Max_Base_Digits</code>

All the arithmetic operators are defined to operate on, and deliver instances of, their base type.

### 4.11.2 Type declarations: root type of type

In declaring a type for an integer, there are two distinct approaches that can be taken. These are illustrated by the two type declarations for an `Exam_mark`.

```
type Exam_Mark is new Integer range 0 .. 100;
type Exam_Mark is range 0 .. 100;
```

The first declaration defines `Exam_Mark` to be a type derived from `Integer` with a permissible range of values `0 .. 100`. Its base type will consequently be that of `root_integer` as `Exam_Mark` is derived from `Integer`.

The second declaration defines `Exam_Mark` to be a type, the values of which are in the range `0 .. 100`. It is derived from `Root_Integer` but the base range of the type does not have to be that of `Root_Integer`. Some implementations may implement an instance of this type and its base type in a single byte.

The following table illustrates the base type of the types described above:

type Exam_Mark is	Base type	Minimum range of root type
<code>new Integer range 0 .. 100;</code>	<code>Root_Integer</code>	<code>System.Min_Int .. System.Max_Int</code>
<code>range 0 .. 100;</code>	Implementation defined	Implementation defined but must hold <code>0 .. 100</code>

When performing arithmetic with an instance of a type's base type, no range checks take place. This allows an implementor to implement the base type in the most efficient or effective way for a specific machine. However, the exception `Constraint_Error` will be generated if the resultant arithmetic evaluation leads to a wrong result. For example, the exception `Constraint_Error` is generated if an overflow is detected when performing calculations with the base type.

### 4.11.3 Arithmetic with types and subtypes

In a program dealing with a student's exam marks, the following program is written to average the marks for a student taking English, Maths and Computing:

```
with Ada.Text_IO;
use  Ada.Text_IO;
procedure Main is
  type Exam_Mark is new Integer range 0 .. 100;
  English  : Exam_Mark;      --English exam mark
  Maths    : Exam_Mark;      --Maths      "    "
  Computing : Exam_Mark;      --Computing  "    "
  Average  : Exam_Mark;      --
begin
  English  := 72;
  Maths    := 68;
  Computing := 76;
  Put("Average exam mark is ");
  Average  := (English + Maths + Computing) / 3;
  Put( Exam_Mark'Image(Average) ); New_Line;
end Main;
```

In executing the statement:

```
Average := (English+Maths+Computing) / 3;
```

the expression:

```
(English+Maths+Computing) / 3
```

will generate a result which is in the range 0 .. 100. However, the component of the statement `English+Maths+Computing` will generate a temporary result which is outside the range of `Exam_Mark`.

In Ada the arithmetic operations are defined to process instances of the root types. In evaluating `English+Maths+Computing`, `English+Maths` will deliver a temporary object of type `Root_Integer` (`Exam_Mark'Base`) which is then added to `Computing`. The result of the addition is divided by 3 at which point a range check is performed on the temporary result before it is assigned to the object `average`.

Of course, for this to work the `Root_Integer` type must be sufficiently large to hold the sum of `English+Maths+Computing`. Remember, this will be of type `Root_Integer` which has a range of  $-2^{15} .. 2^{15}-1$ .

### 4.11.4 Warning

If the declaration for `Exam_mark` were replaced by:

```
type Exam_Mark is range 0 .. 100;
```

then the above program would fail with a `Constraint_Error` if the base type of `Exam_Mark` were to be implemented in a single byte.

## 48 Ada introduction: Part 2

### 4.11.5 Constrained and unconstrained types

In Ada there are no named types only subtypes. The `Integer` and `Float` types are derived from `Root_Integer` and `Root_Float` respectively. Range checks only apply to constrained subtypes, but overflow checks always apply. For example, using the declaration of `Exam_Mark`:

```
type Exam_Mark is new Integer range 0 .. 100;
```

the following properties hold.

Declaration	Instance is	Commentary
<code>Exam_Mark</code>	Constrained	Constrained to the range 0 .. 100.
<code>Exam_Mark'Base</code>	Unconstrained	No range checks applied to assignment of this variable. An implementor may allow this to have a range greater than the base range of the root type.

Declaration	Instance is	Commentary
<code>Integer</code>	Constrained	Constrained to the base range of <code>Integer</code> , which is implementation dependent.
<code>Integer'Base</code>	Unconstrained	No range checks apply; may have a range greater than <code>Integer</code> .

*Note: Regardless of whether an item is constrained or unconstrained, overflow checks will always apply. Thus, the result obtained will always be mathematically correct. Take note of the difference between `Integer` and `Integer'Base`. Instances of the type `Integer` are constrained to the base range of the type, whilst instances of `Integer'Base` are not.*

### 4.11.6 Implementation optimizations

An Ada compiler is allowed to represent an instance of a base type to a greater precision than is necessary. For example, with the following declarations:

```
type Exam_Mark is new Integer range 0 .. 100;
type Temporary is Exam_Mark'Base;
English : Exam_Mark;
Total   : Temporary
```

the variable `Total` may be implemented to hold numbers of a greater range than is allowed by an `Integer` declaration. This is to allow compiler writers the opportunity to perform optimizations such as holding a variable or intermediate result in a CPU register which may have a greater precision than the range of normal `Integer` values. Of course, overflow checks will be performed at all times, so the mathematical result is always correct.

The danger is that a program which compiles and runs successfully using a particular compiler on a machine may fail to run successfully when compiled with a different compiler on the same machine, even though both compilers have the same range for an `Integer`.

## 4.12 Compile-time and run-time checks

The following program declares data types to represent: (a) the number of power points in a room, (b) the capacity of a lecture room in seats, and (c) the capacity of a tutorial room, again in seats. In this program, various assignments are made, some of which will fail to compile, some of which will fail in execution.

```

procedure Dec is
  type    Power_Points is           range 0 .. 6;
  type    Room_Size    is           range 0 .. 120;
  subtype Lecture_Room is Room_Size range 0 .. 75;
  subtype Tutorial_Room is Room_Size range 0 .. 20;

  Points_In_504 : Power_Points;           --Power outlets
  People_In_504  : Lecture_Room;           --Size lecture room
  People_In_616  : Tutorial_Room;          --Size tutorial room
begin
  Points_In_504 := 3;                      --OK
  Points_In_504 := 80;                     --Error / Warning

  People_In_504 := 15;                     --OK
  People_In_616 := People_In_504;          --OK

  People_In_504 := Points_In_504;           -- Type Mismatch

  People_In_504 := Lecture_Room( Points_In_504 ); --Force

  People_In_504 := 50;                     --OK
  People_In_616 := People_In_504;          --Constraint error
end Dec;

```

The compilation or execution of the following lines will fail for the following reasons:

Line	Reason for failure
Points_In_504:= 80;	The range of values allowed for the object Points_In_504 does not include 80. This error will usually be detected at compile-time.
People_In_504 := Points_In_504;	The objects on the LHS and RHS of the assignment statement are of different types and will thus produce a compile-time error.
People_In_616 := People_In_504;	Will cause a constraint error when executed, as the object People_In_504 contains 50. In this example, the error could in theory be detected at compile-time.

*Note: Depending on the quality of the compiler, some errors which in theory could be detected at compile-time, will only be detected at run-time. Conversely possible run-time errors may be flagged through warnings at compile time.*

This shows the strength of Ada's strong type checking: problems in a program can be identified at an early stage of development. However, careful planning needs to be made when writing a program. Decisions about which distinct data types to use and which data types should be a subtype of others are particularly important.

### 4.12.1 Subtypes Natural and Positive

The Ada language pre-defines the following subtypes:

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

## 4.13 Enumerations

In writing a program dealing with different classifications of an item it is good programming practice to give meaningful names to each of the different classifications that an item may have. For example, in a program that deals with colours, an incorrect approach would be to let each colour take a numeric value, as follows:

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Main is
  Car_Colour : Integer;
begin
  Car_Colour := 1;
  case Car_Colour is
    when 1 => Put("A red car"); New_Line;
    when 2 => Put("A blue car"); New_Line;
    when 3 => Put("A green car"); New_Line;
    when others => Put("Should not occur"); New_Line;
  end case;
end Main;
```

*Note: Remember the **when others** is required as a case statement and must cover all possible values.*

This however, is not very elegant and can lead to confusion about which colour 1 represents. There is also the danger that a non valid colour will be assigned to the object Car\_Colour. By using an enumeration, specific names can be given to the colours that a car may have. The declaration for the enumeration Colour is as follows:

```
type Colour is (Red,Blue,Green);
```

The type Colour is then used to elaborate objects which can only take the values of Red, Blue or Green. The enumeration type Colour is used as follows in the re-writing of the previous code fragment:

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Main is
  type Colour is (Red,Blue,Green);
  Car_Colour : Colour;
begin
  Car_Colour := Blue;

  case Car_Colour is
    when Red => Put("A red car"); New_Line;
    when Blue => Put("A blue car"); New_Line;
    when Green => Put("A green car"); New_Line;
  end case;
end Main;
```

*Note: As the only possible values that can now be assigned to Car\_Colour are either Red, Blue or Green, the **case** statement can be simplified.*



### 4.13.1 Enumeration values

As well as symbolic names enumerations may also be character constants. For example, the type `Character` is an enumeration made up of the characters in the standard ISO character set.

Thus the type `Character` is conceptually defined as:

```
type Character is ( nul, soh,          -- etc
                  ' ', '!', '"',      -- etc
                  '@', 'A', 'B', 'C',  -- etc
                  );
```

*Note: The package `standard` contains conceptual definitions of all the pre-defined types. Section C.4, Appendix C contains a listing of the package `Standard`.*

A programmer can define his/her own enumerations containing characters. For example, the following is an enumeration type declaration for a binary digit.

```
type Binary_Digit is ( '0', '1' );
B_Digit : Binary_Digit := '0';
```

### 4.13.2 The attributes `'Val` and `'Pos`

The position of a specific enumeration in a type is delivered with the attribute `'Pos` whilst the representation of an enumeration n'th value is delivered by `'Val`. These attributes are usefully used on the pre-defined enumeration `Character` to deliver respectively the character code for a specific character and the character representing a value. For example, the following program prints the character code for 'A' and the character representing character code 99.

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use   Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
begin
  Put("Character 'A' has internal code ");
  Put( Character'Pos('A') ); New_Line;
  Put("Code 99 represents character    ");
  Put( Character'Val(99) ); New_Line;
end Main;
```

Which when compiled and run will print:

```
Character 'A' has internal code    65
Code 99 represents character      c
```

## 4.14 The scalar type hierarchy

The types that are used in arithmetic operations are derived from the scalar types, with the only exception of the enumerated types. Even though they are considered part of the hierarchy, they may not be used in arithmetic operations. The type hierarchy is illustrated in Figure 4.1.

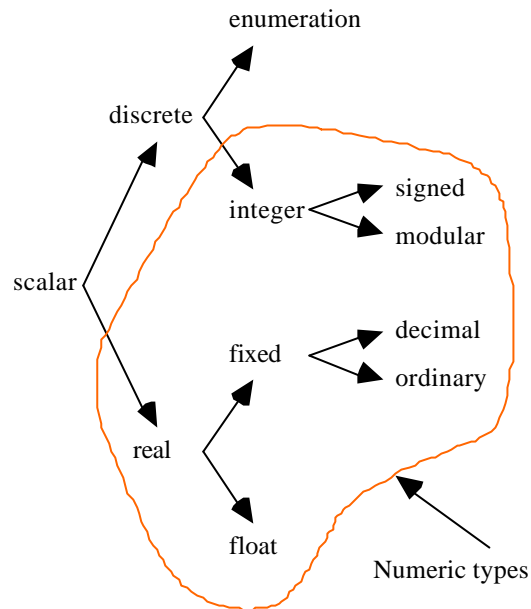


Figure 4.1 Type hierarchy for the scalar types.

Component	Example declaration	Note
Scalar		
discrete		
Enumeration	<code>type colour is ( Red, Green, Blue );</code>	1
Integer	<code>type Miles is range 0 .. 10_000;</code>	
Signed		
Modular	<code>type Byte is mod 256;</code>	2
Real		
Fixed		
Ordinary	<code>type Miles is delta 0.1 range 0.0 .. 10.0;</code>	3
Decimal	<code>type Miles is delta 0.1 digits 8;</code>	
Float	<code>type Miles is digits 8 range 0.0 .. 10.0;</code>	4

*Note 1* The enumeration types include the inbuilt types *Character*, *Wide\_Character* and *Boolean*.

*Note 2* A modular type implements modular arithmetic. Thus, the following fragment of code:

```

type Byte is mod 256;
count : Byte := 255;
begin
    count := count + 1;
  
```

would result in *count* containing 0.

*Note 3 A fixed point number is effectively composed of two components: the whole part and the fractional part stored in an integer value. This can lead to more efficient arithmetic on a machine which does not have floating point hardware or where the implementation of floating point arithmetic is slow. It also provides a precise way of dealing with numbers that have a decimal point.*

*An alternative notation for a decimal fixed point type is:*

**type** Miles **is delta** 0.1 **digits** 8 **range** 0.0 .. 10.0;

*However, even though all compilers must parse this type declaration they only need to support it if the compiler implements the Information systems Annex.*

*Note 4 A floating point number.*

*An alternative type declaration is:*

**type** Miles **is digits** 8;

*which defines the precision 8 digits but not the range of values that may be stored.*

#### 4.14.1 The inbuilt types

Ada provides the following inbuilt types.

Type	Classification	An instance of the type
Boolean	Enumeration	Holds either True or False.
Character	Enumeration	Holds a character based on the ISO 8859-1 character set. In which there are 256 distinct characters.
Float	Float	Holds numbers which contain a decimal place.
Integer	Integer	Holds whole numbers.
Wide_character	Enumeration	Holds a character based on the ISO 10646 BMP character set. In which there are 65536 distinct characters.

The implementation minimum values for these types are given in Section B.6, Appendix B.

#### 4.15 Arithmetic operators

The arithmetic operators in Ada 95 are:

+	Addition
-	Subtraction
*	Multiplication
/	Division

*The following arithmetic operators are defined on integer values only:*

<b>mod</b>	Modulus
<b>rem</b>	Remainder
<b>abs</b>	Returns the absolute value

The operators **mod** and **rem** are similar, and will give identical results when both operands have the same sign. The operator **rem** gives a remainder corresponding to the integer division operation /. The consequence of this is that as integer division truncates towards 0, the absolute value of the result will always be the same regardless of

## 54 Ada introduction: Part 2

the sign of the operands. **mod** meanwhile gives the remainder corresponding to a division with truncation towards minus infinity.

The following tables illustrate the result of using **mod** and **rem**. With both operators an RHS (Right Hand Side) of 0 will cause the exception **Constraint\_error** to be raised. The resultant exception **Constraint\_error** is indicated by the message **Err** in the tables.

mod	-5	-3	0	3	5
-5	0	-2	Err	1	0
-3	-3	0	Err	0	2
0	0	0	Err	0	0
3	-2	0	Err	0	3
5	0	-1	Err	2	0

rem	-5	-3	0	3	5
-5	0	-2	Err	-2	0
-3	-3	0	Err	0	-3
0	0	0	Err	0	0
3	3	0	Err	0	3
5	0	2	Err	2	0

The operator **abs** delivers the absolute value of an interger quantity.

### 4.15.1 Exponentiation

The operator **\*\*** is used to raise a real or integer value to a whole power, which must be greater or equal to zero.

**	Exponentiation
----	----------------

The effect of using **\*\*** for different powers of integer values is shown in the table below. The exception **Constraint\_error** is raised for a negative RHS.

**	-3	-1	0	1	3
-3	Err	Err	1	-3	-27
-1	Err	Err	1	-1	-1
0	Err	Err	1	0	0
1	Err	Err	1	1	1
3	Err	Err	1	3	27

The implementation of  $a ** b$  can be performed by multiplication in any order.  
Hence  $a ** 4$  could be implemented as  $a * a * a * a$  or  $(a * a) ** 2$ .

### 4.15.2 Monadic arithmetic operators

The monadic integer arithmetic operators are as follows:

-	Negation
+	Positive form

These deliver the negative and positive of an integer or floating point expression/number.

## 4.16 Membership operators

The membership operators are:

in	is a member of
not in	is not a member of

These operators check if a value is a member of a subtype or range. For example, to check if a letter belongs to the upper case alphabetic characters the following code may be used:

```

if Ch in 'A' .. 'Z' then
  Put("Character is Upper Alphabetic"); New_Line;
end if;

```

Alternatively, to check if an item is not a member of the upper case alphabetic characters, the code would be:

```

if Ch not in 'A' .. 'Z' then
  Put("Character is not Upper Alphabetic"); New_Line;
end if;

```

The membership test can also be used to check if a value is in the range of a subtype. For example:

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  subtype Exam_Mark is Integer range 0 .. 100;
  Mark : Integer;
begin
  Get( Mark );
  if Mark in Exam_Mark then
    Put("Valid mark for exam"); New_Line;
  end if;
end Main;

```

*Note: However, if Exam\_mark had been declared as a type then a compile-time error would be generated, as the type of operands of **in** are not compatible.*

## 4.17 Use of types and subtypes with membership operator

A program to convert a person's height in inches to metres is shown below:

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Metres_In_Inch : constant Float := 0.0254;    --Conversion
  Max_Height     : constant Float := 120.0;    --
  subtype Metres is Float range 0.0 .. Max_Height*Metres_In_Inch;
  subtype Inches is Float range 0.0 .. Max_Height;
  Height_Inches : Float;                        --Data
  Height_Metres : Metres;                       --Converted
begin
  Put("Enter person's height in Inches ");
  Get( Height_Inches );                        --Get data
  if Height_Inches in Inches then              --Sensible
    Height_Metres := Height_Inches * Metres_In_Inch; --Convert
    Put("Height in Metres is ");
    Put( Height_Metres, Exp=>0, Aft=>2 ); New_Line;
  else
    Put("Height not valid"); New_Line;          --Error
  end if;
end main;

```

In the program, subtypes have been used to help check the consistency of the input data and so that internal consistency checks can be performed on calculations. The person's height is read into the variable Height\_Inches which is of type Float. Validation against the range of the subtype Inches is then performed. The height in inches is then converted to metres and assigned to Height\_Metres. As

## 56 Ada introduction: Part 2

Height\_Metres is of subtype Metres, a range check is performed on the assigned value. No conversion is required when Height\_Metres is output as its type Metres is a subtype of Float.

An example of a user's interaction with the program is shown below:

```
Enter person's height in Inches 73.0
Height in Metres is              1.85
```

### 4.18 Relational operators

The logical comparison operators are:

=	equal
/=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

The relational operators are used to establish the truth of a relationship between two values. The result is of type Boolean. For example:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Temperature : Integer;  --Temperature in Centigrade
  Hot         : Boolean;  --Is it hot
begin
  Get( Temperature );
  if Temperature > 24 then
    Put("It's warm"); New_Line;
  end if;
  Hot := Temperature > 30;
  if Hot then
    Put("It's hot"); New_Line;
  end if;
end Main;
```

#### 4.18.1 Boolean operators

Boolean values may be combined with the following operators:

<b>and</b>	logical and	Note: Both LHS and RHS evaluated
<b>or</b>	logical or	Note: Both LHS and RHS evaluated
<b>and then</b>	logical and	Note: RHS only evaluated if LHS TRUE
<b>or else</b>	logical or	Note: RHS only evaluated if LHS FALSE
<b>xor</b>	Exclusive or, True xor False => True   False xor True => True True xor True => False   False xor False => False	

For example, the following program prints a message on Christmas day.

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Month, Day : Integer;  --Date
begin
  Get( Day ); Get( Month );
  if Day = 25 and Month = 12 then
    Put("Happy Christmas"); New_Line;
  end if;
end Main;
```

By using **and then** or **or else** only the minimal evaluations will be performed to determine the truth of the Boolean expression. For example, the following two fragments of code are equal in effect:

```
if Month = 2 and then Day = 29 then
  -- The 29th of February
end if;
```

```
if Month = 2 then
  if Day = 29 then
    -- The 29th of February
  end if;
end if;
```

*Note: The RHS of the condition will only be evaluated if month = 2 is true.  
In some cases the correct evaluation of the RHS of an **and** or **or** Boolean operator will depend on the evaluation of the LHS of the operator.*

Section B.4.1, Appendix B contains a list of the priority of all the operators.

#### 4.18.2 Monadic Boolean operators

The inverse of a Boolean value is obtained by using the operator:

<b>not</b>	not
------------	-----

This delivers the inverse of the Boolean expression or Boolean value. For example:

```
if not ( Month = 2 ) then
  Put("Not February"); New_Line;
end if;
```

*Note: The brackets are required as = has a lower priority than not.*

#### 4.19 Bitwise operators

These are used for operating on modular quantities. Most programs will only occasionally require the use of these operators.

<b>and</b>	bitwise and
<b>or</b>	bitwise or
<b>xor</b>	bitwise xor
<b>not</b>	Inverse of bit pattern

For example, using the declarations:

## 58 Ada introduction: Part 2

```
K : constant := 1024;
type Word16 is mod 64 * K;
Pattern : Word16;
```

the following code:

- sets the top nibble of the two byte word `Pattern` to zero.

```
Pattern := Pattern and 16#FFF#;
```

- sets bit 9 in the two byte word `Pattern` to 1.

```
Pattern := Pattern or 2#0000001000000000#;
```

*Note: The constant to base 16 is 16#FFF# and to base 2 is 2#0000001000000000#. Section B.4, Appendix B describes how to declare constants to different bases.*

- Flips bit 9 in the two byte word `Pattern`. If bit 9 were a 1 it would now be a 0 and if it were a 0 it would now be a 1.

```
Pattern := Pattern xor 2#0000001000000000#;
```

- Inverts the bits in the two byte word `Pattern`.

```
Pattern := not Pattern;
```

### 4.20 Self-assessment

- Why is it not always appropriate to hold a value using an instance of a `Float`?
- How in a program can you find out the smallest value that can be stored in an instance of `Long_Integer`?
- What are the benefits of user defined types and subtypes in a program?
- Why does the following program fail?

```
procedure Main is
  type Miles is new Integer range 0 .. 100;
  type Kilometres is new Integer range 0 .. 100;
  London_Brighton : Miles := 50;
  To_Brighton      : constant Kilometres := 2;
  Distance_To_London : Miles;
begin
  Distance_To_London := London_Brighton + To_Brighton;
end Main;
```

- What is the difference between a type and a subtype?
- Why are the concepts of universal integer and universal float important?
- How can you convert a value of one type to that of another?
- Using the type declarations:



```

type Miles      is new Integer range 0 .. 10_00;
type Kilometres is range 0 .. 100;

```

what are the ranges of the following variables:

```

London_Brighton : Kilometres'Base;
London_New_York  : Miles'Base;

```

- How can the use of enumerations help improve a program's clarity?

## 4.21 Exercises

Construct the following programs using types and subtypes where appropriate:

- *Is prime*  
A program to say if a number is prime. A prime number is a positive number which is divisible by only 1 and itself.
- *Series*  
A program to print out numbers in the series 1 1 2 3 5 8 13 ... until the last term is greater than 10000.
- *Times table general case*  
Write a program to print a times table for any positive number.

You may wish to use the following approach.

The input procedure `get` may take its input from a string. The following statement: `get ( argument(1), number, last );` will convert the number held as characters in the string `'argument(1)'` into an integer number in the variable `number`. The argument `last` denotes the position in the string of the last character processed.

- *Temperature*  
A program to convert a Fahrenheit temperature to Centigrade. The formula for converting between Fahrenheit and Centigrade is:  
$$\text{Centigrade temperature} = (\text{Fahrenheit temperature} - 32)/1.8$$
- *Weight*  
A program to convert a person's weight input in pounds to kilograms. Assume that there are 2.2046 pounds in a kilogram.
- *Grades*  
A program to read in a student's name of 20 characters followed by his/her exam mark. The output to be the student's name followed by grade. For example, marks in the range 10070 get an A grade, 60—69 a B grade, 50—59 a C grade, 40—49 a D grade and 0—39 an F grade. Thus if the input was:

Andy	74
Bob	46
Charles	56
Dave	67

the output would be:

Andy	A
Bob	D
Charles	C
Dave	B

## 5 Procedures and functions

This chapter introduces procedures and functions. They allow a programmer the ability to abstract code into subprogram units that then may be re-used in different parts of a program or even other programs. This re-use of code, however, is at a very basic level. Other mechanisms, in particular the **package**, allow a much greater flexibility in promoting code re-use in programs.

### 5.1 Introduction

A function or procedure is a grouping together of one or more Ada statements into a subprogram unit, which can be called and executed from any appropriate part of a program. Functions and procedures allow a programmer a degree of abstraction in solving a problem. However, related procedures and functions are more powerfully used when combined with related data items to form a class. The concepts and uses of a class are discussed fully in Chapter 6.

### 5.2 Functions

A function is a subprogram unit that transforms its input value or values into a single output value. For example, a function to convert a distance in miles to Kilometres is shown below:

```
type Miles      is digits 8 range 0.0 .. 25_000.0;
type Kilometres is digits 8 range 0.0 .. 50_000.0;

function M_To_K_Fun(M:in Miles) return Kilometres is
  Kilometres_Per_Mile : constant := 1.609_344;
begin
  return Kilometres( M * Kilometres_Per_Mile );
end M_To_K_Fun;
```

*Note: The function's parameters may only import data into the function: they may not be used to export information back to the caller's environment. Thus, a function is a unit of code that transforms its input into a new value that is returned to the caller. As the parameter M can only be used to import data into the function, it may not be written to.*

The major components of the above function are illustrated in Figure 5.1.

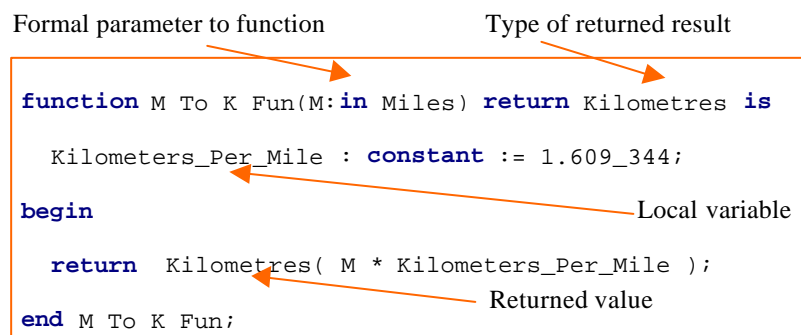


Figure 5.1 Major components of a function.

The function `M_To_K_Fun` is used in the following program that will print a conversion table for miles to kilometres. In Ada, a function or procedure may be declared in the declaration section of a procedure or function. By using this technique, the types for `Miles` and `Kilometres` can be made visible to both the function `M_To_K_Fun` and the main body of code that implements the printing of the conversion table.

```
with Ada.Text_IO, Ada.Float_Text_IO;
use  Ada.Text_IO, Ada.Float_Text_IO;
procedure Main is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;

  function M_To_K_Fun(M:in Miles) return Kilometres is
    Kilometres_Per_Mile : constant := 1.609_344;
  begin
    return Kilometres( M * Kilometres_Per_Mile );
  end M_To_K_Fun;

  No_Miles : Miles;

begin
  Put( "Miles  Kilometres" ); New_Line;
  No_Miles := 0.0;
  while No_Miles <= 10.0 loop
    Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put( "      " );
    Put( Float( M_To_K_Fun( No_Miles ) ), Aft=>2, Exp=>0 );
    New_Line;
    No_Miles := No_Miles + 1.0;
  end loop;
end Main;
```

When compiled, and run the above program will print the following results:

Miles	Kilometres
0.00	0.00
1.00	1.61
2.00	3.22
3.00	4.83
4.00	6.44
5.00	8.05
6.00	9.66
7.00	11.27
8.00	12.87
9.00	14.48
10.00	16.09

### 5.2.1 Local variables

When a variable is declared inside a function, its lifetime is that of the function. When the function is entered, space for any local variables is created automatically on a run-time stack. On exit from the function, the space created for the local variables is returned to the system.

### 5.2.2 Separate compilation of functions

It is possible to compile the above function separately. However, if this is done the same types for `Miles` and `Kilometres` must be used in both the main program and the function. One way of ensuring this is to use a package that acts as a container for the types. This package is then made visible to both program units. The following program illustrates this idea.

Firstly, the package that contains the types `Miles` and `Kilometres` is constructed.

## 62 Procedures and functions

```
package Pack_Types is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;
end Pack_Types;
```

Then this package is made visible to the function `M_To_K_Fun`.

```
with Pack_Types; use Pack_Types;
function M_To_K_Fun(M:in Miles) return Kilometres is
  Kilometres_Per_Mile : constant := 1.609_344;
begin
  return Kilometres( M * Kilometres_Per_Mile );
end M_To_K_Fun;
```

Finally, in the main procedure in addition to the normal input and output packages the package `Pack_Types` and the function `M_To_K_Fun` are made visible.

```
with Ada.Text_Io, Ada.Float_Text_Io, Pack_Types, M_To_K_Fun;
use Ada.Text_Io, Ada.Float_Text_Io, Pack_Types;
procedure Main is
  No_Miles : Miles;
begin
  Put("Miles Kilometres"); New_Line; No_Miles := 0.0;
  while No_Miles <= 10.0 loop
    Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put(" ");
    Put( Float( M_To_K_Fun( No_Miles ) ), Aft=>2, Exp=>0 );
    New_Line; No_Miles := No_Miles + 1.0;
  end loop;
end Main;
```

*Note:* It is only required to **with** the function `M_To_K_Fun`. It would be an error to **use** the function `M_To_K_Fun`.

### 5.3 Procedures

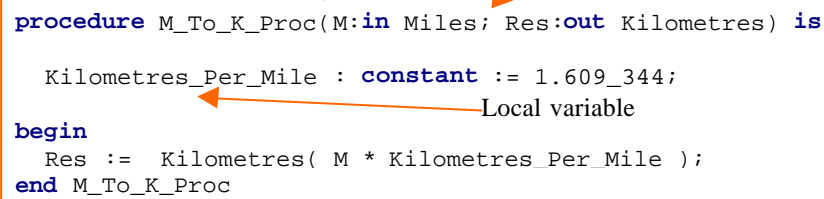
A procedure is a program unit that, unlike a function, does not return a result. If information has to be returned to the calling environment this is done instead by writing to a formal parameter that has been declared as mode **out**. Writing to the formal parameter of a procedure updates the value of the actual parameter passed to the procedure. The full implications of modes of a parameter are discussed in Section 5.5.

```
type Miles      is digits 8 range 0.0 .. 25_000.0;
type Kilometres is digits 8 range 0.0 .. 50_000.0;
procedure M_To_K_Proc(M:in Miles; Res:out Kilometres) is
  Kilometres_Per_Mile : constant := 1.609_344;
begin
  Res := Kilometres( M * Kilometres_Per_Mile );
end M_To_K_Proc;
```

*Note:* A procedure can only export values back to the caller's environment by writing to a parameter that has mode **out** or **in out**.

The major components of a procedure are illustrated in Figure 5.2.

Formal parameters to procedure



```

procedure M_To_K_Proc(M:in Miles; Res:out Kilometres) is

    Kilometres_Per_Mile : constant := 1.609_344;

begin
    Res := Kilometres( M * Kilometres_Per_Mile );
end M_To_K_Proc
  
```

Figure 5.2 Components of an Ada procedure.

This procedure may then be used in a program as follows:

```

with Ada.Text_IO, Ada.Float_Text_IO;
use   Ada.Text_IO, Ada.Float_Text_IO;
procedure Main is
    type Miles is digits 8 range 0.0 .. 25_000.0;
    type Kilometres is digits 8 range 0.0 .. 50_000.0;

    procedure M_To_K_Proc(M:in Miles; Res:out Kilometres) is
        Kilometres_Per_Mile : constant := 1.609_344;
    begin
        Res := Kilometres( M * Kilometres_Per_Mile );
    end M_To_K_Proc;

    No_Miles : Miles;
    No_Km     : Kilometres;

begin
    Put("Miles Kilometres"); New_Line;
    No_Miles := 0.0;
    while No_Miles <= 10.0 loop
        Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put(" ");
        M_To_K_Proc( No_Miles, No_Km );
        Put( Float( No_Km ), Aft=>2, Exp=>0 );
        New_Line;
        No_Miles := No_Miles + 1.0;
    end loop;
end Main;
  
```

when run, this program would produce the same output as the previous program that used a function to convert miles to Kilometres.

### 5.3.1 Separate compilation of procedures

The same strategy as seen in Section 5.2.2 can be used to separately compile a procedure.

## 5.4 Formal and actual parameters

In describing the parameter passing mechanism the following terminology is used:

Terminology	Commentary
Formal parameter	The parameter used in the declaration of a function or procedure. For example, in the function <code>M_To_K_Fun</code> the formal parameter is <code>M</code> .
Actual parameter	The object passed to the function or procedure when the function or procedure is called. For example, in the procedure <code>M_To_K_Proc</code> the actual parameters are <code>No_Miles</code> and <code>No_Km</code> . An expression may also be passed as an actual parameter to a function or procedure, provided the mode of the formal parameter is not <b>out</b> (See Section 5.5).

In discussing functions and procedures it is important to distinguish between the actual parameter passed to a function and the formal parameter used in the body of the code of the function or procedure. This relationship is shown in Figure 5.3.

```
with Ada.Text_IO, Ada.Float_Text_IO;
use  Ada.Text_IO, Ada.Float_Text_IO;
procedure Main is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;

  function M_To_K_Fun(M in Miles) return Kilometres is
    Kilometres_Per_Mile : constant := 1.609_344;
  begin
    return Kilometres( M * Kilometres_Per_Mile );
  end M_To_K_Fun;

  No_Miles : Miles;

begin
  Put("Miles Kilometres"); New_Line;
  No_Miles := 0.0;
  while No_Miles <= 10.0 loop
    Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put(" ");

    Put( Float( M_To_K_Fun( No_Miles ) ), Aft=>2, Exp=>0 );

    New_Line;
    No_Miles := No_Miles + 1.0;
  end loop;
end Main;
```

Formal parameter to function

Actual parameter to function

Figure 5.3 Formal and actual parameters of a function.

## 5.5 Modes of a parameter to a function or procedure

In Ada, as in many languages, objects can be passed to a procedure in several different ways depending on how the object is to be accessed. The simplest and by far the safest mode to use, is **in**. This allows an object to be imported into the procedure, but the user is prevented by the compiler from writing to the object.

A procedure can export information to the actual parameter when the formal parameter is described by mode **out**. Naturally, for this to happen, the actual parameter's mode must allow the object to be written to. It must therefore not be an expression or an object that has a mode of **in** only.

A function in Ada however, is only allowed to have parameters of mode **in**. The different ways that a parameter may be passed to a function or procedure is summarized in the table below:

Mode	Allowed as a parameter to:	Effect
<b>in</b>	a function or a procedure.	The formal parameter is initialized to the contents of the actual parameter and may only be read from.
<b>in out</b>	only a procedure	The formal parameter is initialized to the contents of the actual parameter and may be read from or written to. When the procedure is exited, the new value of the formal parameter replaces the old contents of the actual parameter.
<b>out</b>	only a procedure	The formal parameter is <b>not</b> initialized to the contents of the actual parameter and may be read from or written to. When the procedure is exited, the new value of the formal parameter replaces the old contents of the actual parameter. In Ada 83 an <b>out</b> formal parameter may not be read from.

*Note: The implementation of the above for simple objects is usually performed by copying the contents of the object, whilst for large objects the compiler may implement this by using references to the actual object.*

### 5.5.1 Example of mode **in out**

A procedure swap which interchanges the contents of the actual parameters passed to it is as follows:

```

procedure Swap(First:in out Integer; Second:in out Integer) is
    Temp : Integer;
begin
    Temp := First;
    First := Second; Second := Temp;
end Swap;

```

### 5.5.2 Putting it all together

The function swap may then be used in a program as follows:

```

with Ada.Text_Io, Ada.Integer_Text_Io, Swap;
use Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
    Books_Room_1 : Integer;
    Books_Room_2 : Integer;
begin
    Books_Room_1 := 10; Books_Room_2 := 20;
    Put("Books in room 1 ="); Put(Books_Room_1); New_Line;
    Put("Books in room 2 ="); Put(Books_Room_2); New_Line;
    Put("Swap around"); New_Line;
    Swap(Books_Room_1, Books_Room_2);
    Put("Books in room 1 ="); Put(Books_Room_1); New_Line;
    Put("Books in room 2 ="); Put(Books_Room_2); New_Line;
end Main;

```

which when run produces:

```

Books in room 1 =      10
Books in room 2 =      20
Swap around
Books in room 1 =      20
Books in room 2 =      10

```

## 66 Procedures and functions

### 5.5.3 Summary of access to formal parameters

Formal parameter specified by: (using as an example an Integer formal parameter)	Write to formal parameter allowed	Read from formal parameter	Can be used as a parameter to
item: Integer	✗	✓	procedure or function
item: <b>in</b> Integer	✗	✓	procedure or function
item: <b>in out</b> Integer	✓	✓	procedure only
item: <b>out</b> Integer	✓	✓	procedure only

## 5.6 Recursion

Recursion is the ability of a procedure or function to make a call on itself from within its own code body. Whilst this initially may seem a strange idea, it can lead to very elegant code sequences that otherwise would require many more lines of code. In certain exceptional cases recursion is the only way to implement a problem.

An example of a recursive procedure to write a natural number using only character based output is sketched in outline below:

Write a natural number: (write\_natural)

- Split the natural number into two components
  - The first digit (remainder when number divided by 10)
  - The other digits (number divided by 10).

For example:

123 would be split into:  
3 (first digit)  
12 (other digits).

- If the other digits are greater than or equal to 10 then write the other digits by recursively calling the code to write a decimal number.
- Output the first digit as a character.

The sequence of calls made is

Call	Implemented as
write_natural( 123 )	write_natural(12); output first digit 3
write_natural( 12 )	write_natural(1); output first digit 2
write_natural( 1 )	output first digit 1

This process is diagrammatically expressed in Figure 5.4.

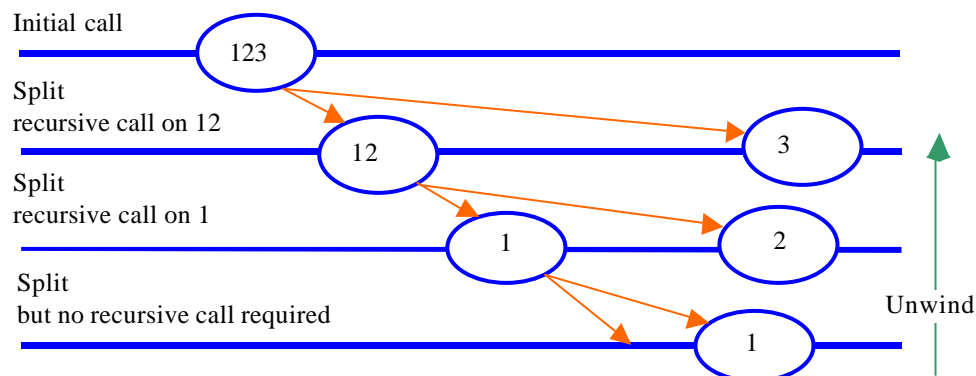


Figure 5.4 Illustration of recursive calls to print the natural number 123.



The process works by solving a small part of the problem, in this case how to output a single digit, then re-executing the code to solve the remainder of the problem, that is, to output the other digits. In this particular example, the recursive call is made before the solution of the remainder of the problem. This still works as the problem to be solved 'the number to be output' is reduced in size in each recursive call.

However, for recursion to work, the code must reduce the problem to be solved before recalling itself recursively. If this does not take place then endless recursive calls will ensue, which will cause eventual program failure when the system cannot allocate any more memory to support the recursion. Stack space is used on each recursive call to store any parameters or local variables plus the function / procedure support information.

### 5.6.1 The procedure `Write_Natural`

The procedure `write_natural`'s implementation is shown below:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Write_Natural( Num : Natural) is
  First_Digit  : Natural;      --Unit digit
  Other_Digits : Natural;      --All except first digit
begin
  First_Digit := Num rem 10;    --Split 1234 => 4
  Other_Digits := Num / 10;     --           => 123
  if Num >= 10 then            --Print other digits
    Write_Natural( Other_Digits ); --Recursive call
  end if;
  Put( Character'Val( First_Digit + Character'Pos('0') ) );
end Write_Natural;
```

### 5.6.2 Putting it all together

The function `Write_Natural` could be used in a program as follows:

```
with Ada.Text_IO, Write_Natural;
use Ada.Text_IO;
procedure Main is
begin
  Write_Natural( 123 );          New_Line;
  Write_Natural( 12345 );       New_Line;
end Main;
```

which when run would produce:

```
123
12345
```

## 5.7 Overloading of functions

Overloading is a process that allows several items providing different facilities to have the same name. The compiler chooses the appropriate definition to use from the context of its use.

This is best illustrated by an example where the overloaded item is a procedure. Firstly, three different procedures are defined which each have a different action. The action is to identify and print the contents of their single parameter.

## 68 Procedures and functions

```
with Ada.Integer_Text_Io;
use  Ada.Integer_Text_Io;
procedure Answer_Is( N:in Integer;
                    Message:in Boolean := True) is
begin
  if Message then Put("The answer = "); end if;
  Put( N, Width=>1 );
  if Message then New_Line; end if;
end Answer_Is;

with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Is_A_Int( An_Int:in Integer ) is
begin
  Put("The parameter is an Integer:  value = ");
  Put( An_Int, Width=>1 ); New_Line;
end Is_A_Int;
```

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Is_A_Float( A_Float:in Float ) is
begin
  Put("The parameter is a Float:      value = ");
  Put( A_Float, Aft=>2, Exp=>0 ); New_Line;
end Is_A_Float;
```

The individual procedures have unique names so that they can be identified and re-used in a program. This is a consequence of each procedure being a separate compilation unit. However, Ada allows the renaming of a procedure or function. By choosing the same name a user can overload a particular name with several different definitions. For example, a program unit can be written which renames the three different procedure names (Is\_A\_Int, Is\_A\_Float, Is\_A\_Char) with the same overloaded name Is\_A.

```
with Is_A_Int, Is_A_Float, Is_A_Char;
procedure Main is
  procedure Is_A( The:in Integer )   renames Is_A_Int;
  procedure Is_A( The:in Float )     renames Is_A_Float;
  procedure Is_A( The:in Character ) renames Is_A_Char;
begin
  Is_A( 'A' );
  Is_A( 123 );
  Is_A( 123.45 );
end Main;
```

*Note: It is possible to write several functions or procedures with the same name directly by using the package construct.*

When run this program would print the type and value of the argument passed to Is\_A.

```
The parameter is a Character: value = A
The parameter is an Integer:  value = 123
The parameter is a Float:     value = 123.450
```

Of course, for this to happen, the actual function called must be different in each case. The name Is\_A is overloaded by three different functions. The binding between the called function and its body is worked out by the compiler at compile-time using the signature of the different functions that have been overloaded to resolve any conflicts.

## 5.8 Different number of parameters

As the compiler can distinguish between overloaded names, several functions that deliver the maximum or larger of their parameters can be written. With re-use in mind the first function Max2 can be written which delivers the maximum of the two Integer parameters passed to it.

```
function Max2( A,B:in Integer ) return Integer is
begin
  if A > B then
    return A;          --a is larger
  else
    return B;          --b is larger
  end if;
end Max2;
```

This function Max2 can be re-used in a function Max3 that will deliver the larger of three parameters passed to it.

```
with Max2;
function Max3( A,B,C:in Integer ) return Integer is
begin
  return Max2( Max2( A,B ), C );
end Max3;
```

Then the following code can be written:

```
with Ada.Text_io, Ada.Integer_Text_Io, Max2, Max3;
use Ada.Text_io, Ada.Integer_Text_Io;
procedure Main is
  function Max(A,B:in Integer) return Integer renames Max2;
  function Max(A,B,C:in Integer) return Integer renames Max3;
begin
  Put("Larger of 2 and 3 is "); Put( Max(2,3) );   New_Line;
  Put("Larger of 2 3 4 is "); Put( Max(2,3,4) );   New_Line;
end Main;
```

*Note: The use of renames to overload the name Max with 2 distinct function definitions.*

which when run produces:

```
Larger of 2 and 3 is      3
Larger of 2 3 4 is      4
```

*Note: The overloading of names in an Ada program can provide a simpler interface for a programmer. However, the overuse of overloading can lead to programs that are difficult to maintain and debug.*

## 5.9 Default values and named parameters

If a default value is given to a parameter, then it may be omitted by a programmer when they write the call to the function or procedure.

For example, the function sum whose four parameters have a default value of zero returns the sum of these parameters. The procedure Answer\_Is prints the first parameter with an additional message when the second parameter has the default value True.

## 70 Procedures and functions

```
function Sum( P1:in Integer := 0;
              P2:in Integer := 0;
              P3:in Integer := 0;
              P4:in Integer := 0 ) return Integer is
begin
    return P1 + P2 + P3 + P4;
end Sum;

with Ada.Text_IO, Ada.Integer_Text_io;
use  Ada.Text_IO, Ada.Integer_Text_io;
procedure Answer_Is( N:in Integer;
                     Message:in Boolean := True ) is
begin
    if Message then Put("The answer = "); end if;
    Put( N, Width=>1 );
    if Message then New_Line; end if;
end Answer_Is;
```

*Note:* Formal parameters to the function Sum are given a default value of 0, if a value has not been supplied by a caller of the function.

Any actual parameter to a function or procedure may be specified either by position or by name. For example, the second formal parameter to the function Answer\_Is can be specified in the following ways:

```
Answer_Is( 27, True );           -- By position
Answer_Is( 27, Message => False ); -- By name
```

*Note:* If a parameter is specified by name, then all subsequent parameters must be specified by name.

### 5.9.1 Putting it all together

The procedures sum and Answer\_Is can be used in a program as follows:

```
with Sum, Answer_Is;
procedure Main is
begin
    Answer_Is( Sum );
    Answer_Is( Sum( 1, 2 ) );
    Answer_Is( Sum( 1, 2, 3 ) );
    Answer_Is( Sum( 1, 2, 3, 4 ), Message => False );
    New_Line;
end Main;
```

The code that is actually compiled for the procedure Main above is:

```
with Sum, Answer_Is;
procedure Main is
begin
    Answer_Is( Sum( 0, 0, 0, 0 ), True );
    Answer_Is( Sum( 1, 2, 0, 0 ), True );
    Answer_Is( Sum( 1, 2, 3, 0 ), True );
    Answer_Is( Sum( 1, 2, 3, 4 ), False );
    New_Line;
end Main;
```

which is more complex for the writer to construct and for a maintainer to follow.

*Note: The syntax `Message => False` is used for specifying a parameter by name. The syntax for the call of the function `sum` when no parameters are specified has no brackets. This can lead to confusion as a reader of the code would not know from the context if `sum` was a simple variable or a function call.*

When run the above program produces:

```
The answer = 0
The answer = 3
The answer = 6
10
```

*Note: Procedures and functions may be nested. The advantage of this approach is that a single program unit may be decomposed into several smaller units and yet hide the internal decomposition.*

## 5.10 Self-assessment

- From a programming safety point of view, what are the advantages of passing parameters by mode **in** rather than by mode **in out**?
- Why is parameter passing using mode **in out** required, if values can already be passed back as the result of a function?
- When might overloading of function names be used?
- What are the disadvantages of overloading names in a program?
- What is the difference between a function and a procedure? Can a procedure which exports several values through the parameter mechanism be easily made into a function? Explain your answer.

## 5.11 Exercises

Construct the following subprograms and programs:

- The function `What_Is_Char` which accepts as a parameter a character and returns its 'type' as defined by the enumeration :  
`type Char is ( Digit, Punctuation, Letter, Other_Ch );`
- Using the function `What_Is_Char` write a program to count the number of digits, letters and punctuation characters in a text file.  
  
 Hint:  
 Nest the function `What_Is_Char` inside a procedure that processes input received by the program.
- Write a procedure `Order3` which takes three parameters of type `Float` and re-orders the parameters into ascending order.

## 72 *Procedures and functions*

- Write a program that finds the average of three rainfall readings taken during the last 24 hours. The program should print the average of the samples plus the readings in ascending order. For example, if the input data was:  
4.0 6.0 5.0  
then the program should produce output of the form:  
Rainfall average is : 5.00  
Data values (sorted) are : 4.00 5.00 6.00

## 6 Packages as classes

This chapter introduces the package construct. A package is an elegant way of encapsulating code and data that interact together into a single unit. A package may be used in a variety of ways. This chapter, however, will promote its use to define classes.

### 6.1 Introduction

The world in which we live is populated by many devices and machines that make everyday living easier and more enjoyable. The TV, for instance, is viewed by almost every person in the country, yet few understand exactly what happens inside 'the box'. Likewise, there are many millions of motorists who drive regularly and do not need a detailed knowledge of the workings of a car to make effective use of it.

To many people, their knowledge of a car is as shown in Figure 6.1. The exact details of what happens inside the car are not important for most day-to-day driving.

In essence the world is populated with many objects which have an interface that allows the humblest of persons to make effective use of the item. We sometimes criticize the interfaces as being ineffective and difficult to use, yet in most cases we would prefer to use the objects as they stand, rather than having to perform the task by other means.

Likewise in the software world, there are objects that a user or programmer can make effective use of without having to know how the object has been implemented. On a very simple level an Ada program may declare objects to hold floating point numbers, which can then be used with arithmetic operations to sum, multiply, etc. these values. Most programmers however, do not know the exact details of how these operations are performed; they accept the interface provided by the programming language.

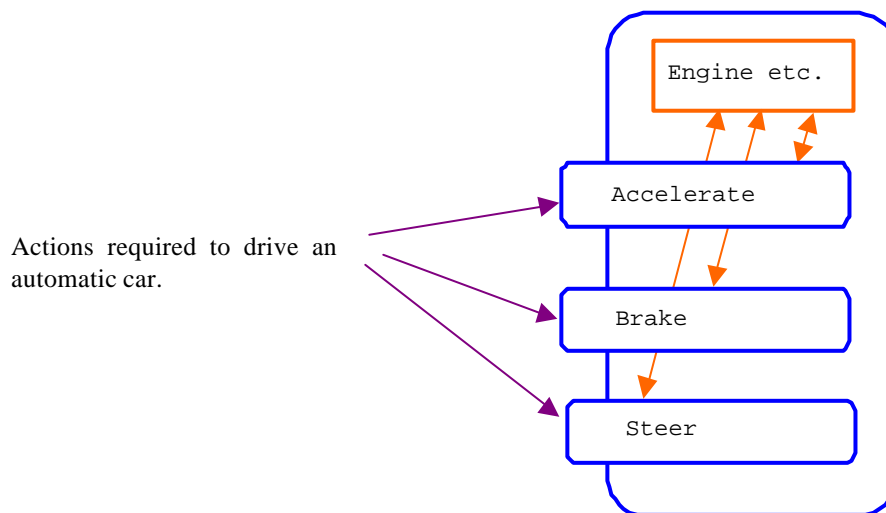


Figure 6.1 Basic understanding of working of an automatic car.

The details of what happens inside the car are not important for most day-to-day driving.

In essence the world is populated with many objects which have an interface that allows the humblest of persons to make effective use of the item. We sometimes criticize the interfaces as being ineffective and difficult to use, yet in most cases we would prefer to use the objects as they stand, rather than having to perform the task by other means.

## 74 Packages as classes

Likewise in the software world, there are objects that a user or programmer can make effective use of without having to know how the object has been implemented. On a very simple level an Ada program may declare objects to hold floating point numbers, which can then be used with arithmetic operations to sum, multiply, etc. these values. Most programmers however, do not know the exact details of how these operations are performed; they accept the interface provided by the programming language.

At one point it was fashionable for programming languages to provide a rich set of data types. The designers of these languages hoped the data types provided would be adequate for all occasions. The problem was, and still is, that no one language could ever hope to provide all the different types of item that a programmer may need or wish to use.

Ada gives a programmer the ability to declare new data types, together with a range of operations that may be performed on an instance of the type. Naturally, a programmer may also use types and operations on these types that have been defined by other programmers.

### 6.2 Objects, messages and methods

A car can be thought of as an object. The car contains complex details and processes that are hidden from the driver. For example, to make the car go faster the driver presses the accelerator pedal. The car receives the message 'go faster' and evokes an internal method to speed up the engine.

In the above description of driving a car many object-oriented ideas have been used. These ideas are as follows:

object	An item that has a hidden internal structure. The hidden structure is manipulated or accessed by messages sent by a user.
message	A request sent to the object to obey one of its methods.
method	A set of actions that manipulates or accesses the internal state of the object. The detail of these actions is hidden from a user of the object.

### 6.3 Objects, messages and methods in Ada

In Ada an object is an instance of either a user-defined type or an instance of one of the in-built types.

An object for a user-defined type can be imagined diagrammatically as Figure 6.2.

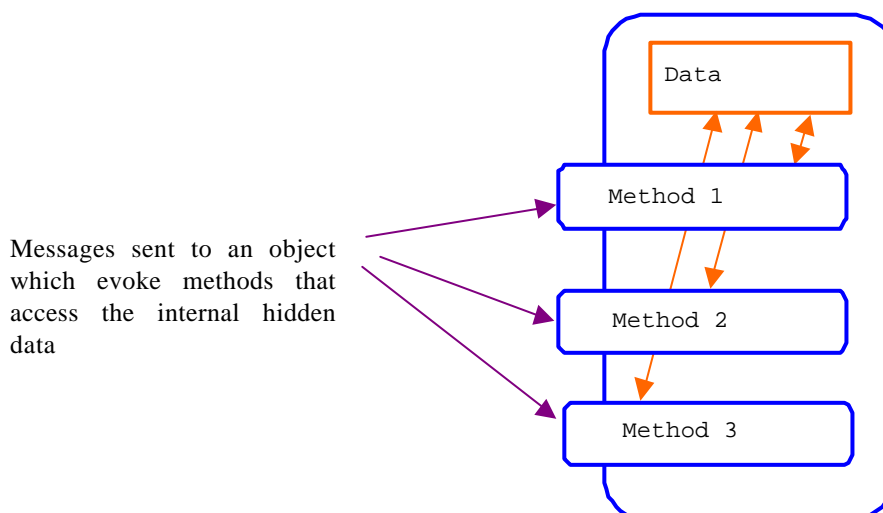


Figure 6.2 Diagrammatic representation of an object.

A message is implemented as either a procedure or function call, the body of which is the method that is evoked when the message is sent to the object. The user of the object has no knowledge of the implementation code contained in the body of the procedure or function.

*Note: The idea of binding code and data together in a unit that does not allow direct access to the data is often referred to as encapsulation.*



### 6.3.1 An object for a bank account

Before looking in detail at the implementation of an object that represents a bank account, it is appropriate to consider the messages that might be sent to such an object. For a very simple type of bank account these messages would be:

- Deposit money into the account.
- Withdraw money from the account.
- Deliver the account balance.

The following program demonstrates the sending of these messages to an instance of an Account.

```
with Ada.Text_IO, Class_Account, Statement;
use Ada.Text_IO, Class_Account;
procedure Main is
  My_Account: Account;
  Obtain    : Money;
begin
  Statement( My_Account );

  Put("Deposit £100.00 into account"); New_Line;  --Deposit
  Deposit( My_Account, 100.00 );
  Statement( My_Account );

  Put("Withdraw £80.00 from account"); New_Line;  --Withdraw
  Withdraw( My_Account, 80.00, Obtain );
  Statement( My_Account );

  Put("Deposit £200.00 into account"); New_Line;  --Deposit
  Deposit( My_Account, 200.00 );
  Statement( My_Account );
end Main;
```

Note: The package *Class\_Account* contains:

- The definition of the type *Account* plus the definition of the operations allowed on an instance of an *Account*;
- The subtype *Money* used to define some of the parameters to messages sent to an instance of *Account*.
- The procedure *Statement* is used to simplify the printing of a mini statement of the balance held in the account.

The messages sent to an instance of an *Account* are: *Deposit*, *Withdraw*, *Balance*. For example, to deposit £100 into *My\_Account* the following procedural notation is used:

```
Deposit( My_Account, 100.00 );
```

This should be read as: send the message *deposit* to the object *My\_Account* with an actual parameter of 100.00.

To withdraw money from the account a programmer would send the message *Withdraw* to the object *My\_Account* with two parameters, the amount to withdraw and a variable that is to be filled with the amount actually withdrawn. The implementation of the method will check that the person has sufficient funds in their account to allow the transaction to take place. This is written as:

```
Withdraw( My_Account, 80.00, obtain );
```

Note: In reality the method is a normal Ada procedure that is passed as parameters, the object on which the action is to take place, plus any additional information as successive parameters.

## 76 Packages as classes

### 6.3.2 The procedure Statement

The procedure Statement is responsible for printing a mini-statement about the contents of an account. This procedure is defined as follow:

```
with Ada.Text_IO, Ada.Float_Text_IO, Class_Account;  
use   Ada.Text_IO, Ada.Float_Text_IO, Class_Account;  
procedure Statement( An_Account:in Account ) is  
begin  
    Put("Mini statement: The amount on deposit is £" );  
    Put( Balance( An_Account), Aft=>2, Exp=>0 );  
    New_Line(2);  
end Statement;
```

*Note: The use of the method Balance to access the amount of money in the account.*

### 6.3.3 Putting it all together

When compiled with an appropriate package body, the above program unit when run will produce the following results:

```
Mini statement: The amount on deposit is £ 0.00  
Deposit £100.00 into account  
Mini statement: The amount on deposit is £100.00  
Withdraw £80.00 from account  
Mini statement: The amount on deposit is £20.00  
  
Deposit £200.00 into account  
Mini statement: The amount on deposit is £220.00
```

### 6.3.4 Components of a package

The package construct in Ada is split into two distinct parts. These parts contain the following object-oriented components:

Ada package component	Object-oriented component
Specification	The type used to elaborate the object, plus the specification of the messages that can be sent to an instance of the type.
Implementation	Implementation of the methods that are evoked when a message is sent to the object.

### 6.3.5 Specification of the package

The specification defines what the packages does, but not how it performs the implementation. It is used by the Ada compiler to check and enforce the correct usage of the package by a programmer.

The specification is split into two distinct parts: a public part and a private part. The public part defines the messages that may be sent to an instance of an Account, whilst the private part defines the representation of the type Account.

As the representation of Account is defined in the private part of the specification, a user of an instance of an Account will not be allowed to access the internal representation. The user however, is allowed to declare and, depending on the description of the type, assign and compare for equality and inequality. In this case the description of the type is private and a user is allowed to declare, assign and compare for equality and inequality instances of Account.

```

package Class_Account is

  type Account is private;
  subtype Money is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Deposit ( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account; Amount:in Pmoney;
                      Get:out Pmoney );
  function Balance ( The:in Account ) return Money;

private
  type Account is record
    Balance_Of : Money := 0.00;      --Amount in account
  end record;
end Class_Account;

```

The component parts of the specification are illustrated in Figure 6.3.

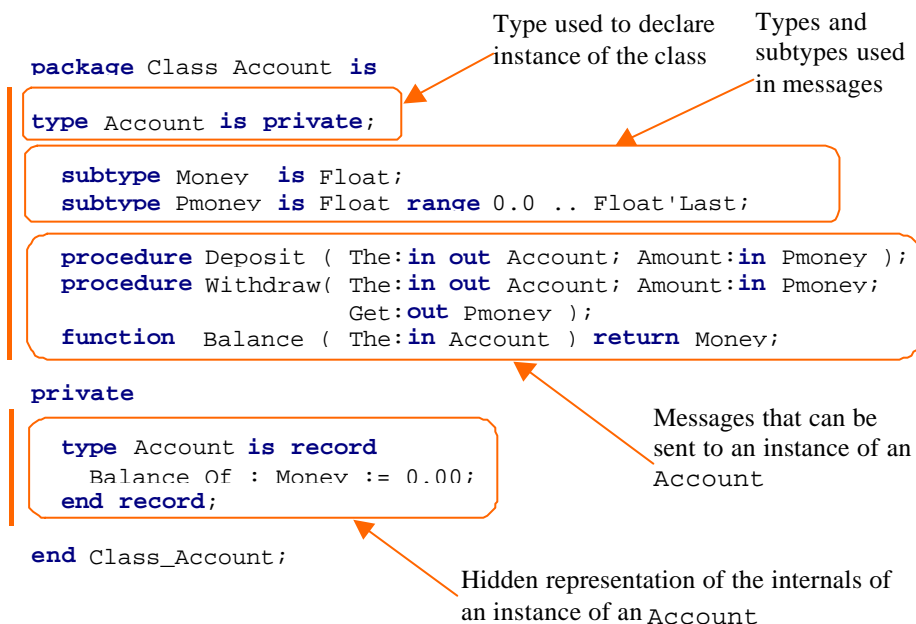


Figure 6.3 Components of the specification part of a package.

The representation of Account (which is defined in the private part) is a subtype of a Float that will have an initial value of 0.00. An Ada record groups together several type declarations into a single named type. In this case the record type Account declares a single object called Balance\_Of. The **record** type is more fully discussed in Section 7.1.

*Note:* The **type** Account is defined in the public part of the specification as **private**. This means that a user of an instance of the type cannot access the internal contents of the object. Apart from the methods defined in the public part of the specification the only operations that a user can perform on this object is to assign it to another instance of an Account or compare two instance of an Account with either = or /=.

## 78 Packages as classes

### 6.3.6 A class diagram showing a class

A class diagram for the class Account using the UML notation is illustrated in Figure 6.4.

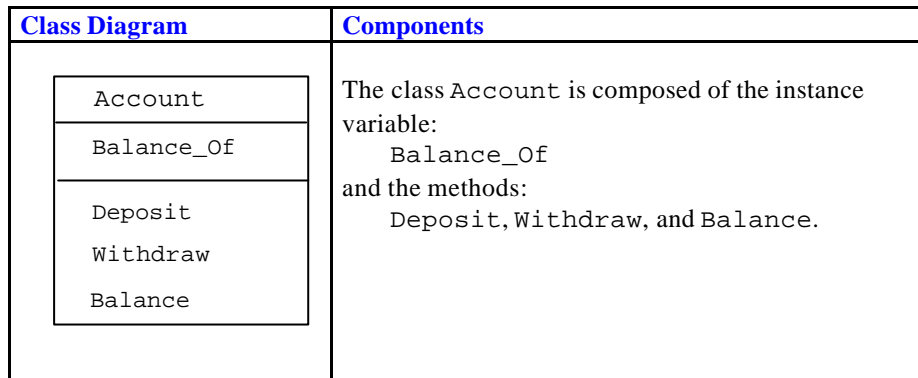


Figure 6.4 Class diagram for the class Account.

### 6.3.7 Representation of the balance of the account

The package Class\_Account represents internally the balance of the account as a subtype of a Float. A Float is an inexact way of representing numbers, as only the most significant digits of the number will be stored. Instances of a type declaration of the form **type Money is delta 0.01 digits 8;** would provide a more reliable way of holding the balance of the account. However, this would require instantiation of a specific package for input and output of objects of this type. To simplify the presentation of this package the representation of the balance of the account is implemented as a subtype of a Float. After reading Chapters 14 and 18 the reader may wish to re-implement this package as a generic package which uses an instantiation of Ada.Text\_IO.Decimal\_io.

### 6.3.8 Implementation of the package

The implementation of the package Class\_Account is as follows:

```
package body Class_Account is

  procedure Deposit ( The:in out Account; Amount:in Pmoney ) is
  begin
    The.Balance_Of := The.Balance_Of + Amount;
  end Deposit;

  procedure Withdraw( The:in out Account; Amount:in Pmoney;
                      Get:out Pmoney ) is
  begin
    if The.Balance_Of >= Amount then
      The.Balance_Of := The.Balance_Of - Amount;
      Get := Amount;
    else
      Get := 0.00;
    end if;
  end Withdraw;

  function Balance( The:in Account ) return Money is
  begin
    return The.Balance_Of;
  end Balance;

end Class_Account;
```

*Note: The use of the overloaded name Get as a parameter to the procedure Withdraw.*

The body of the package contains the definition of the procedures and functions defined in the specification part of the package. In accessing the `Balance_Of` contained in an instance of `Account` the `.` notation is used. For example, in the function `balance` the result returned is obtained using the statement `'return The.Balance_Of;'`. The `.` notation is used to access a component of an instance of a record type. In this case, the instance of the record type is the object `The` and the component of the object is `Balance_Of`.

### 6.3.9 Terminology

The following terminology is used to describe the components of a class.

Terminology	Example: in class <code>Account</code>	Explanation
Instance attribute	<code>Balance_Of</code>	A data component of an object. In Ada this will be a member of the type that is used to declare the object.
Instance method or just method	<code>Deposit</code>	A procedure or function used to access the instance attributes in an object.

*Note: The terminology comes from the language Smalltalk.*

## 6.4 The package as seen by a user

A user will normally only have access to the specification part of a package. This provides a specification of the messages that can be sent to an object but does not show how the methods invoked by the messages have been implemented. The implementation part will not normally be available, the implementor normally providing only a compiled version of the package.

Unfortunately the details of the private type will normally be visible, though they cannot be accessed.

*Note: The details of the private type can be made invisible to a user, but this involves some complexity. One approach to this is shown in Section 15.5.*

## 6.5 The package as seen by an implementor

When building a package the implementor should ensure:

- That a user of the package can make effective use of its facilities.
- That the only visible components are:
  - (a) The messages that can be sent to an object.
  - (b) The private type declaration that is used to elaborate an object.

The visibility hierarchy for the package `Class_Account` is shown in Figure 6.5.

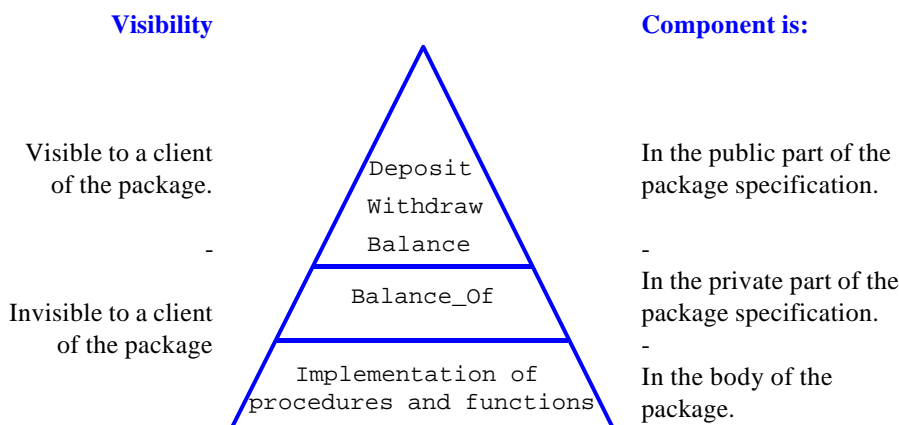


Figure 6.5 Visibility of methods and instance attributes of the package `Class_Account`.

## 6.6 The class

In object-oriented programming one of the important ideas is that of the class. A class is the collective name for all objects that share the same structure and behaviour. For example, in a program dealing with bank transactions, all the objects that represent a particular type of bank account would belong to the same class.

The class construct in a programming language is used to define objects that share a common structure and behaviour. Ada does not have a class construct.

However, Ada's package construct can be used to simulate the class construct found in other object-oriented programming languages. For example, a class `Account` is defined by the following package:

```
package Class_Account is
    type Account is private;

    procedure Deposit ( The:in out Account; Amount:in Pmoney );
    -- Other methods in the class
private
    type Account is record
        Balance_Of : Money := 0.00;      --Amount in account
    end record;
end Class_Account;
```

```
package body Class_Account is
    -- Implementation of the procedures and functions
end Class_Account;
```

In defining a class, I use the following conventions:

- The class is defined in terms of a package which has the class name prefixed with `Class_`.
- The package has a single private type which takes the class name and is used to declare instances of the class. Hence all instances of the class will share the same structure and behaviour.
- Procedures and functions are used to define the behaviour of the class. The first formal parameter to the procedure or function is an instance of the class.
- The implementation of the private type is defined as a **record** type, the components of which define the structure of the class.

## 6.7 Clauses `with` and `use`

The clauses `with Ada.Text_IO;` `use Ada.Text_IO;` make available the contents of the package `Ada.Text_IO` to the following program unit. The package `Ada.Text_IO` contains definitions for performing input and output on character and string objects. The exact effect of these clauses are as follows:

- **with** `Ada.Text_IO;`  
Make available to the unit all the public components of the package. However, when components of the package are used in a program they must be prefixed with the package name.
- **use** `Ada.Text_IO;`  
Permit public components of the package to be used without having to prefix their name with that of the package name.

Thus without the **use** clause, the program to process bank transactions would become:

```
with Ada.Text_Io, Class_Account, Statement;
procedure Main is
  My_Account: Class_Account.Account;
  Obtain     : Class_Account.Money;
begin
  Statement( My_Account );

  Ada.Text_Io.Put("Deposit £100.00 into account");
  Ada.Text_Io.New_Line;
  Class_Account.Deposit( My_Account, 100.00 );
  Statement( My_Account );
```

```
Ada.Text_Io.Put("Withdraw £80.00 from account");
Ada.Text_Io.New_Line;
Class_Account.Withdraw( My_Account, 80.00, Obtain );
Statement( My_Account );

Ada.Text_Io.Put("Deposit £200.00 into account");
Ada.Text_Io.New_Line;
Class_Account.Deposit( My_Account, 200.00 );
Statement( My_Account );
end Main;
```

*Note: Some program guidelines will ban the use of a **use** clause.*

### 6.7.1 To use or not to use the use clause

Using a use clause	Not using a use clause
Program writing is simplified.	A program must explicitly state which package the component is taken from.
Confusion may arise as to which package the item used is a component of.	This can reduce the possibility of program error due to accidental misuse.

### 6.7.2 The package Standard

In Ada the clause '**with** Standard; **use** Standard;' is implicitly added to the start of each program unit. The specification for the package Standard is shown in Appendix C, Section C.4. This package contains definitions for the operators +, -, \*, /, etc. However, the package Standard cannot be directly changed by a programmer.

### 6.7.3 Positioning of with and use in a package declaration

Any **with** and **use** clauses that appear before a specification of a package are implicitly included for the body of the package. If components of the with'ed and used packages are only used in the body of a package, then the clauses **with** and **use** need only be specified for the body. For example, if in the class Account only the body of the package used the package Pack\_Useful then it could be written as:

## 82 Packages as classes

```
package Class_Account is
  -- rest of specification
end Class_Account;

with Pack_Useful;
use Pack_Useful;
package body Class_Account_Other is
  -- rest of implementation
end Class_Account;
```

One consequence of this approach is that the user of the package need not know what packages are used by the implementation code.

### 6.7.4 Conflict in names in a package

A user may wish to use packages that contain items with the same name. For example, a user of the class `Class_Account` also requires to use the class `Class_Account_Other`. In both classes the name of the type that is used to declare an instance of the class is `Account`. By prefixing the type name with the package name the conflict is resolved.

```
with Class_Account, Class_Account_Other;
use Class_Account, Class_Account_Other;
procedure Main is
  My_Account      :Class_Account.Account;
  Other_Account   :Class_Account_Other.Account;
begin
  Deposit( My_Account,100.00 );--statement in Class_account
  Deposit( My_Account,100.00 );--statement in Class_account_other
end Main;
```

*Note: Overload resolution is used to resolve which package the procedure statement is implemented in.*

## 6.8 Mutators and inspectors

The methods in a class can either be inspectors or mutators. The role of each of these methods is illustrated in the table below:

Method is a	Role of method	Example from class Account
Inspector	Does not change the state of the object.	Balance
Mutator	Changes the state of the object.	Withdraw Deposit



## 6.9 Type private

In the specification of the class Account seen in Section 6.3.4, the type of Account is **private**. This restricts a user of an instance of the type to the following operations:

- Elaboration of an instance of the type.
- Assigning an instance of the type to another instance of the type.
- Comparing instances of the type for equality or inequality.
- Passing an instance of the type to a procedure of function.

A user of the type is prevented from reading or changing the internal contents other than by the actions of methods in the class.

### 6.9.1 Type limited private

A user can be further restricted in the operations that they can perform on an instance of Account by declaring it as **limited private**. This removes the user's ability to assign or compare an instance of the type by default. Naturally if in the class Account the comparison operations for equality or inequality are provided, then these definitions will be used and will override the restriction. For example, if in the class Account the type Account were defined as **limited private**, a user of an instance of an Account would be prevented from writing the following:

```
with Class_Account;
use Class_Account;
procedure Main is
  My_Account : Account;
  Other_Account : Account;
  Obtain : Pmoney;
begin
  Deposit( My_Account, 100.00 );
  Other_Account := My_Account;           --Copy and
  Withdraw(Other_Account, 100.00, Obtain);--Withdraw 100.00

  Other_Account := My_Account;           --Copy again and
  Withdraw(Other_Account, 100.00, Obtain);--Withdraw 100.00
end Main;
```

If Account in the class Account had been made **limited private**, its specifications would be:

```
package Class_Account is
  type Account is limited private;

  -- Methods (functions and procedures)

private
  type Account is limited record
    Balance_Of : Money := 0.00;      --Amount in account
  end record;
end Class_Account;
```

Note: The record declaration in the private part of the class is also of **limited** type.  
In Ada 83 the use of **limited in**:

**type Account is limited record**  
is not allowed.

## 84 Packages as classes

The more traditional reason for making a type limited is that a copy operation will not produce the expected result for an instance of the type. Chapter 16 describes such a type that is built using dynamic storage.

The table below summarizes the allowable uses of **private** and **limited private** types.

Operation involving	private	limited private
Assignment	√	×
Comparison using = and /= by default	√	×
Parameter passing	√	√ (see note)

*Note:* To pass an object as a parameter, a copy is not necessarily made.

### 6.10 Initializing an object at declaration time

In Ada it is possible to initialize an object when it is declared, although unfortunately there are restrictions to this initialization. Essentially there are two strategies that can be employed. These strategies are:

- Use a discriminant to specify an initial value. The use of discriminants is fully covered in Section 7.4.
- Use an assignment statement to set the object to a specific value.

For example, the following modified class `Account` uses both these approaches to initialize an object on declaration.

```
package Class_Account is
  subtype Money is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;
  type Account( Number: Natural:= 0 ) is private;

  procedure Statement( The:in Account );
  procedure Deposit ( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account; Amount:in Pmoney;
    Get:out Pmoney );
  function Balance ( The:in Account ) return Money;
  procedure New_Number( The: in out Account; N:in Natural );
  function New_Account( N:in Natural;
    Amount:in Pmoney:=0.0 ) return Account;
private
  type Account( Number: Natural:= 0) is record
    Balance_Of : Float := 0.00;
  end record;
end Class_Account;
```

#### 6.10.1 By discriminant

Here a type can be given a discriminant so that a whole family of types may be declared. The discriminant value is held in an instance of the type. Section 7.4 describes in more detail the use of discriminants.

For example, to set `My_Account` with a specific account number the following code is written:

```
with Class_Account, Statement;
use Class_Account;
procedure Main is
  My_Account: Account(10001);
begin
  Deposit( My_Account, 200.00 );
  Statement( My_Account );
  New_Number( My_Account, 10002 );
  Statement( My_Account );
end Main;
```

*Note:* The discriminant value 10001 in the declaration of an instance of `Account`.  
The use of the procedure `Statement` defined above.

which when run, will produce:

```
Mini statement: Account £      10001
The amount on deposit is £200.00

Mini statement: Account £      10002
The amount on deposit is £200.00
```

### 6.10.2 Restrictions

The following restrictions apply, however:

- Only discrete objects or access values may be used as the discriminant value. If an access value is used then the type must be limited.
- To change the discriminant value the whole record structure must be changed.

Thus the implementation of the procedure `New_Number` that allocates a new account number is:

```
procedure New_Number( The: in out Account; N:in Natural ) is
begin
  The := Account'( N, The.Balance_Of );
end New_Number;
```

*Note: The whole record structure needs to be changed to change the discriminant. Chapter 7 discusses **record** initialization in more detail.*

### 6.10.3 By assignment

In this case the object is assigned an initial value when it is declared. For example, the following code sets `My_Account` with an account number and initial balance:

```
with Class_Account, Statement;
use Class_Account;
procedure Main is
  My_Account : Account := New_Account( 10001, 20.0 );
begin
  Statement( My_Account );
end Mai3;
```

which when run, will produce:

```
Mini statement: Account £      10001
The amount on deposit is £20.00
```

## 86 *Packages as classes*

### 6.10.4 Restrictions

The following restrictions apply, however.

- As an assignment is used, the type may not be limited.
- The effect of the assignment statement may have undesirable consequences. For an explanation of these consequences, see Section 17.4.

### 6.11 A personal account manager

One of the applications on a PDA (Personal Digital Assistant) is a PAM (Personal Account Manager). The PAM provides facilities for recording the transactions that take place on the user's bank account. An example of the use of the PAM is shown below:

```
[a] Deposit
[b] Withdraw
[c] Balance
Input selection: a
Amount to deposit : 10.00
```

```
[a] Deposit
[b] Withdraw
[c] Balance
Input selection: b
Amount to withdraw : 4.60
```

```
[a] Deposit
[b] Withdraw
[c] Balance
Input selection: c
Balance is 5.40
```

The program can be constructed using two classes: Account shown in Section 6.3.4 and a new class TUI that will implement the text interface. The responsibilities of the class TUI are:

Method	Responsibility
Menu	Set up the menu that will be displayed to the user. Each menu item is described by a string.
Event	Return the menu item selected by a user of the TUI.
Message	Display a message to the user.
Dialog	Solicit a response from the user.

The Ada specification of the class TUI is:

```
package Class_TUI is
  type Menu_Item is ( M_1, M_2, M_3, M_4, M_Quit );
  type TUI is private;

  procedure Menu( The:in out TUI; M1,M2,M3,M4:in String );
  function Event( The:in TUI ) return Menu_Item;
  procedure Message( The:in TUI; Mes:in String );
  procedure Dialog(The:in TUI; Mes:in String; Res:out Float);
  procedure Dialog(The:in TUI; Mes:in String; Res:out Integer);
private
  -- Not a concern of the client of the class
end Class_TUI;
```

For example, if an instance of the TUI had been declared with:

```
Screen : TUI;
```

then, to setup the menu system:

```
[a] Print
[b] Calculate

Input selection:
```

the following code sequence would be used:

```
Menu( Screen, "Print", "Calculate", "", "" );
```

*Note: Null or empty menu items are not displayed.  
A string may be of any length. However, to store a string the receiving object must be of the correct size. Ada strings are fully discussed in Section 8.8.*

The user's response to this menu is elicited with the function event. The function event returns an enumeration representing the menu item selected. For example, if the user selected option [b] then the code:

```
case Event( Screen ) is
  when M_1 =>                                --Print
  when M_2 =>                                --Calculate
```

associated with label M\_2 would be obeyed.

*Note: The selected menu item is indicated by an enumeration M\_1 for menu item 1, M\_2 for menu item 2, etc.*

A programmer can display a message onto the TUI by using the procedure message which has the text to be output as its second parameter. Likewise, a programmer can initiate a dialog with the user by using the procedure dialog that returns a floating point number. The TUI currently only supports dialogs that solicit a floating point number or integer number.

The fragment of code below illustrates the use of message and dialog interactions in a program which converts miles to kilometres.

```
Message( Screen, "Distance converter" );
Dialog ( Screen, "Enter distance in miles", Miles );
Message( Screen, "Distance in Kilometres is " &
          Float'Image( Miles * 1.6093 ) );
```

*Note: The operator & concatenates two strings into a single string. For example, "Hello" & " " & "world" delivers the single string "Hello world".*

## 88 Packages as classes

In constructing the main program for the personnel account manager, a nested function `float_image` is used to simplify the construction of the program.

```
with Ada.Float_Text_IO, Class_Account, Class_TUI;
use  Ada.Float_Text_IO, Class_Account, Class_TUI;
procedure Main is
  User      : Account;           --The users account
  Screen    : TUI;              --The display screen
  Cash      : Money;            --
  Received  : Money;            --
```

The nested function `Float_Image` converts a floating point number into an Ada string. This function is provided so that the format of the number may be controlled.

```
function Float_Image( F:in Float ) return String is
  Res : String( 1 .. 10 );      --String of 10 characters
begin
  Put( Res, F, 2, 0 );          --2 digits - NO exp
  return Res;
end Float_Image;
```

*Note: The declaration of a string of 10 characters is filled with the character representation for the floating point number `res`.  
The procedure 'Put( `res`, `f`, `aft=>2`, `exp=>0` );' converts a floating point number into a string.*

The main body of the program processes the option selected by the user.

```
begin
loop
  Menu( Screen, "Deposit", "Withdraw", "Balance", "" );
  case Event( Screen ) is
    when M_1 => --Deposit
      Dialog( Screen, "Amount to deposit", Cash );
      if Cash <= 0.0 then
        Message( Screen, "Must be >= 0.00" );
      else
        Deposit( User, Cash );
      end if;
    when M_2 => --Withdraw
      Dialog( Screen, "Amount to withdraw", Cash );
      if Cash <= 0.0 then
        Message( Screen, "Must be >= 0.00" );
      else
        Withdraw( User, Cash, Received );
        if Received <= 0.0 then
          Message( Screen, "Not enough money" );
        end if;
      end if;
    when M_3 => --Balance
      Message( Screen, "Balance is " &
        Float_Image( Balance(User)) );
    when M_Quit => --Exit
      return;
    when others => --Not used
      Message( Screen, "Program error" );
  end case;
end loop;
end Main;
```

## 6.12 Class TUI

The full specification for the class TUI is:

```
package Class_TUI is

  type Menu_Item is ( M_1, M_2, M_3, M_4, M_Quit );
  type TUI is private;

  procedure Menu( The:in out TUI; M1,M2,M3,M4:in String );
  function Event( The:in TUI ) return Menu_Item;
  procedure Message( The:in TUI; Mes:in String );
  procedure Dialog(The:in TUI; Mes:in String; Res:out Float);
  procedure Dialog(The:in TUI; Mes:in String; Res:out Integer);
private
  type TUI is record
    Selection : Menu_Item := M_Quit;
  end record;
end Class_TUI;
```

In the implementation of the class TUI the most complex method is menu. This method is implemented as a procedure that writes out the menu for the TUI and reads the user's response. It will only complete when a valid response has been received from the user. In the implementation of the procedure the technique of procedural decomposition is used to simplify the code.

In procedural decomposition, a large body of code is split into several procedures or functions. This helps to reduce complexity making construction and maintenance easier.

## 90 Packages as classes

```
with Ada.Text_Io, Ada.Float_Text_Io, Ada.Integer_Text_Io;
use Ada.Text_Io, Ada.Float_Text_Io, Ada.Integer_Text_Io;
package body Class_TUI is
  procedure Menu( The:in out TUI; M1,M2,M3,M4:in String ) is

    Selection      : Character;
    Valid_Response : Boolean := False;
```

As a user may inadvertently select a null menu item, the procedure Set\_Response is used to disallow such an action.

```
procedure Set_Response(Choice:in Menu_Item; Mes:in String) is
begin
  if Mes /= " " then          --Allowable choice
    The.Selection := Choice; Valid_Response := True;
  end if;
end Set_Response;
```

The procedure Display\_Menu\_Item displays onto the TUI only non null menu items.

```
procedure Display_Menu_Item(Prompt, Name:in String) is
begin
  if Name/=" " then
    Put(Prompt & Name); New_Line(2);
  end if;
end Display_Menu_Item;
```

The main body of the procedure displays the menu on the screen and receives the selected menu choice from the user. If an invalid response is received the menu is re-displayed and the user is asked again to select a menu item.

```
begin  -- Menu
  while not Valid_Response loop
    Display_Menu_Item( "[a] ", M1 );
    Display_Menu_Item( "[b] ", M2 );
    Display_Menu_Item( "[c] ", M3 );
    Display_Menu_Item( "[d] ", M4 );
    Put( "Input selection: "); Get( Selection ); Skip_Line;
    case Selection is
      when 'a' | 'A' => Set_Response( M_1, M1 );
      when 'b' | 'B' => Set_Response( M_2, M2 );
      when 'c' | 'C' => Set_Response( M_3, M3 );
      when 'd' | 'D' => Set_Response( M_4, M4 );
      when 'e' | 'E' => Set_Response( M_Quit, "Quit" );
      when others    => Valid_Response := False;
    end case;
    if not Valid_Response then
      Message( The, "Invalid response" );
    end if;
  end loop;
end Menu;
```

The function Event returns the user's selection.

```
function Event( The:in TUI ) return Menu_Item is
begin
  return The.Selection;
end;
```



The procedure Message writes a string onto the screen.

```
procedure Message( The:in TUI; Mes:in String ) is
begin
  New_Line; Put( Mes ); New_Line;
end Message;
```

The procedure Dialog solicits a response from the user.

```
procedure Dialog(The:in TUI; Mes:in String; Res:out Float) is
begin
  New_Line(1); Put( Mes & " : " );
  Get( Res ); Skip_Line;
end Dialog;

procedure Dialog(The:in TUI; Mes:in String; Res:out Integer) is
begin
  New_Line(1); Put( Mes & " : " );
  Get( Res ); Skip_Line;
end Dialog;

end Class_TUI;
```

*Note: In this case the response must be a floating point number or an integer number. Other overloaded procedures can be provided for different forms of dialog.*

## 6.13 Self-assessment

- Why should a program be split into many packages?
- What is a class?
- How do you declare an instance of a class in Ada?
- What is the difference between the declaration of a class and the declaration of an instance of that class?
- When an instance of a class is declared, what happens?
- What is contained in a class?
- How can a user of a class request the execution of a method/function in that class?
- What are the advantages of holding data and the code that operates on the data together?
- Should a function in a class be private? Explain your answer.
- Should a data item in a class be public? Explain your answer.
- How should an implementor of a class allow access to instance attributes contained in an object?

## 6.14 Exercises

Construct the following classes:

- *Account\_with\_overdraft*

Construct a class which represents an account on which a customer is allowed to go overdrawn. You should restrict the amount the customer is allowed to go overdrawn. The methods of this class are:

Method	Responsibility
Balance	Deliver the balance of the account.
Deposit	Deposit money into the account
Set_Overdraft_Limit	Set the overdraft limit.
Statement	Print a statement of the current balance of the account.
Withdraw	Withdraw money from the account.

- *Cinema Performance Attendance*

A class Performance, an instance of which represents the seats at a particular showing of a film, has the following methods:

Method	Responsibility
Book_seats	Book n seats at the performance.
Cancel	Unbook n seats.
Sales	Return the value of the seats sold at this performance.
Seats_Free	Return the number of seats that are still unsold.

Thus on an instance of Performance the following actions can be performed:

- - Book a number of seats
  - Find out the number of unsold seats at the performance
  - Cancel the booking for n seats.
  - Return the value of the seats sold at this performance.
- *Library Book*  
A class to represent a book in a library, such that the following operations can be processed:
  - Loan the book.
  - Mark the book as being reserved. Only one outstanding reservation is allowed on a book.
  - Ask if a book can be loaned. A book can only be loaned if it is not already on loan or is not reserved.
  - Return the book.

Construct the following program:

- *Cinema*

A program to deal with the day-to-day administration of bookings for a cinema for a single day. Each day there are three separate performances: an early afternoon performance at 1pm, an early evening performance at 5pm and the main performance at 8.30pm.

The program should be able to handle the booking of cinema seats for any of these three performances and supply details about the remaining seats for a particular performance.

Hints:

- Use the class TUI.
- Use three instances of the class Performance.
- Use a case statement.
- Use a procedure to process transactions on a particular performance.

# 7 Data structures

This chapter explores the use of data structures. A data structure is used to hold a collection of related data items. This is implemented in Ada with the construct **record**. However, data structures are a low-level construct and in many instances, the use of a class will enable better quality code to be produced. As was seen in the previous chapter a **record** is used to hold the hidden instance attributes in a class.

## 7.1 The record structure

In the construction of a program it is convenient to group like data items together. For example, details about a person may consist of:

- The person's name.
- Their height in centimetres.
- Their sex.

The record structure can be used to group these three distinct data items together into a new type called **Person**. For example, the above description of a **Person** can be defined as follows:

```
Max_Chars : constant := 10;
type Gender is ( Female, Male );
type Height_Cm is range 0 .. 300;
type Person is record
  Name      : String( 1 .. Max_Chars );  --Name as a String
  Height    : Height_Cm := 0;           --Height in cm.
  Sex       : Gender;                  --Gender of person
end record;
```

Then an instance of a **Person** can be declared using the declaration:

```
Mike      : Person;
```

This is similar to a class declaration as seen in the previous chapter. However, all the members of the data structure are visible to a user of the object **Mike**.

## 7.2 Operations on a data structure

The **.** notation is used to access individual members of a data structure. For example, to set up a description of the person **mike**, the following code can be used.

```
Mike.Name      := "Mike";
Mike.Height    := 183;
Mike.Sex       := Male;
```

This initialization can be more elegantly expressed using a record aggregate which is then assigned to the object mike.

```
Mike      := (Name=> "Mike      ", Height=> 183, Sex=> Male);
```

*Note: The construct:  
(name => "Mike ", height => 183, sex => Male )  
is a record aggregate.*

The record aggregate can also be defined using the absolute position of the arguments or a mixture of absolute and named arguments. For example, the following three assignments are all equal in effect.

```
Corinna := (      "Corinna  ",      171,      Female);  
Corinna := (Name=> "Corinna  ", Sex=> Female, Height=> 171);  
Corinna := (      "Corinna  ", Sex=> Female, Height=> 171);
```

*Note: A record aggregate must have all its components specified even if some components have default values. Once a named parameter in an aggregate has been used, all parameters to the right must also be named.*

*If there is only one member of the record aggregate then it must still be enclosed in brackets.*

A data structure may be compared for equality or assigned. For example, using the declarations:

```
Corinna, Mike, Miranda : Person;  
Taller                  : Person;
```

the following code can be written:

```
Mike      := (Name=>"Mike      ", Height=>183, Sex=>Male);  
Corinna:= (Name=>"Corinna  ", Height=>171, Sex=>Female);  
Miranda:= (Name=>"Miranda  ", Height=>74,  Sex=>Female);  
  
Taller    := Mike;  
  
if mike = Taller then  
    Put("Mike taller"); New_Line;  
end if;  
if Mike /= Taller and Corinna /= Taller then  
    Put("Miranda taller"); New_Line;  
end if;
```

## 7.2.1 Other operations allowed on data structures

Chapter 12 describes how new meanings for the inbuilt operators in Ada can be defined. Using these techniques to define an additional meaning for > between instances of a Person would allow the following to be written:

```
if Mike > Corinna then  
    Put("Mike taller"); New_Line;  
else  
    Put("Corinna taller"); New_Line;  
end if;
```

## 96 Data structures

### 7.3 Nested record structures

A data structure declaration may be nested as in the following record declaration for a bus:

```
type Bus is record
  Driver : Person;      --Bus driver
  Seats  : Positive;    --Number of seats on bus
end record;

London : Bus;
```

Individual components are accessed using the . notation as follows:

```
London.Driver.Name := "Jane";
London.Driver.Sex  := Female;
London.Driver.Height := 168;
London.Seats       := 46;
```

*Note: The repeated . is used to access first the Driver and then the data members Name, Sex and Height.*

However, a record aggregate may also be used as shown below:

```
London := ( ("Jane", 168, Female), 46 );
```

### 7.4 Discriminants to records

A record type may have a parameter (discriminant) whose value may be an instance of a discrete type or access type. Access types are fully described in Chapter 15. For example, the data structure for a person can be defined with a discriminant which specifies the number of characters for the String. This new definition for a Person is shown below:

```
type Gender is ( Female, Male );
type Height_Cm is range 0 .. 300;
subtype Str_Range is Natural range 0 .. 20;
type Person( Chs: Str_Range ) is record
  Name   : String( 1 .. Chs );      --Name length
  Height : Height_Cm := 0;          --As String
  Sex    : Gender;                  --Height in cm.
end record;                        --Gender
```

*Note: The discriminant is a component of the record.*

In the declaration of an instance of a Person the length of the String used for a person's name is specified after the type name as follows:

```
Mike    : Person(4);    --Constrained
Corinna : Person(7);    --Constrained
Younger : Person(10);   --Constrained
```

Then an assignment to an instance of Person is:

```
Mike    := (4, Name=>"Mike"    , Height=>183, Sex=>Male);
Corinna:= (7, Name=>"Corinna", Height=>171, Sex=>Female);
```

*Note: The value of the discriminant must be specified in the record aggregate.*

However, Mike, Corinna and Younger are not of the same type so the assignment:

```
Younger := Corinna;  -- Fail at run-time
```

will fail at run-time as the discriminants of the record are not identical. The object Younger contains a String of length of 10 whilst the object Corinna contains a String of length 7.

## 7.5 Default values to a discriminant

A discriminant to a type may have a default value. If a value is not specified with the declaration of a discriminated type then it is an unconstrained discriminated type. An instance of an unconstrained discriminated type may be assigned or compared with other unconstrained discriminants of the same type name.

For example, if the data structure Person is now defined as:

```
type    Gender    is ( Female, Male );
type    Height_Cm is range 0 .. 300;
subtype Str_Range is Natural range 0 .. 20;
type    Person( Chs:Str_Range := 0 ) is record --Length of name
  Name   : String( 1 .. Chs );                --Name as String
  Height : Height_Cm    := 0;                  --Height in cm.
  Sex    : Gender;                          --Gender
end record;
```

then the following code can be written:

```
declare
  Mike    : Person;      --Unconstrained
  Corinna : Person;      --Unconstrained
  Younger : Person;      --Unconstrained
begin
  Mike    := (4, Name=>"Mike"    , Height=>183, Sex=>Male);
  Corinna:= (7, Name=>"Corinna", Height=>171, Sex=>Female);
  Younger := Corinna;

  if Corinna = Younger then
    Put( "Corinna is younger" ); New_Line;
  end if;
end;
```

*Note: It would still be an error to write:*

```
Corinna:=( 10, name=>"Corinna", height=>171, sex=>Female );
as the length of "Corinna" is not 10 characters.
```

## 98 Data structures

### 7.5.1 Constrained vs. unconstrained discriminants

Using the last definition of type `Person`:

Declaration	The object <code>Mike</code> is	Comment
<code>Mike: Person;</code>	Unconstrained	The variable <code>Mike</code> may be compared with or assigned any other instance of <code>Person</code> .
<code>Mike: Person(4);</code>	Constrained	May only be assigned or compared with another <code>Person(4)</code> .

### 7.5.2 Restrictions on a discriminant

A discriminant must be a discrete type or access type. If it is an access type then the record must be limited. This unfortunately means that a `Float` cannot be used as a discriminant to a record.

## 7.6 Variant records

There will be occasions when a data structure contains data items that are mutually exclusive. For example, in a description of a person who may be a `Lecturer` or a `Student` the data members are:

Data member	Belongs to	Description
Name	Both a <code>Lecturer</code> and a <code>Student</code>	The name of the person.
Class_Size	<code>Lecturer</code>	The size of the group that the lecturer teaches.
Full_Time Grade	<code>Student</code>	Whether or not the student is full-time or part-time. The mark out of 100 that the student gains at the end of the course.

In this example the storage for `Class_Size` can overlay all or part of the storage for `Full_Time` and `Grade` as the components:

- `Class_Size`
- `Full_Time` and `Grade`

of the data structure will not be used simultaneously.

This can be visualized as:

role -> <code>Student</code>	Name	Full_Time	grade
role -> <code>Lecturer</code>	Name	Class_Size	



In Ada a variant record allows two or more data items to occupy the same physical storage. This will result in a lower memory usage for the data in a program. However, access to the variant components must be carefully controlled to prevent information being stored or extracted as the wrong type. For example, if the record represents a lecturer then it should not be possible to access the component `Full_Time` as this is only present when the record represents a student. Ada with its strict typing will prevent such occurrences.

The data structure to represent either a lecturer or a student is defined as:

```

type Occupation is (Lecturer, Student);
type Mark is range 0 .. 100;
subtype Str_Range is Natural range 0 .. 20;
type Person( Chs : Str_Range :=0;
  Role: Occupation:=Student ) is record
  Name : String( 1 .. Chs );      --Name as string
  case Role is                  --Variant record
    when Lecturer =>            -- Remember storage overlaid
      Class_Size: Positive;      --Size of taught class
    when Student =>
      Grade : Mark;              --Mark for course
      Full_Time : Boolean := True; --Attendance mode
  end case;
end record;

```

*Note: If one discriminant item is given a default value than all discriminant items that follow must also be given a default value.*

Using the above declaration of `Person` allows the following assignments to be made:

```

declare
  Mike : Person;                --Unconstrained
  Clive: Person;                --Unconstrained
  Brian: Person(5,Student);     --Constrained
begin
  Mike :=(4, Lecturer, Name=>"Mike", Class_Size=>36);
  Clive:=(5, Student, Name=>"Clive", Grade=>70,Full_Time=>True);
  -- insert --
end;

```

In Ada, this process is safe as the compiler will check and disallow access to a variant part which is not specified by the discriminant. For example, if the following statements were inserted at the point `-- insert --` above, they would be flagged as invalid at either compile-time or run-time.

Invalid statement	Reason
<code>Clive.Role:= STUDENT</code>	Not allowed to change just a discriminant as this would allow data to be modified/extracted as the wrong type. Detectable at compile-time.
<code>Mike.Grade:= 0</code>	Access to a component of the data structure which is not active. Mike is a lecturer and hence has no grade score. Detected at run-time.
<code>Brian := (5, Lecturer, Name=&gt;"Brian", Class_Size=&gt;36);</code>	The object Brian is constrained to be a student. Detectable at compile-time.

## 7.7 Limited records

If a record is limited then the compiler will prevent assignment and comparison of an instance of the record. For example, with the declaration:

```
type Person is limited record
  Name    : String( 1 .. Max_Chars ); --Name as a String
  Height  : Height_Cm := 0;           --Height in cm.
  Sex     : Gender;                  --Gender of person
end record;

Mike     : Person;
Corinna  : Person;
```

The following code will fail to compile:

```
Mike     := Corinna;           --Fails to compile as record is limited

if Corinna = Mike then        --Fails to compile as record is limited
  Put("This is strange"); New_Line;
end if;
```

## 7.8 Data structure vs. class

The table below summarizes the differences between a class and a data structure.

	Data structure	Class
A user of the construct can directly access and change the internal structure.	√	×
Can be used where a normal type can be used.	√	√
Code to manipulate the data encapsulated with the construct.	×	√
Representation of the data items is hidden from the user.	×	√

Thus to provide data hiding, a class must be used.

## 7.9 Self-assessment

- Explain how you can access an individual component of a record.
- What are the major differences between a class and an Ada **record**?
- Is the use of variant records safe in Ada? Explain your answer.
- What is the difference between an unconstrained and a constrained record declaration?

## 7.10 Exercises

Construct the following:

- An Ada record to describe the computer you are using. This should include for example: the amount of main memory, the amount of cache memory, the amount of disk space.

You may need a coding scheme for the size of memory components, as in the case of disk space this can be a very large number.

- Generalize this record so that it can hold details about many different types of computer. Include a variant part to allow for the following different types:
  - A computer used for word processing, with no network devices or multimedia components.
  - A multimedia computer with sound and video capability.
  - A workstation with a network and file server connections.

# 8 Arrays

This chapter introduces arrays that implement a collection facility for objects. With this facility, objects are stored and retrieved using an index of a discrete type.

## 8.1 Arrays as container objects

An array is a collection of objects that can be accessed using an instance of a discrete type. For example, the number of computer terminals in five rooms could be described with the following declaration:

```
Computers_In_Room : array ( 1 .. 5 ) of Natural;
```

*Note:* `Computers_In_Room` is a collection of `Natural` numbers and the integer numbers 1 through 5 are used to select a particular object in this collection.

*The compiler will check that the subscript is valid. If it cannot be checked directly, code should be inserted which will perform the check at run-time. The exception `Constraint_error` is raised if the index is out of bounds.*

The number of terminals in each room can be recorded in the collection `Computers_In_Room` by using an array index to select a particular computer room. For example, to set the number of computers in room 1 to 20, 2 to 30, 3 to 25, 4 to 10 and 5 to 15, the following code can be used:

```
Computers_In_Room(1) := 20;  
Computers_In_Room(2) := 30;  
Computers_In_Room(3) := 25;  
Computers_In_Room(4) := 10;  
Computers_In_Room(5) := 15;
```

This can be visualized diagrammatically as shown in Figure 8.1.

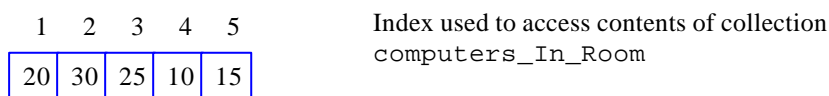


Figure 8.1 Diagrammatic representation of an array.

Once information about the number of computers in each room has been set up, it can be printed with the following code:

```
for I in 1 .. 5 loop  
  Put("Computers in room "); Put( I, Width=>1 ); Put(" is ");  
  Put( Computers_In_Room(I), Width=>2 ); New_Line;  
end loop;
```

which when combined with appropriate declarations and run, would produce the following results:

```
Computers in room 1 is 20
Computers in room 2 is 30
Computers in room 3 is 25
Computers in room 4 is 10
Computers in room 5 is 15
```

In the example above, any Integer object can be used as an index to the array. This freedom may lead to program errors when a value other than the intended subscript is used. Unfortunately such an error would not be detected until run-time. To allow Ada to perform strict type checking so that such an error may be caught at compile-time, a separate type for the bounds of the array can be defined. This is achieved by defining a range type that is then used to describe the bounds of the array. For example, the previous object Computers\_In\_Room could have been defined as:

```
type Rooms_Index is range 1 .. 5;
type Rooms_Array is array ( Rooms_Index ) of Natural;

Computers_In_Room : Rooms_Array;
```

In the declaration the following types and subtypes have been defined:

Type / Subtype	Description
Rooms_index	A type used to define an object that is used to index the array.
Rooms_array	A type representing the array.

The range of elements in the array is defined by Rooms\_Index or Rooms\_Array'Range. In Ada the attribute 'Range gives the bounds of an array object or array type. For example, using the above declarations, the attribute 'Range would have the following values.

Attribute	Description	Value
Rooms_array'Range	Equivalent to the range Rooms_array'First .. Rooms_array'Last	1 .. 5
Computers_In_Room'Range	As above	

*Note: The type of the elements of the range will of course be Rooms\_Index.*

Using the above types and subtypes, the **for** loop which prints the number of computers in each room would now become:

```
Computers_In_Room : Rooms_array;

-- Set up contents of Computers_In_Room

for I in Computers_In_Room'Range loop
  Put("Computers in room "); Put( Integer(I), Width=>1 );
  Put(" is "); Put( Computers_In_Room(I), Width=>2 ); New_Line;
end loop;
```

*Note: As the index i to the for loop is now of type Rooms\_range it must be converted to an Integer before it can be output using the procedure Put. Mechanisms to output objects of different discrete types will be explored later in Chapter 18.*

## 104 Arrays

A compile-time error message will be generated if the programmer incorrectly uses the above mechanism to index the array, for example, when using an object which is neither of type `Room_index` nor a subtype of `Room_index`.

### 8.2 Attributes of an array

As arrays in Ada are self-describing, various attributes can be extracted from an instance of an array. For example, with the following declarations for the array `Marks`:

```
type    Marks_Index is new Character range 'a' .. 'f';
type    Marks_Array is array (Marks_Index ) of Natural;
Marks : Marks_Array;
```

a number of attributes can be extracted:

Attribute	Description	Value
<code>Marks'Length</code>	A Universal integer representing the number of elements in the one dimensional array.	6
<code>Marks'First</code>	The first subscript of the array which is of type <code>Marks_Range</code>	'a'
<code>Marks'Last</code>	The last subscript of the array which is of type <code>Marks_Range</code>	'f'
<code>Marks'Range</code>	Equivalent to <code>Marks'First .. Marks'Last</code>	'a'..'f'

*Note: This would also be true for `Marks_Array'Length` etc.*

A fuller description of the attributes that can be extracted from an object or type are given in Section B.2, Appendix B.

### 8.3 A histogram

A program to print a histogram representing the frequency of individual letters occurring in a piece of text can be developed by first implementing a class that performs the following operations on an instance of a `Histogram`.

Method	Responsibility
<code>Add_To</code>	Add a character to the histogram, recording the updated total number of characters.
<code>Put</code>	Write a histogram to the output source representing the currently gathered data.
<code>Reset</code>	Clear any previously gathered data, setting various internal objects to an initial state.

Using the class `Histogram` that has been implemented in the package `Class_Histogram`, code can be written which will produce a histogram of the frequency of characters taken from the standard input:

```

with Ada.Text_IO, Class_Histogram;
use  Ada.Text_IO, Class_Histogram;
procedure Main is
  Ch:Character;           --Current character
  Text_Histogram: Histogram; --Histogram object
begin
  Reset(Text_Histogram);  --Reset to empty
  while not End_Of_File loop --For each line
    while not End_Of_Line loop --For each character
      Get(Ch);             --Get current character
      Add_To( Text_Histogram, Ch ); --Add to histogram
    end loop;
    Skip_Line;             --Next line
  end loop;
  Put( Text_Histogram );   --Print histogram
end Main;

```

The class histogram is defined by the following package specification:

```

package Class_Histogram is
  type Histogram is private;
  Def_Height : constant Positive := 14;
  procedure Reset( The:in out Histogram );
  procedure Add_To( The:in out Histogram; A_Ch:in Character );
  procedure Put(The:in Histogram; Height:in Positive:=Def_Height);
private
  type Alphabet_Index is new Character range 'A' .. 'Z';
  type Alphabet_Array is array (Alphabet_Index) of Natural;
  type Histogram is record
    Number_Of : Alphabet_Array := ( others => 0 );
  end record;
end Class_Histogram;

```

The histogram is calculated using the individual letter frequencies that are stored in an array of Naturals indexed by the upper case letters 'A' .. 'Z'. The implementation is simplified by allowing each letter to index directly the frequency count for the letter.

In the implementation of the class histogram shown below, the procedure `reset` is used to set the individual frequency counts for each letter to 0.

```

with Ada.Text_IO, Ada.Float_Text_IO, Ada.Characters.Handling;
use  Ada.Text_IO, Ada.Float_Text_IO, Ada.Characters.Handling;
package body Class_Histogram is

  procedure Reset(The:in out Histogram) is
  begin
    The.Number_Of := ( others => 0 ); --Reset counts to 0
  end Reset;

```

The procedure `Add_To` uses the functions `Is_Lower`, `To_Upper` and `Is_Upper` which are contained in the package `Ada.Characters.Handling`. A full description of these functions can be found in Section C.7, Appendix C.

```

procedure Add_To(The:in out Histogram; A_Ch:in Character) is
  Ch : Character;
begin
  Ch := A_Ch;                                --As write to ch
  if Is_Lower(Ch) then                        --Convert to upper case
    Ch := To_Upper( Ch );
  end if;

  if Is_Upper( Ch ) then                      --so record
    declare
      C : Alphabet_Index := Alphabet_Index(Ch);
    begin
      The.Number_Of(C) := The.Number_Of(C) + 1;
    end;
  end if;

end Add_To;

```

The histogram is displayed as a bar graph corresponding to the accuracy of the output device, which in this case is an ANSI terminal. The size of the histogram is set by the defaulted parameter Height.

```

procedure Put(The:in Histogram;
              Height:in Positive:=Def_Height) is
  Frequency   : Alphabet_Array;    --Copy to process
  Max_Height  : Natural := 0;      --Observed max
begin
  Frequency := The.Number_Of;      --Copy data (Array)
  for Ch in Alphabet_Array'Range loop --Find max frequency
    if Frequency(Ch) > Max_Height then
      Max_Height:= Frequency(Ch);
    end if;
  end loop;

  if Max_Height > 0 then
    for Ch in Alphabet_Array'Range loop --Scale to max height
      Frequency(Ch):=(Frequency(Ch)*Height)/(Max_Height);
    end loop;
  end if;

  for Row in reverse 1 .. Height loop --Each line
    Put( " | " );                      --start of line
    for Ch in Alphabet_Array'Range loop
      if Frequency(Ch) >= Row then
        Put( '*' );                    --bar of hist >= col
      else
        Put( ' ' );                    --bar of hist < col
      end if;
    end loop;
    Put( " | " ); New_Line;            --end of line
  end loop;

```

```

  Put( " +-----+" ); New_Line;
  Put( "  ABCDEFGHIJKLMNOPQRSTUVWXYZ " ); New_Line;
  Put( " * = (approx) " );
  Put( Float(Max_Height) / Float(Height), Aft=>2, Exp=>0 );
  Put( " characters " ); New_Line;
end Put;
end Class_Histogram;

```

*Note:* By implementing the printing of the histogram as part of the package `Class_Histogram` this severely limits the cases when this code can be re-used.

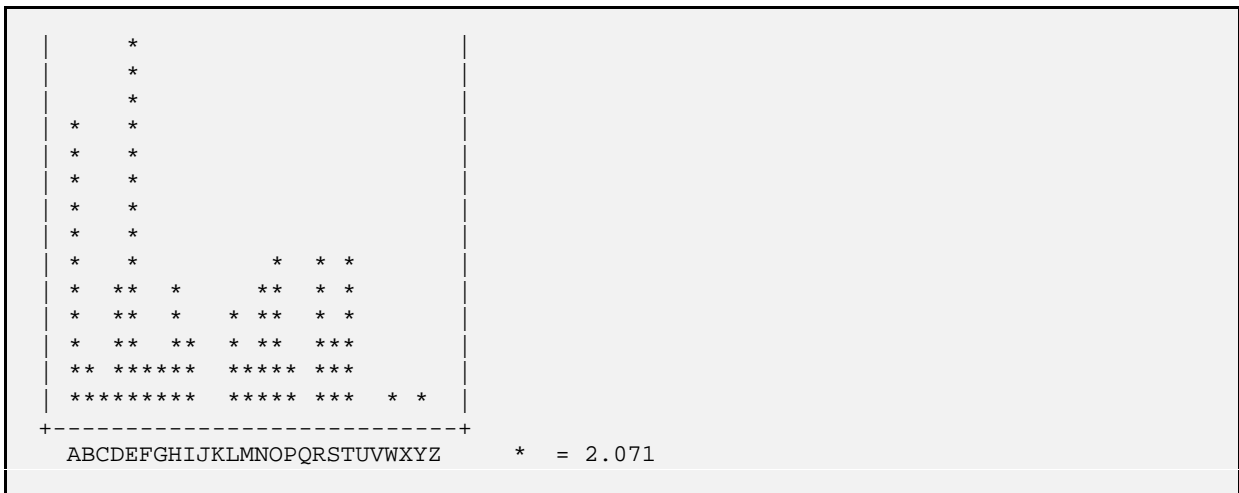


### 8.3.1 Putting it all together

When run with the following data:

```
Ada is a language developed for the American Department
of Defense.
Ada is named after the first programmer Ada (Byron) Lovelace
who helped Charles Babbage with his work on the analytical engine.
She was the daughter of the poet Lord Byron.
```

the program would produce the following output:



*Note: The exact number of characters shown for each \* in the bar graph is guaranteed to be accurate only for the most frequently occurring character.*

## 8.4 The game of noughts and crosses

The children's game of noughts and crosses is played on a three-by-three grid of squares. Players either play X or O. Each player takes it in turn to add their mark to an unoccupied square. The game is won when a player has three of their marks in a row either diagonally, horizontally or vertically. If no unoccupied square remains, the game is a draw (Figure 8.2).

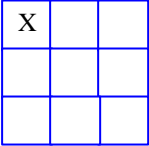
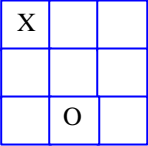
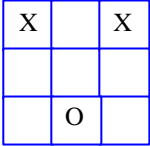
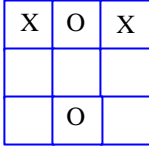
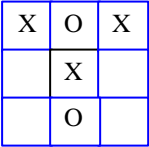
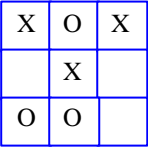
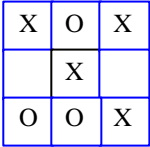
X's first move	O's first move	X's second move	O's second move
			
X's third move	O's third move	X's fourth move	As can be seen to go first is a clear advantage.
			

Figure 8.2 A game of noughts and crosses

A program to display the current state of a game of noughts and crosses between two contestants is developed with the aid of a class `Board`. The operations, `Add`, `Valid`, `State`, `Cell`, and `Reset`, can be performed on an instance of `Board`.

The responsibilities of these methods is as follows:

Method	Responsibility
Add	Add the player's mark to the board. The player's move is specified by a number in the range 1 to 9 and their mark by a character.
Valid	Return true if the presented move is valid. The method checks that the move is in the range 1 to 9 and that the specified cell is not occupied.
State	Returns the state of the current game. Possible states are: <code>Win</code> , <code>Playable</code> , and <code>Draw</code> .
Cell	Returns the contents of a cell on the board. This method is included so that the code that displays the board can be separated from the code that manipulates the board.
Reset	Reset the board back to an initial state.

### 8.4.1 The class Board

The specification for the class Board is defined by the package Class\_board as follows:

```
package Class_Board is

  type Board      is private;
  type Game_State is ( Win, Playable, Draw );

  procedure Add( The:in out Board; Pos:in Integer;
                 Piece:in Character );
  function Valid( The:in Board; Pos:in Integer ) return Boolean;
  function State( The:in Board ) return Game_State;
  function Cell( The:in Board; Pos:in Integer )
                 return Character;
  procedure Reset( The:in out Board );
  -- Not a concern of the client
end Class_Board;
```

### 8.4.2 Implementation of the game

Using the above package specification, the following code will facilitate the playing the game of noughts and crosses between two human players. The procedure Display will display the state of the board onto the user's terminal. By factoring out the code to display the board, from the class Board the class can be re-used in other programs that may use a different form of display, for example a GUI (Graphical User Interface).

```
with Class_Board, Ada.Text_IO, Ada.Integer_Text_IO, Display;
use Class_Board, Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Player : Character;           --Either 'X' or 'O'
  Game    : Board;             --An instance of Class Board
  Move    : Integer;           --Move from user
begin
  Player := 'X';               --Set player
  while State( Game ) = Playable loop --While playable
    Put( Player & " enter move (1-9) : "); -- move
    Get( Move ); Skip_Line;      -- Get move
    if Valid( Game, Move ) then --Valid
      Add( Game, Move, Player ); -- Add to board
      Display( Game );          -- Display board
      case State( Game ) is
        --Game is
        when Win =>
          Put( Player & " wins");
        when Playable =>
          case Player is --Next player
            when 'X' => Player := 'O'; -- 'X' => 'O'
            when 'O' => Player := 'X'; -- 'O' => 'X'
            when others => null; --
          end case;
        when Draw =>
          Put( "It's a draw ");
        end case;
      New_Line;
    else
      Put( "Move invalid"); New_Line; --for board
    end if;
  end loop;
  New_Line(2);
end Main;
```

## 110 Arrays

*Note: That the **case** statement which effects the change between the player's mark has a **when others** clause. This is required as in theory a player can take any character value. The code for this 'impossible' eventuality is the **null** statement.*

The character object `player` holds a representation of the current player's mark, in this case either the character 'X' or 'O'. The object `player` is initially set to 'X', and after each player's move is changed to the other player's mark.

### 8.4.3 Displaying the Board

The procedure `Display` uses the method `Cell` in class `Board` to help display a textual representation of the board on the user's terminal. The procedure `Display` is implemented as follows:

```
with Class_Board, Ada.Text_IO;
use Class_Board, Ada.Text_IO;
procedure Display( The:in Board ) is
begin
  for I in 1 .. 9 loop
    Put( Cell( The, I ) );
    case I is
      when 3 | 6 => --after printing counter
                    -- print Row Separator
                    New_Line; Put( "-----" ); --
                    New_Line;
      when 9 => -- print new line
                    New_Line;
      when 1 | 2 | 4 | 5 | 7 | 8 => -- print Col separator
                    Put( " | " );
    end case;
  end loop;
end Display;
```

The procedure `Display` prints the board on to the player's terminal. The strategy for printing the board is to print each cell followed by a character sequence appropriate for its position on the board. The text to be printed after each square of the array `sqr_s` has been printed is as follows:

Printed board showing array index of cell in array <code>sqr_s</code>	After printing cell:	Text to be printed (Using <code>Ada.Text_IO</code> )
1   2   3 ----- 4   5   6 ----- 7   8   9	1,2,4,5,7 and 8  3 and 6  9	<code>Put( "   " );</code>  <code>New_Line;</code> <code>Put( "-----" );</code> <code>New_Line;</code>  <code>New_Line;</code>

### 8.4.4 The class Board

The full specification of the class Board is as follows:

```
package Class_Board is

  type Board      is private;
  type Game_State is ( Win, Playable, Draw );

  procedure Add( The:in out Board; Pos:in Integer;
                 Piece:in Character );
  function Valid( The:in Board; Pos:in Integer ) return Boolean;
  function State( The:in Board ) return Game_State;
  function Cell( The:in Board; Pos:in Integer )
                return Character;
  procedure Reset( The:in out Board );
private
  subtype Board_Index is Integer range 1 .. 9;
  type Board_Array is array( Board_Index ) of Character;
  type Board is record
    Sqrns : Board_Array := ( others => ' ' );  --Initialize
    Moves : Natural      := 0;
  end record;
end Class_Board;
```

The noughts and crosses board is represented by a single dimensional array of nine characters. The board is initialized to all spaces with the assignment `Board_grid := ( others => ' ' )`. This style of initialization is explained in Section 8.6 *Initializing an array*. In defining the noughts and crosses board the following type and subtype are used:

Type / Subtype	Description
Board_Index	A subtype used to describe an index object used to access an element of the noughts and crosses board. By making Board_Index a subtype of Integer, Integers may be used as an index of the array.
Board_Array	A type used to describe a noughts and crosses board.

The implementation of the class Board is defined in the body of the package Class\_board as follows:

```
package body Class_Board is
```

The procedure add adds a counter either the character 'X' or 'O' to the board.

```
procedure Add( The:in out Board; Pos:in Integer;
               Piece:in Character ) is
begin
  The.Sqrns( Pos ) := Piece;
  The.Moves := The.Moves + 1;
end Add;
```

## 112 Arrays

The function `Valid` returns **true** if the square selected is not occupied by a previously played counter.

```
function Valid(The:in Board; Pos:in Integer) return Boolean is
begin
    return Pos in Board_Array'Range and then
        The.Sqrs( Pos ) = ' ';
end Valid;
```

*Note: The use of **and then** so that the check on the board is only made if the position is valid.*

The function `Cell` returns the contents of a cell on the noughts and crosses board. This method is used to interrogate the state of the board, without having to know how the state is stored. Using this method, printing of the state of the board can be separated from the code that manipulates the board.

```
function Cell(The:in Board; Pos:in Integer) return Character is
begin
    return The.Sqrs( Pos );
end Cell;
```

The procedure `Reset` sets the state of the board back to its initial state.

```
procedure Reset( The:in out Board ) is
begin
    The.Sqrs := ( others => ' ');    --All spaces
    The.Moves := 0;                 --No of moves
end Reset;
```

The Procedure `State` returns the current state of play as represented by the board. A two-dimensional array `All_Win_Lines` holds the co-ordinates of the eight possible win lines. The co-ordinates in this array are used to find any line that contains three cells either containing all X's or O's.

```
function State( The:in Board ) return Game_State is
    subtype Position is Integer range 1 .. 9;
    type Win_Line is array( 1 .. 3 ) of Position;
    type All_Win_Lines is range 1 .. 8;
    Cells: constant array ( All_Win_Lines ) of Win_Line :=
        ( (1,2,3), (4,5,6), (7,8,9), (1,4,7),
          (2,5,8), (3,6,9), (1,5,9), (3,5,7) ); --All win lines
    First : Character;
begin
    for Pw1 in All_Win_Lines loop                --All Pos Win Lines
        First := The.Sqrs( Cells(Pw1)(1) ); --First cell in line
        if First /= ' ' then                    -- Looks promising
            if First = The.Sqrs(Cells(Pw1)(2)) and then
                First = The.Sqrs(Cells(Pw1)(3)) then
                return Win;                      -- Found a win
            end if;
        end if;
    end loop;
    if The.Moves >= 9 then                      --Check for draw
        return Draw;                            -- Board full so draw
    else
        return Playable;                       -- Still playable
    end if;
end State;

end Class_Board;
```

### 8.4.5 Putting it all together

When compiled and run, a possible interaction between two players could be as follows:

X's first move	O's first move	X's second move	O's second move
<pre> x       -----       -----       </pre>	<pre> x       -----       -----   o    </pre>	<pre> x      x -----       -----   o    </pre>	<pre> x   o   x -----       -----   o    </pre>
X's third move	O's third move	X's fourth move	
<pre> x   o   x -----   x    -----   o    </pre>	<pre> x   o   x -----   x    ----- o   o    </pre>	<pre> x   o   x -----   x    ----- o   o   x </pre>	As can be seen to go first is a clear advantage.

## 8.5 Multidimensional arrays

Arrays can have any number of dimensions. For example, the noughts and crosses board could have been represented by a two-dimensional array as follows:

```

Size_TTT : constant := 3;
subtype Board_Index is Integer range 1 .. Size_TTT;
type Board_Array is
  array( Board_Index, Board_Index ) of Character;
type Board is record
  SqrS : Board_Array := ( others => (others => ' ') );
end record;
The: Board;
```

*Note: The two-dimensional initialization of the board to spaces is as follows:*  
*sqrS : Board\_grid:=( others => (others => ' ') );*  
*This type of initialization is explained in Section 8.6 Initializing an array.*

With this representation of the board individual elements are accessed as follows:

```

The.SqrS(1,2) := 'X';
The.SqrS(2,3) := 'X';
The.SqrS(3,2) := 'X';
```

## 114 Arrays

Using the new representation of Board, a procedure for displaying the contents of the board would now become:

```
procedure Display( The:in Board ) is
begin
  for I in Board_Array'Range(1) loop      --For each Row
    for J in Board_Array'Range(2) loop    -- For each column
      Put( The.Sqrs( I,J ) );             -- display counter;
      case J is                           -- column postfix
        when 1 | 2 => Put( " | " );
        when 3     => null;
      end case;
    end loop;
    case I is                             -- row postfix
      when 1 | 2 => New_Line; Put("-----"); New_Line;
      when 3     => New_Line;
    end case;
  end loop;
end Display;
```

*Note:* The statement **null** has no action, but is necessary as a statement must follow a **when** clause. The clause **when 3** cannot be omitted as this would leave the case statement not covering all the possible values for *j*.

### 8.5.1 An alternative way of declaring multidimensional arrays

Rather than declare the board as a two-dimensional array, it is also possible to declare the board as an array of rows of the board. This is accomplished as follows:

```
Size_TTT : constant := 3;
subtype Board_Index is Integer range 1 .. Size_TTT;
type Board_Row is array( Board_Index ) of Character;
type Board_Array is array( Board_Index ) of Board_Row ;
type Board is record
  Sqrs : Board_Array := ( others => (others => ' ') );
end record;
The: Board;
```

*Note:* This technique will scale to any number of dimensions.

Now to access the individual elements of the board a slightly different notation is used. The first subscript selects the row and the second subscript selects the element within the row.

```
The.Sqrs(1)(2) := 'X';
The.Sqrs(2)(3) := 'X';
The.Sqrs(3)(2) := 'X';
```

*Note:* The initialization of the two-dimensional array is performed in the same way in both cases.



### 8.5.2 Attributes of multidimensional arrays

Like single dimensional arrays, various attributes can also be extracted from multidimensional arrays. For multidimensional arrays it is, however, necessary to specify which dimension is to be interrogated. This is achieved by appending the appropriate dimension to the attribute. For example, to find the number of elements in the second dimension of the object `Sqrs` use `The.Sqrs.Length(2)`. Section B.2, Appendix B lists attributes that can be extracted from an object or type.

## 8.6 Initializing an array

A pixel on a true colour computer screen is represented by the three primary colours: red, blue, and green. Each colour has an intensity ranging from 0 (dark) to 255 (bright). This is the RGB additive colour model used by computer terminals and TVs, which is different from the CYMB subtractive colour model used in printing. In Ada, a pixel could be represented by an array of three elements representing the intensities of the primary colours. To represent the colour white, the intensity of each of the primary colours would be set to 255. A pixel can be represented by the type `Pixel_Array` as follows:

```
type Colour      is ( Red, Green, Blue );
type Intensity   is range 0 .. 255;
type Pixel_Array is array( Colour ) of Intensity;
```

A single point on the screen could be represented by the object `dot` as follows:

```
Dot      : Pixel_Array;
```

which could be initialized to black or white with the following assignments:

```
Dot := ( 0, 0, 0 );           --Black
Dot := ( 255, 255, 255 );    --White
```

The values can be named by using the subscript to the array as follows:

```
Dot := ( Red=> 255, Green=>255, Blue=>255);    --White
```

An **others** clause can be used to let the remaining elements of the array take a particular value as in:

```
Dot := Pixel_Array'( Red=>255, others=>0 );    --Red
```

*Note: When the **others** clause is used, and at least one other element is given a value by a different means, then the type of the constant must be specified. This is achieved by prefixing the constant with its type name followed by a '.*

Using a similar notation to that used in the **case** statement introduced in Section 3.8.1, a range of values may also be specified:

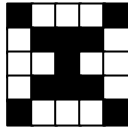
```
Dot := ( Red=>255, Green=>255, Blue=>0 );      --Yellow
Dot := ( Red | Blue => 255, Green=>0 );        --Purple
Dot := ( Red .. Blue => 127 );                 --Grey
```

## 8.6.1 Multidimensional initializations

A cursor on a black and white screen can be defined by the following declaration of an object `cursor_style`:

```
Bits_Cursor : constant Positive := 5;
type Bit     is new Integer range 0 .. 1;
type Cursor_Index is new Positive range 1 .. Bits_Cursor;
type Cursor    is array( Cursor_Index,
                        Cursor_Index ) of Bit;

Cursor_Style : Cursor;
```



Black is represented by 1

White is represented by 0

Figure 8.3 Black and white cursor.

The cursor as illustrated in Figure 8.3 could be set up in `cursor_style` with any of the following three declarations:

- by initializing every cell in the cursor individually:

```
Cursor_Style := Cursor'( (1, 0, 0, 0, 1),
                        (0, 1, 1, 1, 0),
                        (0, 0, 1, 0, 0),
                        (0, 1, 1, 1, 0),
                        (1, 0, 0, 0, 1) );
```

*Note:* The prefix to the array constant is optional in this case.

- by using an **others** clause to set up the white elements:

```
Cursor_Style := Cursor'( 1=> ( 1=>1, 5=>1, others => 0 ),
                        2=> ( 2..4 =>1, others => 0 ),
                        3=> ( 3=>1, others => 0 ),
                        4=> ( 2..4 =>1, others => 0 ),
                        5=> ( 1=>1, 5=>1, others => 0 ) );
```

- by using `|` clauses to combine any identical initializations:

```
Cursor_Style := Cursor'( 1|5=> ( 1|5 =>1, others => 0 ),
                        2|4=> ( 2..4=>1, others => 0 ),
                        3  => ( 3  =>1, others => 0 ) );
```

In a program using a colour monitor, the cursor could have been described as follows:

```
Bits_Cursor: constant Positive := 5;
type Colour    is ( Red, Green, Blue );
type Intensity is range 0 .. 255;
type Pixel_Array is array( Colour ) of Intensity;

type Cursor_Index is new Positive range 1 .. Bits_Cursor;
type Cursor    is array( Cursor_Index,
                        Cursor_Index ) of Pixel_Array;

Cursor_Style : Cursor;
```

The following code would be used to initialize the cursor to the colour grey:

```
Cursor_Style :=
Cursor'( 1|5=> ( 1|5 => (others=>127), others => (others=>0) ),
         2|4=> ( 2..4=> (others=>127), others => (others=>0) ),
         3  => ( 3  => (others=>127), others => (others=>0) ) );
```

## 8.7 Unconstrained arrays

In earlier sections, the types used to represent an array have been constrained types. These can be used to create only objects that have the specific bounds defined by the type declaration. Ada allows a user to define a type for an array that can be constrained to represent a whole family of instances of arrays, where each member of the family can potentially have different bounds. This mechanism is required when an array of arbitrary size is to be passed as a parameter to a procedure or function. For example, a function `sum` that sums the contents of an array passed to the function, can be written as follows:

```
function Sum( List:in Numbers_Array ) return Integer is
  Total : Integer := 0;
begin
  for I in List'range loop           --Depends on # of elements
    Total := Total + List( I );
  end loop;
  return Total;
end Sum;
```

The type `Numbers_Array` is an unconstrained array that has the following definition:

```
type Numbers_Array is array ( Positive range <> ) of Integer;
```

This defines a type that can be used to elaborate array objects with `Positive` type bounds. For example, an instance of the type `Numbers_Array` can be declared as follows:

```
Computers_In_Room :Numbers_Array(513..519) := (2,2,2,3,2,1,3);
```

*Note: The specific bounds of `Computers_In_Room`, an instance of `Numbers_Array` needs to be specified in the declaration.*

### 8.7.1 Slices of an array

A slice of a one-dimensional array can be obtained by selecting elements from a contiguous range from within the array. For example, to select the computers in rooms 517 to 519 from the object `Computers_In_Room` the following slice of the array can be extracted: `Computers_In_Rooms( 517 .. 519 )`.

*Note: A slice can only be taken from a one-dimensional array.*

## 118 Arrays

### 8.7.2 Putting it all together

A package `Pack_Types` is defined so that any program unit may use the constrained type declaration for `Numbers_Array` to declare arrays of this type.

```
package Pack_Types is
  type Numbers_Array is array ( Positive range <> ) of Integer;
end Pack_Types;
```

*Note: No body is required as the specification has no implementation part.*

This is then used by a program to illustrate the use of the function `sum`.

```
with Ada.Text_IO, Ada.Integer_Text_IO, Pack_Types;
use Ada.Text_IO, Ada.Integer_Text_IO, Pack_Types;
procedure Main is
  Computers_In_Room :Numbers_Array(513..519) := (2,2,2,3,2,1,3);

  -- The function sum

begin
  Put("The total number of computers is: " );
  Put( Sum( Computers_In_In_Room ) ); New_Line;

  Put("Computers in rooms 517, 518 and 519 is: " );
  Put( Sum( Computers_In_In_Room( 517 .. 519 ) ) ); New_Line;
end Main;
```

When compiled with the body of the function `sum`, the above program when run would print the results:

```
The total number of computers is:          15
Computers in rooms 517, 518 and 519 is:    6
```

## 8.8 Strings

The type `String` is a predefined, unconstrained array whose definition is:

```
type String is array ( Positive range <> ) of Character;
```

This type is defined in the package `Standard` listed in Section C.4, Appendix C. A limitation of the `String` type is that in the declaration of each instance of a string the number of characters that are to be assigned to that particular string must be specified. For example, the following program writes out the name and address of the University of Brighton.

```

procedure Main is
  type String is array ( Positive range <> ) of Character;
  Institution : String(1 .. 22);
  Address      : String(1 .. 20);
  Full_Address: String(1 .. 44);
begin
  Institution := "University of Brighton";
  Address     := "Brighton East Sussex";
  Full_Address:= Institution & ", " & Address;
  Put( Full_Address ); New_Line;
end Main;

```

When run, this would print:

```
University of Brighton, Brighton East Sussex
```

*Note: The concatenation operator & is used to deliver the join of two one-dimensional arrays.*

## 8.9 Dynamic arrays

In Ada the bounds of an array need not be fixed at compile-time, as they can be specified by an object whose value is not fixed until run-time. Such an array is known as a dynamic array. However, once elaborated, the bounds of the dynamic array cannot be changed. Unlike many other languages, Ada allows a dynamic array to be returned as the result of a function. For example, a function `Reverse_String` can be written which reverses the characters passed to it. An implementation of the function `Reverse_String` is as follows:

```

function Reverse_String( Str:in String ) return String is
  Res : String( Str'Range );           --Dynamic bounds
begin
  for I in Str'Range loop
    Res( Str'First+Str'Last-I ) := Str( I );
  end loop;
  return Res;
end Reverse_String;

```

### 8.9.1 Putting it all together

The above function `Reverse_String` is used in the following program to illustrate the use of a dynamic array:

```

with Ada.Text_IO, Reverse_String; use Ada.Text_IO;
procedure Main is
begin
  Put( Reverse_String( "madam i'm adam" ) ); New_Line;
end Main;

```

## 120 Arrays

When run, this would deliver the following results:

```
mada m'i madam
```

*Note: Even though dynamic arrays can be created, they can only be used to store an object that is type compatible. In particular, the number and type of the elements in the receiving object must be the same as in the delivered object.*

### 8.10 A name and address class

A class for managing a person's name and address has the following responsibilities.

Method	Responsibility
Set	Set the name and address of a person. The name and address is specified with a / character separating each line.
Deliver_Line	Deliver the n'th line of the address as a string.
Lines	Deliver the number of lines in the address.

The specification for the class is as follows:

```
package Class_Name_Address is
  type Name_Address is tagged private;

  procedure Set( The:out Name_Address; Str:in String );
  function Deliver_Line( The:in Name_Address;
    Line:in Positive ) return String;
  function Lines( The:in Name_Address ) return Positive;
private
  Max_Chars : constant := 200;
  subtype Line_Index is Natural range 0 .. Max_Chars;
  subtype Line_Range is Line_Index range 1 .. Max_Chars;

  type Name_Address is tagged record
    Text : String( Line_Range ); --Details
    Length : Line_Index := 0; --Length of address
  end record;
end Class_Name_Address;
```

In the implementation of the class the method `set` stores the string given as a parameter into the instance attribute `Text`. A check is made to see if the string is too long. If it is, the string is truncated and the procedure recalled recursively with the shortened name.

```
package body Class_Name_Address is

    function Spaces( Line:in Positive ) return String;

    procedure Set( The:out Name_Address; Str:in String ) is
    begin
        if Str'Length > Max_Chars then
            Set( The, Str( Str'First .. Str'First+Max_Chars-1 ) );
        else
            The.Text( 1 .. Str'Length ) := Str;
            The.Length := Str'Length;
        end if;
    end Set;
```

The function `Deliver_Line` returns a string representing the *n*'th line of the address with a staggered left margin. Spaces for the staggered left margin are calculated and delivered by the function `Spaces`.

```
function Deliver_Line( The:in Name_Address;
                      Line:in Positive ) return String is
    Line_On : Positive := 1;
begin
    for I in 1 .. The.Length loop
        if Line_On = Line then
            for J in I .. The.Length loop
                if The.Text(J) = '/' then
                    return Spaces(Line_On) & The.Text(I .. J-1);
                end if;
            end loop;
            return Spaces(Line_On) & The.Text(I..The.Length);
        end if;
        if The.Text(I) = '/' then Line_On := Line_On+1; end if;
    end loop;
    return " ";
end Deliver_Line;
```

The number of lines in an address is delivered by the function `Lines`. This function counts the number of `'/'` characters in the string `Text`.

```
function Lines( The:in Name_Address ) return Positive is
    No_Lines : Positive := 1;
begin
    for I in 1 .. The.Length loop
        if The.Text(I) = '/' then No_Lines := No_Lines + 1; end if;
    end loop;
    return No_Lines;
end Lines;
```

## 122 Arrays

The function Spaces delivers a string of Line spaces.

```
function Spaces( Line:in Positive ) return String is
  Spaces_Are : String( 1 .. Line ) := (others=>' ');
begin
  return Spaces_Are;
end Spaces;

end Class_Name_Address;
```

### 8.10.1 Putting it all together

A program to illustrate the use of the class Name\_Address is shown below:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Name_Address;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure main is
  Name      : Name_Address;
  Address   : String := "A.N.Other/Brighton/East Sussex/UK";
begin
  Set( Name, Address );
  Put( Address ); New_Line; Put("There are ");
  Put( Lines( Name ) ); Put(" lines"); New_Line;
  for I in 1 .. Lines(Name)+1 loop
    Put("Line #"); Put(I); Put( "  ");
    Put( Deliver_Line(Name, I) ); New_Line;
  end loop;
end Main;
```

which, when compiled and run, will produce the following output:

```
A.N.Other/Brighton/East Sussex/UK
There are          4 lines
Line #            1   A.N.Other
Line #            2   Brighton
Line #            3   East Sussex
Line #            4   UK
Line #            5
```

*Note: The standard library packages `Ada.Strings.Bounded` and `Ada.Strings.Unbounded` provide an elegant mechanism for handling strings of variable length. Section C.8, Appendix C lists the members of the package `Ada.Strings.Bounded`.*

## 8.11 An electronic piggy bank

Arrays are made up of objects of any type including instances of classes. For example, to implement a program to deal with a small bank's transactions, a class Piggy\_Bank can be defined which has the following methods:

Method	Responsibility
Deposit	Deposit money into a named person's account.
Withdraw	Withdraw money from a named person's account.
Balance	Obtain the balance in a named person's account.
New_Account	Allocate a new account number.



The Ada specification of the class Piggy\_Bank is as follows:

```
with Class_Account;
use Class_Account;
package Class_Piggy_Bank is
  type Piggy_Bank is private;           --Class
  subtype Money is Class_Account.Money; --Make visible
  subtype Pmoney is Class_Account.Pmoney; --Make visible

  procedure New_Account( The:in out Piggy_Bank; No:out Positive );
  procedure Deposit ( The:in out Piggy_Bank; No:in Positive;
                      Amount:in Pmoney );
  procedure Withdraw ( The:in out Piggy_Bank; No:in Positive;
                      Amount:in Pmoney; Get:out Pmoney );
  function Balance( The:in Piggy_Bank;
                   No:in Positive) return Money;
  function Valid( The:in Piggy_Bank;
                 No:in Positive) return Boolean;

private
  No_Accounts : constant := 10;
  subtype Accounts_Index is Integer range 0 .. No_Accounts;
  subtype Accounts_Range is Accounts_Index range 1 .. No_Accounts;
  type Accounts_Array is array ( Accounts_Range ) of Account;
  type Piggy_Bank is record
    Accounts: Accounts_Array; --Accounts in the bank
    Last : Accounts_Index := 0; --Last account
  end record;
end Class_Piggy_Bank;
```

*Note: The number of accounts that can be held is fixed and is defined by the constant No\_Accounts. In Chapter 17, ways of storing a variable number of (in this case instances of Account) objects are explored.*

The following code uses the class Piggy\_Bank to perform transactions on a newly allocated account:

```
with Ada.Text_io, Class_Piggy_Bank, Statement;
use Ada.Text_io, Class_Piggy_Bank;
procedure Main is
  Bank_Accounts: Piggy_Bank;           --A little bank
  Customer      : Positive;            --Customer
  Obtain        : Money;               --Money processed
begin
  New_Account( Bank_Accounts, Customer );
  if Valid( Bank_Accounts, Customer ) then
    Statement( Bank_Accounts, Customer );

    Put("Deposit £100.00 into account"); New_Line;
    Deposit( Bank_Accounts, Customer, 100.00 );
    Statement( Bank_Accounts, Customer );

    Put("Withdraw £60.00 from account"); New_Line;
    Withdraw( Bank_Accounts, Customer, 60.00, Obtain );
    Statement( Bank_Accounts, Customer );

    Put("Deposit £150.00 into account"); New_Line;
    Deposit( Bank_Accounts, Customer, 150.00 );
    Statement( Bank_Accounts, Customer );
  else
    Put("Customer number not valid"); New_Line;
  end if;
end Main;
```

## 124 Arrays

*Note: The procedure Statement is shown later at the end of this section.*

When compiled and run, the code would produce the following output:

```
Mini statement for account # 1
The amount on deposit is _ 0.00

Deposit _100.00 into account
Mini statement for account # 1
The amount on deposit is _100.00

Withdraw _60.00 from account
Mini statement for account # 1
The amount on deposit is _40.00

Deposit _150.00 into account
Mini statement for account # 1
The amount on deposit is _190.00
```

In the implementation of the package `Class_Piggy_Bank` shown below, the procedure `New_Account` allocates an account number to a new customer.

```
package body Class_Piggy_Bank is
  procedure New_Account(The:in out Piggy_Bank; No:out Positive) is
  begin
    if The.Last = No_Accounts then
      raise Constraint_Error;
    else
      The.Last := The.Last + 1;
    end if;
    No := The.Last;
  end New_Account;
```

*Note: The exception `Constraint_Error` is raised if no new accounts can be created. Chapter 14 shows how a user defined exception can be raised.*

The procedure and functions for `Deposit`, `Withdraw` and `Balance` call the appropriate code from the package `Class_Account`.

```
procedure Deposit ( The:in out Piggy_Bank; No:in Positive;
                   Amount:in Pmoney ) is
begin
  Deposit( The.Accounts(No), Amount );
end Deposit;

procedure Withdraw( The:in out Piggy_Bank; No:in Positive;
                   Amount:in Pmoney; Get:out Pmoney ) is
begin
  Withdraw( The.Accounts(No), Amount, Get);
end Withdraw;

function Balance( The:in Piggy_Bank;
                  No:in Positive) return Money is
begin
  return Balance( The.Accounts(No) );
end Balance;
```

The function valid checks the validity of an account number.

```
function Valid( The:in Piggy_Bank;
               No:in Positive) return Boolean is
begin
    return No in 1 .. The.Last;
end Valid;
end Class_Piggy_Bank;
```

The procedure Statement prints a mini statement for the selected account. This is implemented as follows:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO,
     Class_Piggy_Bank;
use   Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO,
     Class_Piggy_Bank;
procedure Statement( Bank:in Piggy_Bank; No:in Positive ) is
    In_Account : Money;
begin
    Put("Mini statement for account #");
    Put( No, Width=>3 ); New_Line;
    Put( "The amount on deposit is £" );
    In_Account := Balance( Bank, No );
    Put( In_Account, Aft=>2, Exp=>0 );
    New_Line(2);
end Statement;
```

*Note: By not having any input or output statements in the package Class\_Piggy\_Bank the package may be re-used easily in other programs.*

## 8.12 Self-assessment

- Can the index to an array be of type real?
- How can a programmer reduce the possibility of using an incorrect subscript value?
- Are there any restrictions on what type of objects can be used as array elements?
- Are there any limitations on how many dimensions an array might have?
- How is an array of objects declared?
- What is the difference between the two declarations for Board\_Array below, and how may an individual character be accessed in both cases?

```
subtype Board_Index is Integer range 1 .. 3;
type Board_Array is
    array( Board_Index, Board_Index ) of Character;
```

```
subtype Board_Index is Integer range 1 .. 3;
type Board_Row is array( Board_Index ) of Character;
type Board_Array is array( Board_Index ) of Board_Row ;
```

## 8.13 Exercises

Construct the following programs:

- *A program to play the game noughts and crosses.*

Using as a base the code for the noughts and crosses program, implement a complete program that checks for a win by a player. A win is when a player has three of their counters in a row, either diagonally, horizontally or vertically.

- *A program which maintains the records of books in a small school library.*

Each book in the library has a class mark which is a number in the range 1 — 999. A person may:

- (a) Take a book out of the library.
- (b) Return a book to the library.
- (c) Reserve a book that is out on loan.
- (d) Enquire as to the status of a book.

The program should be able to handle the recording and extracting of information required by the above transactions. In addition, a facility should be included which will provide a summary about the status of the books in the library.

Hints:

- Define the class `Book` to represent individual books in the library.
- Define a class `Library` to represent the library. The hidden internal structure of `Library` contains an array of `Books`.
- Re-use the class `TUI` to display a menu.

## 9 Case study: Design of a game

This chapter looks at the implementation of the game reversi. The problem is analysed using the fusion methodology and from this analysis and design is developed a program to play the game of reversi between two human players.

### 9.1 Reversi

In the game of reversi two players take it in turn to add counters to a board of 8-by-8 cells. Each player has a stack of counters, black one side and white the other. One player's counters are placed white side up, whilst the other player's are black side up. The object of the game is to capture all your opponent's counters. You do this by adding one of your counters to the board so that your opponent's counter(s) are flanked by two of your counters. When you do this, the counters you have captured are flipped over to become your counters. If you can't capture any of your opponent's counters during your turn, you must pass and let your opponent go.

The game is won when you have captured all your opponent's counters. If neither player can add a counter to the board, then the player with the most counters wins. If the number of counters for each player is equal, then the game is a draw.

The initial starting position is set so that the 4 centre squares in the 8-by-8 board of cells is as illustrated in Figure 9.1.

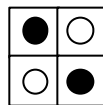
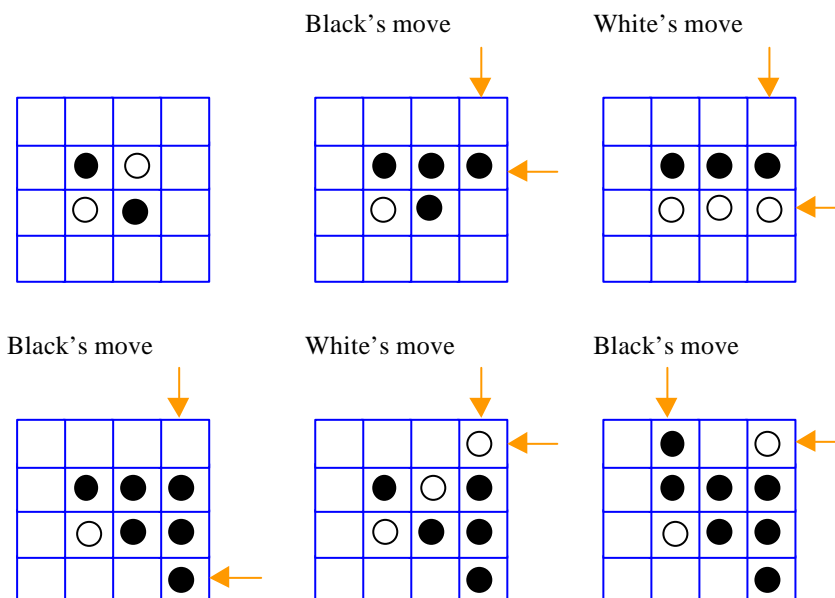


Figure 9.1 The 4 center squares of a reversi board..

On a reduced board of 4-by-4 cells a game might be as illustrated in Figure 9.2.



## 128 Case study: Design of a game

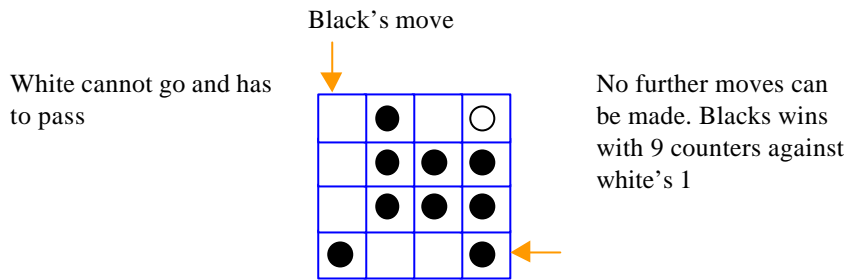


Figure 9.2 A game of reversi.

### 9.1.1 A program to play reversi

A controller of the game (games master) asks each player in turn for a move. When a move is received from a player, the board is asked to validate the move. If this is a valid move, the counter of the current player is added to the board. The board is displayed and the new state of the board is evaluated. This process is repeated until either the board is filled or neither player can make a move. The player making the last move is asked to announce the result of the game.

The interactions by the controller with the system are shown in Figure 9.3

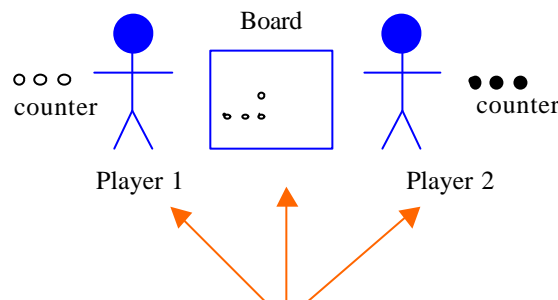


Figure 9.3 Interactions by the controller with the objects in the system.

## 9.2 Analysis and design of the problem

Using a design methodology based on a simplified version of fusion the above specification can be analysed and a design created for an eventual implementation in Ada. In preparation for this, it is appropriate to identify the objects and system actions from the written specification. An easy but incomplete way of identifying objects and system actions is to identify the major nouns and verbs. The nouns in the specification become the objects and the major verbs become the system actions.

With the major **nouns** indicated in bold type and the major *verbs* in bold italic type, the specification for the game reversi can now be read as:

In the **game** of reversi two **players** take it in turn to *add* **counters** to a **board** of 8-by-8 **cells**. Each **player** has a stack of **counters** black one side and white the other. One **player's** **counters** are placed white side up, whilst the other **player's** are black side up. The object of the game is to *capture* all your opponent's **counters**. You do this by adding one of your **counters** to the **board** so that your opponent's **counter(s)** are flanked by two of your **counters**. When you do this, the **counters** you have captured are *flipped* over to become your **counters**. If you can't capture any of your opponent's **counters** during your turn, you must pass and let your opponent go.

The **game** is won when you have *captured* all your opponent's **counters**. If neither **player** can add a **counter** to the **board**, then the **player** with the most **counters** wins. If the number of **counters** for each **player** is equal, then the game is a draw.

A controller of the **game** (games master) *asks* each player in turn for a move. When a move is received from a **player** the **board** is asked to *validate* the move. If this is a valid move the **counter** of the current **player** is *added* to the **board**. The **board** is *displayed* and the new state of the board is *evaluated*. This process is repeated until either the **board** is filled or neither **player** can make a move. The **player** making the last move is asked to *announce* the result of the **game**.

The major objects and verbs identified are:

Objects (nouns)	Messages (verbs)
board	add
game	announce
cell	ask
counter	evaluated
player	capture
game	display
	validate

The following messages are sent to individual objects:

board  
     Display a representation of the board.  
     Add a counter to the board.  
     Evaluate the current state of the board.  
     Validate a proposed move.

player  
     Announce the result of the game.  
     Ask for the next move.

cell  
     Add a counter into a cell on the board.

counter  
     Display a representation of the counter.

Play  
     Play the game.

It is more appropriate to deal with classes than to deal with objects. For example, Board is the class to which the object board belongs. Using this approach the messages sent to these classes can be refined into the following list:

Class	Message	Responsibility of method
Board	Add	Add a counter into the board.
	Check_Move	Check if a player can drop a counter into a column.
	Contents	Return the contents of a cell.
	Display	Display a representation of the board.
	Now_playing	Say who is now playing on the board.
	Set_Up	Populate the board with the initial contents.
	Status	Evaluate the current state of the board.
Player	Announce	Announcing that the player has either won or drawn the game.
	Get_Move	Get the next move from the player.
	My_Counter	Return the counter that the player plays with.
	Set	Set a player to play with a particular counter.

## 130 Case study: Design of a game

<b>Cell</b>	Add	Add a counter to a cell.
	Display	Display the contents of a cell.
	Flip	Flip the contents of a cell.
	Holds	Return the contents of a cell.
	Initialize	Initialize a cell.

<b>Class</b>	<b>Message</b>	<b>Responsibility of method</b>
<b>Counter</b>	Display	Display a counter.
	Flip	Flip a counter.
	Rep	Return the colour of a counter.
	Set	Set a counter to be black/white.
<b>Game</b>	Play	Play the game.

Note: Some of the original messages (verbs) have been renamed to a more specific name when producing this list.

### 9.3 Class diagram

A class diagram for the game of draughts is shown below in Figure 9.4

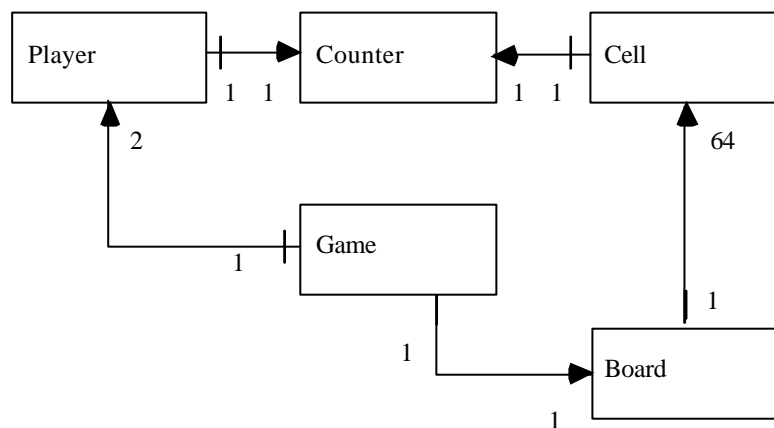


Figure 9.4 Relationship between the classes in the game of four counters.

### 9.4 Specification of the Ada classes

The Ada class specifications for the above classes are implemented as follows:

<b>Class</b>	<b>Ada specification</b>
<b>Game</b>	<pre>package Class_Game is   type Game      is private;   procedure Play( The:in Game ); private end Class_Game;</pre>



Counter	<pre> package Class_Counter is   type Counter          is private;   type Counter_Colour is ( Black, White );   procedure Set( The:in out Counter; Rep:in Counter_Colour );   procedure Display( The:in Counter );   procedure Display_None( The:in Counter );   procedure Flip( The:in out Counter );   function Rep( The:in Counter ) return Counter_Colour; private end Class_Counter; </pre>
Player	<pre> package Class_Player is   type Player is private;    procedure Set( The:in out Player; C:in Counter_Colour );   procedure Get_Move(The:in Player; Row,Column:out Integer);   function My_Counter( The:in Player ) return Counter;   procedure Announce( The:in Player; What:in State_Of_Game ); private end Class_Player; </pre>
Cell	<pre> package Class_Cell is   type Cell is private;   type Cell_Holds is ( C_White, C_Black, Empty );    procedure Initialize( The:in out Cell );   function Holds( The:in Cell ) return Cell_Holds;   procedure Add( The:in out Cell; Players_Counter:in Counter );   procedure Display( The:in Cell );   procedure Flip( The:in out Cell );   function To_Colour( C:in Cell_Holds ) return Counter_Colour; private end Class_Cell; </pre>
Board	<pre> package Class_Board is    type Board          is private;   type State_Of_Game is ( Play, Win, Draw, Lose );   type Move_Status    is ( Ok, Invalid, Pass );    procedure Set_Up( The:in out Board );   procedure Add( The:in out Board; X,Y:in Integer;                 Move_Is:in Move_Status );   procedure Now_Playing( The:in out Board; C:in Counter_Colour );   procedure Display( The:in Board );   function Check_Move( The:in Board; X,Y:in Integer )                     return Move_Status;   function Status( The:in Board ) return State_Of_Game;   function Contents( The:in Board; X,Y:in Integer )                     return Cell_Holds; private end Class_Board; </pre>

## 9.5 Implementation of the main class Game

Using the design carried out above, the specification of the class Game is :

```
with Class_Board, Class_Player, Class_Counter;
use Class_Board, Class_Player, Class_Counter;
package Class_Game is
  type Game is private;
  procedure play( The:in out Game );
private
  type Player_Array is array(Counter_Colour) of Player;
  type Game is record
    Reversi      : Board;                --The playing board
    Contestant   : Player_Array;
  end record;
end Class_Game;
```

and the implementation is as follows:

```
package body Class_Game is
  procedure Play( The:in out Game ) is      --Play reversi
    Current_State : State_Of_Game;         --State of game
    Person        : Counter_Colour;        --Current player
    X, Y          : Integer;               --Move
    Move_Is       : Move_Status;           --Last move is
  begin
    Set_Up( The.Reversi );                 --Set up board
    Set( The.Contestant(Black), Black );    --Set player black
    Set( The.Contestant(White), White );    --Set player white
```

```

Current_State := Play;  Person := Black;  --Black starts
Display( The.Reversi );  --Initial board
while Current_State = Play loop  --Playable game
    Now_Playing( The.Reversi, Person );  --set player

    loop  --Get move
        Get_Move(The.Contestant(Person), X, Y);
        Move_Is:=Check_Move(The.Reversi, X, Y);--Validate
        exit when Move_Is=Ok or Move_Is=Pass;  --OK
    end loop;

    Add( The.Reversi, X, Y, Move_Is );  --Add move to board

    Display( The.Reversi );  --Display new board
    Current_State := Status( The.Reversi );  --State of play is

    if Current_State = Play then  --Is still playable
        case Person is  --next player
            when Black => Person := White;
            when White => Person := Black;
        end case;
    end if;

end loop;  --Next move

Announce( The.Contestant(Person), Current_State );  --Result
end Play;
end Class_Game;

```

### 9.5.1 Running the program

Then to run the game the following procedure is used to send the message Play to an instance of the class Game.

```

with Class_Game;
use Class_Game;
procedure Main is
    A_Game : Game;
begin
    Play( A_Game );
end Main;

```

### 9.5.2 Example of a typical game

A typical game might be:

## 134 Case study: Design of a game

				x	o			
				o	x			
Player X has 2 counters -								
Player O has 2 counters								
Please enter move X row column: 4 6								

				x	x	x		
				o	x			
Player X has 4 counters -								
Player O has 1 counters								
Please enter move O row column: 5 6								

				x	x	x		
				o	o	o		
Player X has 3 counters -								
Player O has 3 counters								
Please enter move O row column: 6 6								

				x	x	x		
				o	x	x		
						x		
Player X has 6 counters -								
Player O has 1 counters								
Please enter move O row column: 0 0								

-----									
-----									
-----									
				X	X	X			
-----									
				O	X	X			
-----									
					X				
-----									
-----									
-----									
Player X has 6 counters -									
Player O has 1 counters									
Please enter move X row column: 6 4									

-----									
-----									
-----									
				X	X	X			
-----									
				X	X	X			
-----									
				X		X			
-----									
-----									
-----									
Player X has 8 counters -									
Player O has 0 counters									
Player X has won									

## 9.6 Implementation of the other classes

The main program is implemented using the classes described earlier. The package Pack\_Screen is responsible for handling the machine specific action of clearing the screen. Using an ANSI terminal, its implementation is as follows:

```
package Pack_Screen is
  procedure Screen_Clear;           --Home clear screen
  procedure Screen_Home;           --Home no clear screen
private
  Esc: constant Character := Character'Val(27);
end Pack_Screen;
```

The implementation of this package uses ANSI escape sequences to implement these procedures. If an ANSI compatible terminal is not available, the bodies of these procedures can be changed to implement an appropriate alternative.

```
with Text_IO; use Text_IO;
package body Pack_Screen is
  procedure Screen_Clear is         --Terminal dependent I/O
  begin                             --Clear screen
    Put( Esc & "[2J" );           --Escape sequence
  end Screen_Clear;
  procedure Screen_Home is         --Home
  begin                             --Escape sequence
    Put( Esc & "[0;0H" );
  end Screen_Home;
end Pack_Screen;
```

The specification of the class Counter is as follows:

## 136 Case study: Design of a game

```
package Class_Counter is
  type Counter is private;
  type Counter_Colour is ( Black, White );
  procedure Set( The:in out Counter; Rep:in Counter_Colour );
  procedure Display( The:in Counter );
  procedure Display_None( The:in Counter );
  procedure Flip( The:in out Counter );
  function Rep( The:in Counter ) return Counter_Colour;
private
  type Counter is record
    Colour: Counter_Colour;           --Colour of counter
  end record;
end Class_Counter;
```

The procedure Set sets a counter to a specific colour.

```
with Ada.Text_IO;
use Ada.Text_IO;
package body Class_Counter is
  procedure Set( The:in out Counter; Rep:in Counter_Colour ) is
  begin
    The.Colour := Rep;
  end Set;
```

The procedure Display and Display\_None respectively display the contents of a counter or no counter.

```
procedure Display( The:in Counter ) is
begin
  case The.Colour is
    when Black => Put('X'); --Representation of a black piece
    when White => Put('O'); --Representation of a white piece
  end case;
end Display;

procedure Display_None( The:in Counter ) is
begin
  Put(' ');           --Representation of NO piece
end Display_None;
```

The procedure Flip flips a counter. By flipping a counter the other player's colour is exposed, whilst the procedure Rep returns the colour of the counter.

```
procedure Flip( The:in out Counter ) is
begin
  case The.Colour is
    when Black => The.Colour := White;   --Flip to White
    when White => The.Colour := Black;   --Flip to Black
  end case;
end Flip;

function Rep( The:in Counter ) return Counter_Colour is
begin
  return The.Colour; --Representation of the counter colour
end Rep;
end Class_Counter;
```

The specification for the class Cell which holds a counter is as follows:

```
with Class_Counter;
use Class_Counter;
package Class_Cell is
  type Cell is private;
  type Cell_Holds is ( C_White, C_Black, Empty );

  procedure Initialize( The:in out Cell );
  function Holds( The:in Cell ) return Cell_Holds;
  procedure Add( The:in out Cell; Players_Counter:in Counter );
  procedure Display( The:in Cell );
  procedure Flip( The:in out Cell );
  function To_Colour( C:in Cell_Holds ) return Counter_Colour;
private
  type Cell_Is is ( Empty_Cell, Not_Empty_Cell );
  type Cell is record
    Contents: Cell_Is := Empty_Cell;
    Item      : Counter;           --The counter
  end record;
end Class_Cell;
```

In the implementation of the package the procedure Initialize sets the contents of the cell to empty.

```
package body Class_Cell is
  procedure Initialize( The:in out Cell ) is
  begin
    The.Contents := Empty_Cell;  --Initialize cell to empty
  end Initialize;
```

The procedure Holds returns the contents of the cell which is defined by the enumeration type Cell\_Holds.

```
function Holds( The:in Cell ) return Cell_Holds is
begin
  case The.Contents is
    when Empty_Cell =>           --Empty
      return Empty;              -- No counter
    when Not_Empty_Cell =>       --Counter
      case Rep( The.Item ) is
        when White => return C_White; -- white counter
        when Black => return C_Black; -- black counter
      end case;
    end case;
  end Holds;
```

The next three procedures implement:

- Adding of a new counter into a cell.
- Displaying the contents of a cell.
- Flipping the counter in the cell to the other colour.

```
procedure Add(The:in out Cell; Players_Counter:in Counter) is
begin
  The := (Not_Empty_Cell,Players_Counter);
end Add;
```

## 138 Case study: Design of a game

```
procedure Display( The:in Cell ) is
begin
  if The.Contents = Not_Empty_Cell then
    Display( The.Item );      --Display the counter
  else
    Display_None( The.Item );  --No counter
  end if;
end Display;

procedure Flip( The:in out Cell ) is
begin
  Flip( The.Item );          --Flip counter
end Flip;
```

The function `To_Colour` converts the enumeration `Cell_Holds` to the enumeration `Counter_Colour`. This method is required so that the contents of a `Cell` can be processed as a `Counter_Colour`. The board holds the colour of the current player. It is an error to ask for the colour of an empty cell.

```
function To_Colour(C:in Cell_Holds) return Counter_Colour is
begin
  case C is
    when C_White => return White;
    when C_Black => return Black;
    when others => raise Constraint_Error;
  end case;
end To_Colour;

end Class_Cell;
```

Note: The code associated with the **when others** clause will never be executed.

The package `Class_Board` is by far the most complex of the packages used in this implementation. As well as several visible functions and procedures, it also has several private functions and procedures. The main complexity occurs in the function `Check_Move` and the function `add`.

```
with Class_Counter, Class_Cell;
use Class_Counter, Class_Cell;
package Class_Board is

  type Board          is private;
  type State_Of_Game is ( Play, Win, Draw, Lose );
  type Move_Status    is ( OK, Invalid, Pass );

  procedure Set_Up( The:in out Board );
  procedure Add( The:in out Board; X,Y:in Integer;
                Move_Is:in Move_Status );
  procedure Now_Playing( The:in out Board; C:in Counter_Colour );
  procedure Display( The:in Board );
  function Check_Move( The:in Board; X,Y:in Integer )
    return Move_Status;
  function Status( The:in Board ) return State_Of_Game;
  function Contents( The:in Board; X,Y:in Integer )
    return Cell_Holds;
```



```

private
  Size: constant := 8;                                --8 * 8 Board
  subtype Board_Index is Integer range 1 .. Size; --

  type Board_Array is array (Board_Index, Board_Index) of Cell;
  type Score_Array is array (Counter_Colour) of Natural;
  type Move_Array is array (Counter_Colour) of Move_Status;

  type Board is record
    Sqr      : Board_Array;           --Game board
    Player    : Counter_Colour;       --Current Player
    Opponent  : Counter_Colour;       --Opponent
    Score     : Score_Array;          --Running score
    Last_Move : Move_Array;           --Last move is
  end record;
end Class_Board;

```

The body of `Class_Board` contains specifications of functions and procedures which are used in the decomposition of the methods of the class `Board`.

```

with Ada.Text_IO, Ada.Integer_Text_IO, Pack_Screen;
use   Ada.Text_IO, Ada.Integer_Text_IO, Pack_Screen;
package body Class_Board is

  procedure Next( The:in Board; X_Co,Y_Co:in out Board_Index;
                  Dir:in Natural; Res:out Boolean);
  function Find_Turned( The:in Board; X,Y: in Board_Index )
    return Natural;
  procedure Turn_Counters(The: in out Board; X,Y: in Board_Index;
                          Total: out Natural );
  function No_Turned(The:in Board; O_X,O_Y:in Board_Index;
                    Dir:in Natural;
                    N:in Natural := 0 ) return Natural;
  procedure Capture(The:in out Board; X_Co, Y_Co:in Board_Index;
                   Dir:in Natural );

```

The procedure `setup` populates the board with empty cells and the initial central grid of four counters.

```

procedure Set_Up( The:in out Board ) is
  Black_Counter: Counter;           --A black counter
  White_Counter: Counter;           --A white counter
begin
  Set( Black_Counter, Black );       --Set black
  Set( White_Counter, White );       --Set white
  for X in The.Sqr's'range(1) loop
    for Y in The.Sqr's'range(2) loop
      Initialize( The.Sqr(X,Y) );    --To empty
    end loop;
  end loop;
  Add( The.Sqr( Size/2, Size/2 ), Black_Counter );
  Add( The.Sqr( Size/2, Size/2+1 ), White_Counter );
  Add( The.Sqr( Size/2+1, Size/2 ), White_Counter );
  Add( The.Sqr( Size/2+1, Size/2+1 ), Black_Counter );
  The.Score( Black ) := 2; The.Score( White ) := 2;
end Set_Up;

```

## 140 Case study: Design of a game

The procedure `Now_Playing` records the colour of the current player. This information is used by subsequent methods `add` and `Check_Move`.

```
procedure Now_Playing(The:in out Board; C:in Counter_Colour) is
begin
  The.Player := C;                                --Player
  case C is                                       --Opponent
    when White => The.Opponent := Black;
    when Black => The.Opponent := White;
  end case;
end Now_Playing;
```

The procedure `Display` displays a representation of the reversi board on the output device. For this implementation of the game, the output device is an ANSI text-based terminal.

```
procedure Display( The:in Board ) is
  Dashes: String( 1 .. The.Sqrs'Length*4+1 ) := ( others=>'-' );
begin
  Screen_Clear;                                   --Clear screen
  Put( Dashes ); New_Line;                       --Top
  for X in The.Sqrs'range(1) loop
    Put( " | " );                                --Cells on line
    for Y in The.Sqrs'range(2) loop
      Put( " " ); Display( The.Sqrs(X,Y) ); Put( " | " );
    end loop;
    New_Line; Put( Dashes ); New_Line;           --Bottom lines
  end loop;
  New_Line;
  Put( "Player X has " );
  Put( Integer(The.Score(Black)), Width=>2 );
  Put( " counters" ); New_Line;
  Put( "Player O has " );
  Put( Integer(The.Score(White)), Width=>2 );
  Put( " counters" ); New_Line;
end Display;
```

The function `Check_Move` checks the validity of a proposed move on the board. This function is decomposed into the function `Find_Turned` which calculates the number of pieces that will be turned if a move is made into the specified square.

```
function Check_Move( The:in Board; X,Y:in Integer )
  return Move_Status is
begin
  if X = 0 and then Y = 0 then
    return Pass;
  elsif X in Board_Index and then Y in Board_Index then
    if Holds( The.Sqrs( X, Y ) ) = Empty then
      if Find_Turned(The, X, Y) > 0 then
        return Ok;
      end if;
    end if;
  end if;
  return Invalid;
end Check_Move;
```

The function `Find_Turned` finds the number of the opponent's counters that would be turned for a particular move. The strategy for `Find_Turned` is to sum the number of opponent's counters which will be flipped in each compass direction. For any position on the board there are potentially eight directions to check. The directions are illustrated in Figure 9.5.

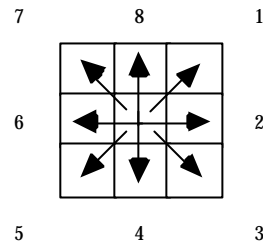


Figure 9.5 Compass direction to check when a new counter is added to the board.

```
function Find_Turned( The:in Board; X,Y: in Board_Index )
    return Natural is
        Sum      : Natural := 0;           --Total stones turned
begin
    if Holds( The.Sqrs( X, Y ) ) = Empty then
        for Dir in 1 .. 8 loop           --The 8 possible directions
            Sum := Sum + No_Turned( The, X, Y, Dir );
        end loop;
    end if;
    return Sum;                         --return total
end Find_Turned;
```

The recursive function `No_Turned` counts the number of the opponent's pieces that would be captured. This may of course be zero.

```
function No_Turned(The:in Board; O_X,O_Y:in Board_Index;
    Dir:in Natural;
    N:in Natural := 0 ) return Natural is
    Ok : Boolean;           --Result from next
    Nxt: Cell_Holds;        --Next in line is
    Col: Counter_Colour;    --Counter colour
    X   : Board_Index := O_X; --Local copy
    Y   : Board_Index := O_Y; --Local copy
begin
    Next( The, X,Y, Dir, Ok );           --Next cell
    if Ok then                           --On the board
        Nxt := Holds( The.Sqrs(X,Y) );   --Contents are
        if Nxt = Empty then              --End of line
            return 0;
        else
            Col := To_Colour( Nxt );      --Colour
            if Col = The.Opponent then   --Opponents counter
                return No_Turned(The, X,Y, Dir, N+1); --Try next cell
            elsif Col = The.Player then   --End of counters
                return N;                 --Counters turned
            end if;
        end if;
    else
        return 0;                       --No line
    end if;
end No_Turned;
```

## 142 Case study: Design of a game

The procedure `Next` returns the position of the next cell in the current direction. If there is no such cell because the edge of the board has been reached, then `Res` is set to `False`.

```
procedure Next( The:in Board; X_Co,Y_Co:in out Board_Index;
  Dir:in Natural; Res:out Boolean) is
  X, Y : Natural;
begin
  X := X_Co; Y := Y_Co;           --May go outside Board_range
  case Dir is
    when 1 => Y:=Y+1; -- Direction to move
    when 2 => X:=X+1; Y:=Y+1; --      8   1   2
    when 3 => X:=X+1; --
    when 4 => X:=X+1; Y:=Y-1; --      7   *   3
    when 5 => Y:=Y-1; --
    when 6 => X:=X-1; Y:=Y-1; --      6   5   4
    when 7 => X:=X-1; --
    when 8 => X:=X-1; Y:=Y+1; --
    when others => raise Constraint_Error;
  end case;
  if X in Board_Index and then Y in Board_Index then
    X_Co := X; Y_Co := Y; --
    Res := True; --Found a next cell
  else
    Res := False; --No next cell
  end if;
end Next;
```

The procedure `Add` adds a player's move to the board. Naturally this must be a valid move which has previously been validated with the function `Check_Move`. The type of move is recorded so that a draw can be detected when both players have passed on their last move.

```
procedure Add( The:in out Board; X,Y:in Integer;
  Move_Is:in Move_Status ) is
  Plays_With: Counter; --Current player's counter
  Turned : Natural; --Number counters turned
begin
  Set( Plays_With, The.Player ); --Set current colour
  The.Last_Move( The.Player ) := Move_Is; --Last move is
  if Move_Is = Ok then --Not Pass
    Turn_Counters(The, X,Y, Turned); --and flip
    Add( The.Sqrs( X, Y ), Plays_With ); --to board
    The.Score( The.Player ) :=
      The.Score( The.Player ) + Turned + 1;
    The.Score( The.Opponent ):=
      The.Score( The.Opponent ) - Turned;
  end if;
end Add;
```

The procedure `Turn_Counters` implements the turning of the opponents counters on the board. Naturally, for this to be called, the move made must be valid.

```

procedure Turn_Counters(The: in out Board; X,Y: in Board_Index;
                        Total: out Natural ) is
    Num_Cap  : Natural := 0;
    Captured : Natural;
begin
    if Holds( The.Sqrs( X, Y ) ) = Empty then
        for Dir in 1 .. 8 loop
            Captured := No_Turned( The, X, Y, Dir );
            if Captured > 0 then
                Capture( The, X, Y, Dir );
                Num_Cap := Num_Cap + Captured;
            end if;
        end loop;
    end if;
    Total := Num_Cap;
end Turn_Counters;

```

The recursive procedure `Capture` implements the physical capture of the opponent's counters. The strategy is to flip the opponent's counters in the current direction until a square containing the current player's counters is found.

```

procedure Capture(The: in out Board; X_Co, Y_Co: in Board_Index;
                  Dir: in Natural ) is
    Ok      : Boolean;           --There is a next cell
    X, Y    : Board_Index;      --Coordinates of cell
    Nxt     : Cell_Holds;       --Next in line is
begin
    X := X_Co; Y := Y_Co;
    Next( The, X, Y, Dir, Ok ); --Calculate pos next cell
    if Ok then                  --Cell exists (Must)
        Nxt := Holds( The.Sqrs(X,Y) );
        if To_Colour( Nxt ) = The.Opponent then
            Flip( The.Sqrs(X, Y) ); --Capture
            Capture(The, X, Y, Dir ); --Implement capture
        else
            return;              --End of line
        end if;
    else
        raise Constraint_Error;   --Will never occur
    end if;
end Capture;

```

## 144 Case study: Design of a game

The procedure Status returns the current state of the game. This may be a draw if both players have passed on their last go.

```
function Status ( The:in Board ) return State_Of_Game is
begin
  if The.Score( The.Opponent ) = 0 then
    return Win;
  end if;
  if (The.Sqrs'Length(1) * The.Sqrs'Length(2) =
    The.Score(The.Opponent)+The.Score(The.Player)) or
    (The.Last_Move(Black)=Pass and The.Last_Move(White)=Pass)
  then
    if The.Score(The.Opponent) = The.Score(The.Player)
    then return Draw;
    end if;
    if The.Score(The.Opponent) < The.Score(The.Player)
    then return Win;
    else
      return Lose;
    end if;
  end if;
  return Play;
end;
```

Whilst not used, the function Contents is provided so that another user of the class Board could find the contents of individual cells.

```
function Contents( The:in Board; X,Y:in Integer )
  return Cell_Holds is
begin
  return Holds( The.Sqrs( X, Y ) );
end Contents;

end Class_Board;
```

The package Class\_Player is responsible for communicating with the actual human player playing the game.

```
with Class_Counter, Class_Board;
use Class_Counter, Class_Board;
package Class_Player is
  type Player is private;

  procedure Set( The:in out Player; C:in Counter_Colour );
  procedure Get_Move(The:in Player; Row,Column:out Integer);
  function My_Counter( The:in Player ) return Counter;
  procedure Announce( The:in Player; What:in State_Of_Game );
private
  type Player is record
    Plays_With : Counter;           --Player's counter
  end record;
end Class_Player;
```

In the implementation of the class `Class_Player` the procedure `Set` sets the colour for the player's counter.

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
package body Class_Player is
  procedure Set(The:in out Player; C:in Counter_Colour ) is
    A_Counter : Counter;
  begin
    Set( A_Counter, C );           --Set colour
    The.Plays_With := A_Counter;  --Player is playing with
  end Set;
```

The procedure `Get_Move` communicates with the human player using a simple text based interaction.

```
procedure Get_Move(The:in Player; Row,Column:out Integer) is
  Valid_Move : Boolean := False;
begin
  while not Valid_Move loop
    begin
      Put("Please enter move "); Display( The.Plays_With );
      Put(" row column : "); Get( Row ); Get( Column );
      Valid_Move := True;
    exception
      when Data_Error =>
        Row := -1; Column := -1; Skip_Line;
      when End_Error =>
        Row := 0; Column := 0;
        return;
    end;
  end loop;
end Get_Move;
```

*Note:* A player can pass a turn by entering a coordinate of 0, 0.

The counter that the player plays with is returned by the function `My_Counter`.

```
function My_Counter( The:in Player ) return Counter is
begin
  return The.Plays_With;
end My_Counter;
```

## 146 Case study: Design of a game

The procedure `Announce` communicates with the human player the result of the game.

```
procedure Announce(The:in Player; What:in State_Of_Game) is
begin
  case What is
    when Win =>
      Put( "Player " ); Display( The.Plays_With );
      Put( " has won" );
    when Lose =>
      Put( "Player " ); Display( The.Plays_With );
      Put( " has lost" );
    when Draw =>
      Put( "It's a draw" );
    when others =>
      raise Constraint_Error;
  end case;
  New_Line;
end Announce;

end Class_Player;
```

## 9.7 Self-assessment

- What is the function of the class `Player`?
- What is the function of the class `Cell`?
- Could the recursive function `no_turned` in the class `Board` be written non-recursively?
- The procedure `announce` in the class `Board` has a `case` statement with a `when others` clause that can never occur. Why is this clause necessary?

## 9.8 Exercises

- *Better 'reversi'*  
Modify the program to have a separate class for all input and output.
- *Graphic 'reversi'*  
The program could be modified by providing additional classes to present a graphical display of the board. The display could enable the user to drop a counter into a cell selected by a using a mouse. Describe the modifications required to implement this new version.
- *Implementation of a graphic 'Reversi'*  
Implement this new graphical version.



# 10 Inheritance

This chapter introduces the concept of inheritance in which an existing class can be specialized without modifying the original class. By using this technique software re-use can become a practical consideration when developing software. Thus a programmer can become a builder of software using previously developed components.

## 10.1 Introduction

Inheritance is the ability to create a new class by using the methods and instance attributes from an existing class in the creation of a new class. For example, a class `Account` that provides the methods `deposit`, `withdraw`, `balance`, and `statement` can be used as the base for a new class that provides the ability to pay interest on the outstanding balance in the account. The new class `Interest_Account` inherits all the methods and instance attributes from the class `Account` and adds to these the new methods of `Calc_Interest`, `Add_Interest`, and `Set_Rate` plus the instance attribute `Accumulated_Interest`. This is illustrated in Figure 10.1.

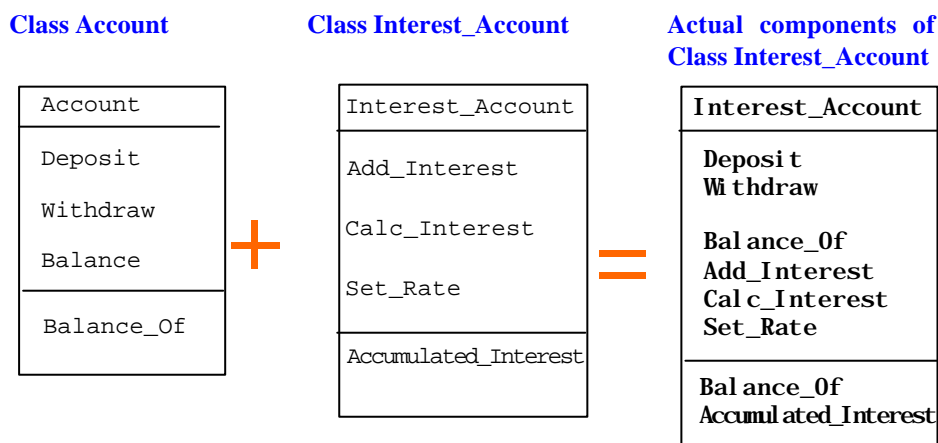


Figure 10.1 components of `Account` and the derived class `Interest_account`.

*Note: The class `Interest_Account` has the same visibility of components in the base class `Account` as would a client of the class. In particular, it has no access to the private instance attributes of the base class. Thus methods in the class `Interest_Account` cannot access the base class instance attribute `Balance_Of`.*

However, for a class type to be fully extended it must be declared as tagged. If a class type is not tagged then it can only be extended by adding new methods. New instance attributes may not be added.

The consequence of this is that an implementor of a class must explicitly declare the class record type tagged if new classes are to be derived from it.

## 10.2 Tagged types

A tagged record type declaration is very similar to an ordinary type declaration. For example, the specification of a class Account shown in Section 6.3.4 can be amended to allow inheritance to take place. The new specification for the class Account is as follows:

```
package Class_Account is
  type Account is tagged private;
  subtype Money is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Deposit( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account;
    Amount:in Pmoney; Get:out Pmoney );
  function Balance( The:in Account ) return Money;
private
  type Account is tagged record
    Balance_Of : Money := 0.00;           --Amount on deposit
  end record;
end Class_Account;
```

*Note:* The only difference from the previous class specification for Account is the inclusion of the keyword *tagged*.

The implementation of the class Class\_Account remains the same. This implementation is shown in Section 6.3.6.

### 10.2.1 Terminology

Terminology	Explanation
Base class / Super class	A class from which other classes are derived from.
Derived class / Sub class	A new class that specializes an existing class

## 10.3 The class Interest\_account

From the class Account can be derived a new type of account that pays interest on the outstanding balance at the end of each day. This new class will have the additional methods of:

Method	Responsibility
Calc_Interest	Calculate at the end of the day the interest due on the balance of the account. This will be accumulated and credited to the account at the end of the accounting period.
Add_Interest	Credit the account with the accumulated interest for the accounting period.
Set_Rate	Set the interest rate for all instances of the class.

The method Set\_Rate is special as it has the responsibility of setting the interest rate for all instances of the class. This is implemented by setting the shared class attribute `the_Interest_Rate`. When a variable is declared outside the class record type there is only a single instance of the attribute and this single class attribute is shared between, and visible to all instances of the class. However, it is not visible outside the class as it is declared within the private part of the package specification. This is illustrated in Figure 10.2

Two instances of the class `Interest_Account` sharing the same class attribute `the_interest_rate`.

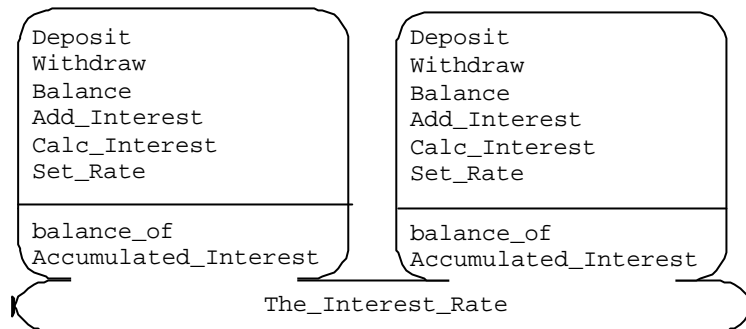


Figure 10.2 Illustration of a class global variable.

The Ada specification for the inherited class `Class_Interest_Account` is:

```

with Class_Account;
use Class_Account;
package Class_Interest_Account is

  type Interest_Account is new Account with private;

  procedure Set_Rate( Rate:in Float );
  procedure Calc_Interest( The:in out Interest_Account );
  procedure Add_Interest( The:in out Interest_Account );
private
  Daily_Interest_Rate: constant Float := 0.00026116; --10%
  type Interest_Account is new Account with record
    Accumulated_Interest : Money := 0.00; --To date
  end record;
  The_Interest_Rate : Float := Daily_Interest_Rate;
end Class_Interest_Account;
  
```

*Note:* The declaration of the class `Interest_Account` is defined as an extension to the existing class `Account`. The specification for the additional procedures are defined in the public part of the specification.  
 The class attribute `The_Interest_Rate` is shared amongst all the instances of the class `Interest_Account`.  
 As the procedure `set_rate` only accesses class attributes, an instance of the class is not required as a parameter. This type of method is referred to as a class method.

The class `Interest_account` contains:

- The following methods:

Defined in Class_Account	Defined in Class_interest_account
Deposit	Calc_Interest
Withdraw	Add_Interest
Balance	Set_Rate
Statement	

- The following instance and class attributes:

Defined in Class_Account	Defined in Class_interest_account
Balance_Of	Accumulated_Interest
	The_Interest_Rate

*Note:* Only `accumulated_interest` and `The_Interest_Rate` may be accessed by methods defined in the class `Interest_Account`.

## 150 *Inheritance*

The implementation of the class `Class_Interest_Account` is:

```
package body Class_Interest_Account is

  procedure Set_Rate( Rate:in Float ) is
  begin
    The_Interest_Rate := Rate;
  end Set_Rate;

  procedure Calc_Interest( The:in out Interest_Account ) is
  begin
    The.Accumulated_Interest := The.Accumulated_Interest +
      Balance(The) * The_Interest_Rate;
  end Calc_Interest;
```

```
  procedure Add_Interest( The:in out Interest_Account ) is
  begin
    Deposit( The, The.Accumulated_Interest );
    The.Accumulated_Interest := 0.00;
  end Add_Interest;

end Class_Interest_Account;
```

The procedure `Statement` will print a mini-statement for an `Account`

```
with Ada.Text_IO, Ada.Float_Text_IO, Class_Account;
use   Ada.Text_IO, Ada.Float_Text_IO, Class_Account;
procedure Statement( An_Account:in Account ) is
begin
  Put( "Mini statement: The amount on deposit is f" );
  Put( Balance(An_Account), Aft=>2, Exp=>0 );
  New_Line(2);
end Statement;
```

The two classes `Account` and `Interest_Account` can be used in a program to perform some simple bank transactions. A program to illustrate the use of the new class `Interest_Account` and the original class `Account` is shown below:

```
with Ada.Text_io,
     Class_Interest_Account, Class_Account, Statement;
use   Ada.Text_io,
     Class_Interest_Account, Class_Account;
procedure Main is
  Mike      :Account;           --Normal Account
  Corinna   :Interest_Account;  --Interest bearing account
  Obtained:Money;
begin
  Set_Rate( 0.00026116 );      --For all instances of
                               --interest bearing accounts

  Statement( Mike );

  Put("Deposit £50.00 into Mike's account"); New_Line;
  Deposit( Mike, 50.00 );
  Statement( Mike );

  Put("Withdraw £80.00 from Mike's account"); New_Line;
  Withdraw( Mike, 80.00, Obtained );
  Statement( Mike );

  Put("Deposit £500.00 into Corinna's account"); New_Line;
  Deposit( Corinna, 500.00 );
  Statement( Account(Corinna) );

  Put("Add interest to Corinna's account"); New_Line;
  Calc_Interest( Corinna );
  Add_Interest( Corinna );
  Statement( Account(Corinna) );
end Main;
```

In this program, the procedure `Statement` takes as its parameter an instance of an `Account`. However, as the account for Corinna is an instance of a `Interest_Account` it must first be converted to an instance of an `Account` before it can be passed as an actual parameter to `Statement`. This is accomplished with a view conversion `Account(Corinna)`. The actual parameter passed to `Statement` is now viewed as if it were an `Account`.

which, when compiled and run, would produce the following output:

```
Mini statement: The amount on deposit is £ 0.00

Deposit £50.00 into Mike's account
Mini statement: The amount on deposit is £50.00

Withdraw £80.00 from Mike's account
Mini statement: The amount on deposit is £50.00

Deposit £500.00 into Corinna's account
Mini statement: The amount on deposit is £500.00

Add interest to Corinna's account
Mini statement: The amount on deposit is £500.13
```

### 10.3.1 Terminology

The following terminology is used to describe the shared components of a class.

Terminology	Example: in class <i>Interest_account</i>	Explanation
Class attribute	The_Interest_Rate	A variable which is shared between all members of the class.
Class method	Set_Rate	A procedure or function used to access only class attributes.

### 10.4 Visibility rules (Normal inheritance)

The derived class can only access public methods of the base class. As instance attributes are declared in the private part of the base class they are not accessible to the derived class. For example, the class *Interest\_Account* can access the methods *Deposit*, *Withdraw*, and *Balance* but cannot access the instance attribute *Balance\_Of*. The visibility of items in the base class and derived class is illustrated in Figure 10.3.

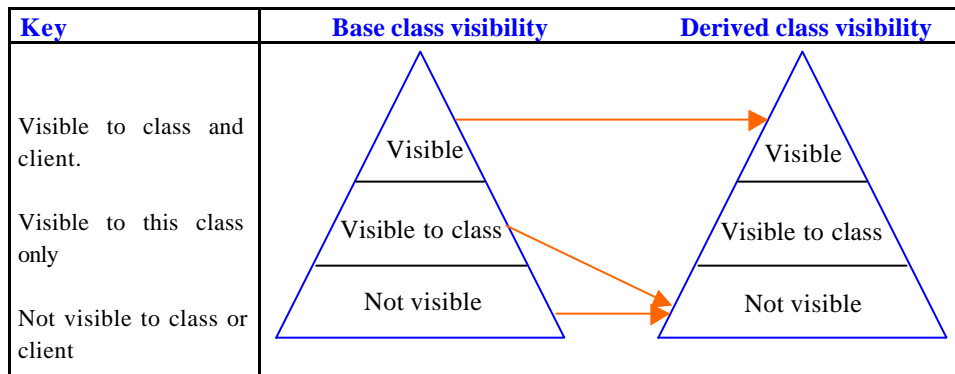


Figure 10.3 Visibility of components in base and derived classes.

### 10.5 Converting a derived class to a base class

A derived class may be converted to its base class, the effect of which is to remove the instance attributes added by the derived class. For example, in the following program Corinna's interest bearing account is converted to a normal account. However, a base class cannot be converted directly to a derived class.

```

with Class_Interest_Account, Class_Account, Statement;
use   Class_Interest_Account, Class_Account;
procedure Main is
  Corinna : Interest_Account;
  New_Acc : Account;
begin
  Deposit( Corinna, 100.00 );
  New_Acc := Account(Corinna);  --derived -> base conversion
  Statement( Account(Corinna) ); --Interest_account
  Statement( New_Acc );         --Account
end Main;

```

*Note:* The effect of a conversion from a derived class to a base class is to remove the additional components that have been defined in the derived class.

which when run, would give the following results:

```
Mini statement: The amount on deposit is £100.00
Mini statement: The amount on deposit is £100.00
```

## 10.6 Abstract class

If a class is to be used purely as a specification of the facilities that are to be provided by later derived classes, then it can be made abstract. An abstract class therefore has no implementation part. For example, an abstract specification of a bank account is as follows:

```
package Class_Abstract_Account is

  type Abstract_Account is abstract tagged null record;
  subtype Money is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Deposit ( The:in out Abstract_Account;
                      Amount:in Pmoney ) is abstract;
  procedure Withdraw ( The:in out Abstract_Account;
                      Amount:in Pmoney;
                      Get:out Pmoney ) is abstract;
  function Balance ( The:in Abstract_Account )
                   return Money is abstract;
end Class_Abstract_Account;
```

*Note: An elaboration of an abstract class can never be made.*

The keyword `abstract` is used to indicate:

- That the type is abstract, and hence no actual definition will be provided.
- That the methods (functions or procedures) are abstract and consequently there will be no implementation part.

*Note:* As the type `Abstract_account` contains no instance attributes it has been left public. It could have been defined as:

```
type Abstract_account is abstract tagged private;
```

*with the private part of the specification containing:*

```
type Abstract_account is abstract tagged null record;
```

*The component 'null record' is shorthand for :*

```
'record null end record'
```

## 154 Inheritance

The abstract class can then be used to derived specific types of bank account. In the following case it has been used to derive a simple bank account.

```
with Class_Abstract_Account;  
use Class_Abstract_Account;  
package Class_Account is  
  
    type Account is new Abstract_Account with private;  
    subtype Money is Class_Abstract_Account.Money;  
    subtype Pmoney is Class_Abstract_Account.Pmoney;  
    procedure Deposit ( The:in out Account; Amount:in Pmoney );  
    procedure Withdraw ( The:in out Account; Amount:in Pmoney;  
                        Get:out Pmoney );  
    function Balance ( The:in Account ) return Money;  
private  
    type Account is new Abstract_Account with record  
        Balance_Of : Money := 0.00;      --Amount in account  
    end record;  
end Class_Account;
```

*Note: Subtype has been used to make the subtypes Money and Pmoney visible to clients of the class. If this had not been done, users of the class would in most cases have to **with** and **use** the package Class\_abstract\_account.*

The implementation of which would be the same as the class Account shown in Section 6.3.5.

Once a class has been derived from an existing class it too may be used as a base class in deriving a new class. For example, an account that allows a customer to only make three withdrawals in a week can be derived from Class\_Account as follows:

```
with Class_Account;  
use Class_Account;  
package Class_Account_Ltd is  
  
    type Account_Ltd is new Account with private;  
  
    procedure Withdraw ( The:in out Account_Ltd;  
                        Amount:in Pmoney; Get:out Pmoney );  
    procedure Reset( The:in out Account_Ltd );  
private  
    Withdrawals_In_A_Week : Natural := 3;  
    type Account_Ltd is new Account with record  
        Withdrawals : Natural := Withdrawals_In_A_Week;  
    end record;  
end Class_Account_Ltd;
```

*Note: The derived class overloads the method Withdraw with a new specialized meaning. The method Reset is used to set the number of withdrawals that may be made in the current week to three.*

The implementation for the class is as follows:

```
package body Class_Account_Ltd is
```



The specialization of the procedure `Withdraw` calls the method `withdraw` in class `Account` to process the withdrawal. To avoid infinite recursion, the parameter `The` is converted to type `Account` before it is passed as a parameter to `Withdraw`. This is termed a view conversion. Overload resolution is then used to determine which version of `Withdraw` to call.

```

procedure Withdraw ( The:in out Account_Ltd;
    Amount:in Pmoney; Get:out Pmoney ) is
begin
    if The.Withdrawals > 0 then                --Not limit
        The.Withdrawals := The.Withdrawals - 1;
        Withdraw( Account(The), Amount, Get ); --In Account
    else
        Get := 0.00;                            --Sorry
    end if;
end Withdraw;

```

The function `reset` resets the number of withdrawals that may be made in the current week.

```

procedure Reset( The:in out Account_Ltd ) is
begin
    The.Withdrawals := Withdrawals_In_A_Week;
end Reset;

end Class_Account_Ltd;

```

### 10.6.1 Putting it all together

A program to illustrate the use of the class `Class_Account_ltd` is shown below:

```

with Class_Account, Class_Account_ltd, Statement;
use Class_Account, Class_Account_ltd;
procedure Main is
    Mike : Account_Ltd;
    Obtain: Money;
begin
    Deposit( Mike, 300.00 );                --In credit
    Statement( Account(Mike) );
    Withdraw( Mike, 100.00, Obtain ); --Withdraw some money
    Withdraw( Mike, 10.00, Obtain ); --Withdraw some money
    Withdraw( Mike, 10.00, Obtain ); --Withdraw some money
    Withdraw( Mike, 20.00, Obtain ); --Withdraw some money
    Statement( Account(Mike) );
end Main;

```

*Note: The **with** and **use** of the package `Class_Account` so that the subtype `Money` is directly visible. The procedure `Statement` seen earlier is used to print a mini statement.*

which when run, produces the following output:

```

Mini statement: The amount on deposit is £300.00
Mini statement: The amount on deposit is £180.00

```

*Note: The final withdrawal of £20 is not processed as three withdrawals have already been made this week.*

## 156 Inheritance

### 10.6.2 Visibility of base class methods

It is of course possible to call the base class method `withdraw` directly using:

```
Withdraw( Account(Mike), 20.00, Obtain ); --Cheat
```

as the Class `Account` is visible. This could have been prevented by not **with**'ing and **use**'ing the package `Class_Account`. This had been done to make the subtype `Money` directly visible. Remember `Money` is defined in the class `Account` (it has not been made visible to class `Account_Ltd`) so that both classes `Account` and `Account_Ltd` can be **with**'ed and **use**'ed in the same unit.

To avoid the possibility of the accidental use of `withdraw` in the base class the above program could have been written as:

```
with Class_Account, Class_Account_Ltd, Statement;
use Class_Account;
procedure Main is
  Mike : Account_Ltd;
  Obtain: Class_Account.Money;
begin
  Deposit( Mike, 300.00 );           --In credit
  Statement( Account(Mike) );
  Withdraw( Mike, 100.00, Obtain ); --Withdraw some money
  Withdraw( Mike, 10.00, Obtain ); --Withdraw some money
  Withdraw( Mike, 10.00, Obtain ); --Withdraw some money
  Withdraw( Mike, 20.00, Obtain ); --Withdraw some money
  Statement( Account(Mike) );
end Main;
```

*Note: The package `Class_Account` has been used explicitly to access the subtype `Money`.*

### 10.7 Multiple inheritance

Multiple inheritance is the ability to create a new class by inheriting from two or more base classes. For example, the class `Named_Account`, a named bank account, can be created from the class `Account` and the class `Name_Address`. The class `Account` is described in Section 6.3.4 and the class `Name_Address` is described in Section 8.10 The inheritance diagram for the class `Named_Account` is illustrated in Figure 10.4.

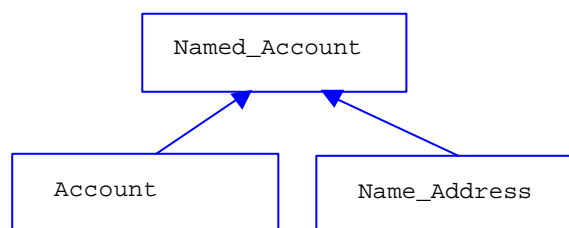


Figure 10.4 Inheritance diagram for a named bank account.

The responsibilities of the class `Named_Account` are all those of the classes `Account` and `Name_Address` plus the additional responsibility to print the person's name with their printed statement. The new responsibility of printing a statement with the account holder's name overrides the responsibility of printing a statement in `Account`. The methods in the three classes are:

In Class <code>Named_Account</code>	In Class <code>Account</code>	In Class <code>Name_Address</code>
	<code>Deposit</code>	<code>Set</code>
	<code>Withdraw</code>	<code>Print_Name</code>
	<code>Balance</code>	<code>Print_Address</code>

*Note: The methods of the class `Named_Account` will be all the methods of an `Account` plus all the methods of `Name_Address` plus any methods of `Named_Account` itself.*

Unfortunately multiple inheritance is not directly supported in Ada 95. However, an easy work around is to define the class `Named_Account` whose instance attributes are instances of the classes `Account` and `Name_Address`. The public methods of the class `Named_Account` have the same specification as the combined methods of `Account` and `Named_Account` except that the type of object operated on is an instance of the class `Named_Account`.

The specification for the class `Named_Account` is shown below:

```
with Class_Account, Class_Name_Address;
use Class_Account, Class_Name_Address;
package Class_Named_Account is

  type Named_Account is tagged private;
  subtype Pmoney is Class_Account.Pmoney;
  subtype Money is Class_Account.Money;

  procedure Set( The:out Named_Account; Str:in String );
  function Deliver_Line( The:in Named_Account;
                        Line:in Positive ) return String;
  function Lines( The:in Named_Account ) return Positive;
  procedure Deposit( The:in out Named_Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Named_Account; Amount:in Pmoney;
                     Get:out Pmoney );
  function Balance( The:in Named_Account ) return Pmoney;
private
  type Named_Account is tagged record
    Acc : Account;           --An account object
    Naa : Name_Address;      --A Name and address object
  end record;
end Class_Named_Account;
```

*Note: To allow a client of the class `Named_Account` to directly use the subtype `Money` in the class `Account` the declaration :*

***subtype Money is Class\_Account.Money;***

*has been added to the class.*

The implementation of the class `Named_Account` is split into three distinct parts:

- The implementation of the methods in `Name_Address`.
- The implementation of the methods in `Account`.
- The implementation of the methods of `Named_Account` itself.

The implementation of the methods in `Named_Account` which are inherited from `Name_Address` are as follows:

## 158 Inheritance

```
with Ada.Text_IO, Ada.Float_Text_IO;
use   Ada.Text_IO, Ada.Float_Text_IO;
package body Class_Named_Account is

  procedure Set( The:out Named_Account; Str:in String ) is
  begin
    Set( The.Naa, Str );
  end Set;

  function Deliver_Line( The:in Named_Account;
    Line:in Positive ) return String is
  begin
    return Deliver_Line( The.Naa, Line );
  end Deliver_Line;

  function Lines( The:in Named_Account ) return Positive is
  begin
    return Lines( The.Naa );
  end Lines;
```

The implementation of the methods in Named\_Account which are inherited from Account are as follows:

```
procedure Deposit(The:in out Named_Account; Amount:in Pmoney) is
begin
  Deposit( The.Acc, Amount );
end Deposit;

procedure Withdraw( The:in out Named_Account; Amount:in Pmoney;
  Get:out Pmoney ) is
begin
  Withdraw( The.Acc, Amount, Get );
end Withdraw;

function Balance ( The:in Named_Account ) return Pmoney is
begin
  return Balance( The.Acc );
end Balance;
end Class_Named_Account;
```

A procedure Statement to print a mini statement of the state of an instance of a Named\_Account is defined as follows:

```
with Ada.Text_IO, Ada.Float_Text_IO, Class_Named_Account;
use   Ada.Text_IO, Ada.Float_Text_IO, Class_Named_Account;
procedure Statement( An_Account:in Named_Account ) is
begin
  Put("Statement for : " ); Put( Deliver_Line(An_Account, 1)); New_Line;
  Put("Mini statement: The amount on deposit is f" );
  Put( Balance( An_Account), Aft=>2, Exp=>0 );
  New_Line(2);
end Statement;
```

*Note: Again by not permitting an input or output operations to be part of the class, the scope for its potential re-use is increased.*

### 10.7.1 Putting it all together

A program to illustrate the use of the class `Named_Account` is shown below:

```
with Class_Named_Account, Statement;
use Class_Named_Account;
procedure Main is
  Mike : Named_Account;
  Get   : Money;
begin
  Set      ( Mike, "A.N.Other/Brighton/UK" );
  Deposit ( Mike, 10.00 );
  Statement( Mike );
  Withdraw ( Mike, 5.00, Get );
  Statement( Mike );
end Main;
```

which when run, produces the following output:

```
Statement for : A.N.Other
Mini statement: The amount on deposit is £10.00

Statement for : A.N.Other
Mini statement: The amount on deposit is £ 5.00
```

## 10.8 Initialization and finalization

When an instance of a class is created, there can be static initialization of the instance or class attributes in the object. For example, in the class `Account` in Section 6.3.4 the initial amount in the account was set to £0.00 when an instance of the class was elaborated. However, this initialization is limited to a simple assignment. In some cases a more complex initialization is required.

Consider the case of a bank account that records an audit trail of all transactions made on the account. The audit trail consists of a record written to disk for each transaction made on the account. The audit trail file descriptor is shared between all instances of the class and is initialized by the first elaboration of an instance of the class. The file descriptor is closed when the last instance of the class is finalized.

By inheriting from the package `Ada.Finalization` user defined initialization and finalization can be defined for a class. This takes the form of two user defined procedures `Initialize` and `Finalize` which are called respectively when an instance of the class is elaborated and when an instance of the class is destroyed. For example, the point when initialization and finalization take place is annotated on the following fragment of code.

```
procedure Ex5 is
  Mike : Account_At;           --Initialization on mike
begin
  Deposit( Mike, 100.00 );
  declare
    Corinna : Account_At;      --Initialization on corinna
  begin
    Deposit( Corinna, 100.00 ); --Finalization on corinna
  end;
                                --Finalization on mike
end Ex5;
```

## 160 Inheritance

The responsibilities for the Class `Account_At` that provides an audit trail are as follows:

Method	Responsibility
Withdraw	Withdraw money from the account and write an audit trail record.
Deposit	Deposit money into the account and write an audit trail record.
Balance	Return the amount in the account and write an audit trail record.
Initialize	If this is the only active instance of the class <code>Account</code> then open the audit trail file.
Finalization	If this is the last active instance of the class <code>Account</code> then close the audit trail file.

The specification for the class `Account_At` that creates an audit trail of all transactions is:

```
with Ada.Text_Io, Ada.Finalization;
use   Ada.Finalization;
package Class_Account_At is

    type Account_At is new Limited_Controlled with private;
    subtype Money    is Float;
    subtype Pmoney   is Float range 0.0 .. Float'Last;

    procedure Initialize( The:in out Account_At );
    procedure Finalize  ( The:in out Account_At );

    procedure Deposit( The:in out Account_At; Amount:in Pmoney );
    procedure Withdraw( The:in out Account_At;
                        Amount:in Pmoney; Get:out Pmoney );
    function  Balance( The:Account_At ) return Money;
private
```

```
    type Account_At is new Limited_Controlled with record
        Balance_Of : Money    := 0.00;      --Amount on deposit
        Number      : Natural := 0;
    end record;
    The_Audit_Trail: Ada.Text_Io.File_Type; --File handle
    The_Active      : Natural := 0;         --No of accounts
end Class_Account_At;
```

*Note: Account\_At is inherited from Limited\_Controlled that is defined as a limited type. Thus assignments of instances of Account\_At are not allowed.*

The variables `The_Audit_Trail` and `The_Active` are shared amongst all instances of the class. These variables contain respectively the open file descriptor and the number of active instances of the class.

In a program using the class `Class_Account_At`, the number of active instances of the class are required so that the file descriptor `The_Audit_Trail` may be initialized when the first instance of the class is elaborated and closed when the last active instance of the class goes out of scope.

### 10.8.1 Implementation

The procedure `Initialize` is called whenever an instance of `Account_At` is elaborated. This procedure checks if this is the first elaboration, determined by the reference count `The_Active`. If it is the first concurrent elaboration then the audit trail file `log.txt` is opened in append mode and associated with the file descriptor `The_Audit_Trail`.

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use   Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
package body Class_Account_At is

  procedure Initialize( The:in out Account_At ) is
  begin
    The_Active := The_Active + 1;      --Another object
    if The_Active = 1 then            -- first time for class
      Open( File=>The_Audit_Trail,
            Mode=>Append_File, Name=>"log.txt" );
    end if;
  end Initialize;
```

The procedure `Finalize` is called when the elaborated storage for an instance of `Account_At` goes out of scope. The reference count of the number of active instantiations of `Account_At` is checked and when the last concurrent instance goes out of scope the file descriptor `The_Audit_Trail` is closed.

```
procedure Finalize( The:in out Account_At ) is
begin
  if The_Active = 1 then Close( The_Audit_Trail ); end if;
  The_Active:=The_Active-1;
end Finalize;
```

The rest of the implementation follows very closely the previous implementation of the class `Account` with the additional functionality of writing the audit trail record for each transaction performed.

```
procedure Deposit( The:in out Account_At; Amount:in Pmoney ) is
begin
  The.Balance_Of := The.Balance_Of + Amount;
  Audit_Trail( The, " Deposit   : Amount = ", Amount );
end Deposit;
```

```
procedure Withdraw( The:in out Account_At;
                   Amount:in Pmoney; Get:out Pmoney ) is
begin
  if The.Balance_Of >= Amount then
    The.Balance_Of := The.Balance_Of - Amount;
    Get := Amount;
  else
    Get := 0.00;
  end if;
  Audit_Trail( The, " Withdraw  : Amount = ", Get );
end Withdraw;

function Balance( The:in Account_At ) return Money is
begin
  Audit_Trail( The, " Balance   : Balance = ", The.Balance_Of );
  return The.Balance_Of;
end Balance;

end Class_Account_At;
```

## 162 Inheritance

### 10.8.2 Putting it all together

The following short test program demonstrates the working of the class `Account_At`.

```
with Class_Account_At, Statement;
use Class_Account_At;
procedure Main is
  Bank : array ( 1 .. 10 ) of Account_At;
  Obtain: Money;
begin
  Deposit( Bank(1), 100.00 );           --Deposit 100.00
  Withdraw( Bank(1), 80.00, Obtain );  --Withdraw 80.00
  Deposit( Bank(2), 200.00 );           --Deposit 200.00
end Main;
```

*Note: The procedure Initialize will be called ten times when the object Bank is elaborated, once for each element of the array. Likewise when the object Bank is finalized, the procedure Finalize will be called ten times, once for each element of the array.*

This when compiled and run will generate the file `log.txt` which contains the audit trail of all transactions made. The contents of the audit trail file are illustrated below:

```
Deposit : 100.00
Withdraw : 80.00
Deposit : 200.00
```

### 10.8.3 Warning

There are two base types in `Ada.Finalization` from which user defined initialization and finalization is facilitated. These are `Controlled` and `Limited_Controlled` the properties of which are:

Type in <code>Ada.Finalization</code>	Properties
<code>Controlled</code>	Allow user defined initialization and finalization for inheriting types. Instances of these types may be assigned.
<code>Limited_Controlled</code>	Allow user defined initialization and finalization for inheriting types. Instances of these types may not be assigned.

When the base type for a class is `Controlled` then as part of an assignment operation `Finalization` is called on the target of the assignment. This will result in `Finalization` being called at least twice on an object. The procedure `Finalization` is called once when an object is assigned too and once when its storage is de-allocated. Thus if you use `Controlled` as the base type, the code for `Finalization` must allow for such an eventuality. The code for `Finalization` in the class `Account_At` cannot be called twice. The exact details of how to use `Controlled` are explained in Chapter 17.



## 10.9 Hiding the base class methods

The base class methods may be hidden in a class by defining the inheritance only in the private part of the specification. For example, a restricted type of account that only allows a statement to be printed and money to be deposited into the account can be created.

For this type of account we wish to prevent the user from calling base class methods. Remember that normally with inheritance the base class members are visible. Even if a base class method is overloaded in the derived class it can still be called.

The specification of the class `Restricted_Account` is:

```
with Class_Account;
use Class_Account;
package Class_Restricted_Account is

    type Restricted_Account is private;
    subtype Money is Class_Account.Money;
    subtype Pmoney is Class_Account.Pmoney;

    procedure Deposit( The:in out Restricted_Account;
                      Amount:in Pmoney );
private
    type Restricted_Account is new Account with record
        null;
    end record;
end Class_Restricted_Account;
```

*Note: The use of subtype to make the types from class Account visible.*

The implementation of the class is:

```
package body Class_Restricted_Account is

    procedure Deposit( The:in out Restricted_Account;
                      Amount:in Pmoney ) is
    begin
        Deposit( Account(The), Amount );
    end Deposit;

end Class_Restricted_Account;
```

Here the base class methods are called from within the body of the derived class methods. Remember the body of the package can see the methods of the base class.

## 164 Inheritance

### 10.9.1 Visibility rules (Hidden base class)

The visibility of items in the base class and derived class is illustrated in Figure 10.5.

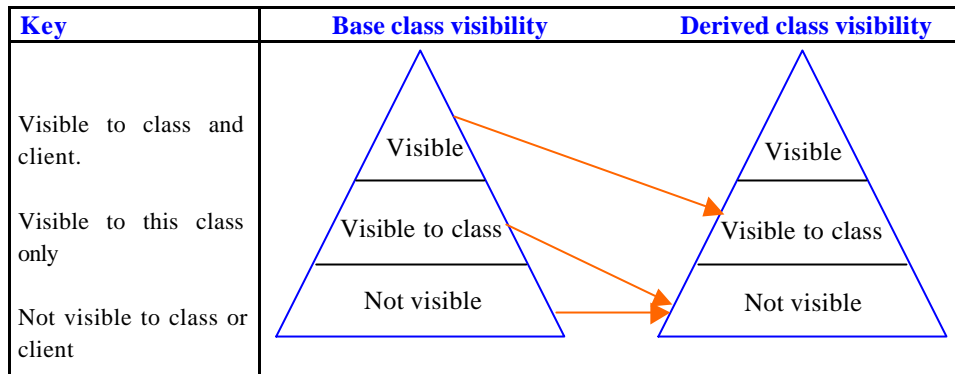


Figure 10.5 Visibility of components in base and derived classes.

The consequence of this is that any methods from the base class that a client of the derived class may wish to use have to be explicitly made available by providing an appropriate procedure or function in the derived class.

### 10.9.2 Putting it all together

The program below illustrates the use of this restricted account.

```
with Statement;  
procedure Main is  
  Corinna :Restricted_Account;  --Can only deposit  
begin  
  Statement( Corinna );  
  Deposit( Corinna, 50.00 );  
  Statement( Corinna );  
end Ex1;
```

which when compiled with a suitable definition for the procedure `Statement` would produce output of the form::

```
Mini statement: The amount on deposit is £ 0.00  
Mini statement: The amount on deposit is £50.00
```

## 10.10 Self-assessment

- How can the use of inheritance save time in the production of software?
- Can any previously defined class be used as a base class from which other classes are derived?
- Can a derived class see the private data attributes of the base class? Explain why this is so.
- What is the purpose of the pre-defined package `Ada.Finalization`?

- For an object *o* which is an instance of a derived class, how does a programmer call the method *m* in the base class which has been overloaded by another method *m* in the derived class?
- Why can the code of Finalization in the class *Account\_At* not be called twice?

## 10.11 Exercises

Construct the following:

- *Employee\_Pay*  
A class *Employee\_Pay* which represents a person's salary has the following methods:

Method	Responsibility
Set_hourly_rate	Set the hourly rate.
Add_hours_worked	Accumulate the number of hours worked so far.
Pay	Deliver the pay for this week.
Reset	Reset the hours worked back to zero.
Hours_Worked	Deliver the number of hours worked so far this week.
Pay_rate	Deliver the hourly pay rate.

Tax is to be deducted at 20% of total pay.

- *Test*  
A program to test the class *Employee\_Pay*.
- *Better\_Employee\_Pay*  
A class *Better\_Employee\_Pay* which represents a person's salary. This extends the class *Employee\_Pay* to add the additional methods of:

Method	Responsibility
Set_Overtime_Pay	Set the overtime pay rate.
Normal_Pay_Hours	Set the number of hours in a week that have to be worked before the overtime pay rate is applied.
Pay	Deliver the pay for this week. This will consist of the hours worked at the normal pay rate plus the hours worked at the overtime rate.

- *Test*  
A program to test the class *Better\_employee\_pay*.
- *Employee\_Pay\_With\_Repayment*.  
A class *Employee\_Pay\_With\_Repayment* which represents a person's salary after the deduction of the weekly repayment of part of a loan for travel expenses. This extends the class *Better\_employee\_pay* to add the additional methods of:

Method	Responsibility
Set_Deduction	Set the weekly deduction
Pay	Deliver the pay for this week. This will include the deduction of the money for the employee loan if possible.

Remember to include the possibility of an employee not being able to repay the weekly repayment of their loan as they have not worked enough hours.

- *test*  
A program to test the class *Employee\_Pay\_With\_Repayment*.

# 11 Child libraries

This chapter introduces child libraries. A child library is a way of adding to an existing package without changing the original package. In addition, a child of an existing package is allowed to access the private components of the parent. By using child libraries a large package or class can be split into manageable components.

## 11.1 Introduction

In developing software, extensions to an existing class are occasionally required which modify or access the private instance attributes. This is not possible with inheritance as an inheriting class is not allowed to access private instance attributes of the base class. Rather than change the code of the class directly, a child library of the class can be created which is a separate entity that is allowed to access private components of a package.

The original class is not re-compiled, and thus does not need re-testing. However, the combined parent and child library needs to be tested as the child library can modify private instance or class attributes of the parent.

This is similar in effect to inheritance, in that new methods are added to an existing class. The class type however, may not be extended. For example, the class `Interest_account` (in Section 10.3) can have an additional method `inspect_interest` added which will allow inspection of the accumulating interest that is to be added to the account at the end of the accounting period. This is implemented as a child package whose specification is as follows:

```
package Class_Interest_Account.Inspect_Interest is
  function Interest_Is( The:in Interest_Account )
    return Money;
end Class_Interest_Account.Inspect_Interest;
```

The child package name is defined as two components, the original package name followed by the name of the child package. In this case the two components are `Class_Interest_Account.Inspect_Interest`.

The implementation of these is as follows:

```
package body Class_Interest_Account.Inspect_Interest is
  function Interest_Is( The:in Interest_Account )
    return Money is
  begin
    return The.Accumulated_Interest;
  end Interest_Is;
end Class_Interest_Account.Inspect_Interest;
```

*Note: You can access private components of the parent package.*

This specialization of an interest bearing account could not be created by inheriting from the class `Interest_account` as `accumulated_interest` is a private instance attribute of the class `Interest_account`. Access to this variable breaks the encapsulation of the class `Interest_account`.

The intent is for child packages to allow the specialization of an existing package without having to change the parent. In particular, the parent will not need re-testing but the combined parent and child must be tested as the child package may affect the working of the parent.

### 11.1.1 Putting it all together

The package `Interest_account` and its child `inspect_interest` are used as follows:

```
with Ada.Text_Io, Ada.Float_Text_Io, Class_Account,
      Class_Interest_Account, Class_Interest_Account.Inspect_Interest,
      Statement;
use   Ada.Text_Io, Ada.Float_Text_Io, Class_Account,
      Class_Interest_Account, Class_Interest_Account.Inspect_Interest;
procedure Main is
  My_Account: Interest_Account;
  Obtained  : Money;
begin
  Statement( My_Account );
  Put("Deposit 100.00 into account"); New_Line;
  Deposit( My_Account, 100.00 );      --Day 1
  Calc_Interest( My_Account );        --End of day 1
  Calc_Interest( My_Account );        --End of day 2
  Statement( My_Account );            --Day 3
  Obtained := Interest_Is( My_Account ); --How much interest
  Put("Interest accrued so far : £" );
  Put( Obtained, Aft=>2, Exp=>0 ); New_Line;
end Main;
```

*Note:* When a child is included, its ancestors are automatically **with**'ed.

When run it will produce the following output:

```
Mini statement: The amount on deposit is £ 0.00

Deposit 100.00 into account
Mini statement: The amount on deposit is £100.00

Interest accrued so far : £ 0.05
```

### 11.1.2 Warning

A child package breaks the encapsulation rules of a package. In particular a child package can access the private data components of its parent package. In accessing the private instance attributes of a class, the child package may compromise the integrity of the parent package.

## 11.2 Visibility rules of a child package

The private part of a child package can access all components of its ancestors, even the private components. However, the visible part of a child package has no access to the private components of its ancestors. This is to prevent possible renaming allowing a client direct access to the private components of one of the child's ancestors.

A child package allows a programmer the ability to extend an existing package without the need to change or re-compile the package.

For example, Figure 11.1 illustrates a hierarchy of package specifications rooted at package P. The ancestor of packages P.C2 and P.C1 is package P, whilst the ancestors of package P.C2.G1 are the packages P.C2 and P.

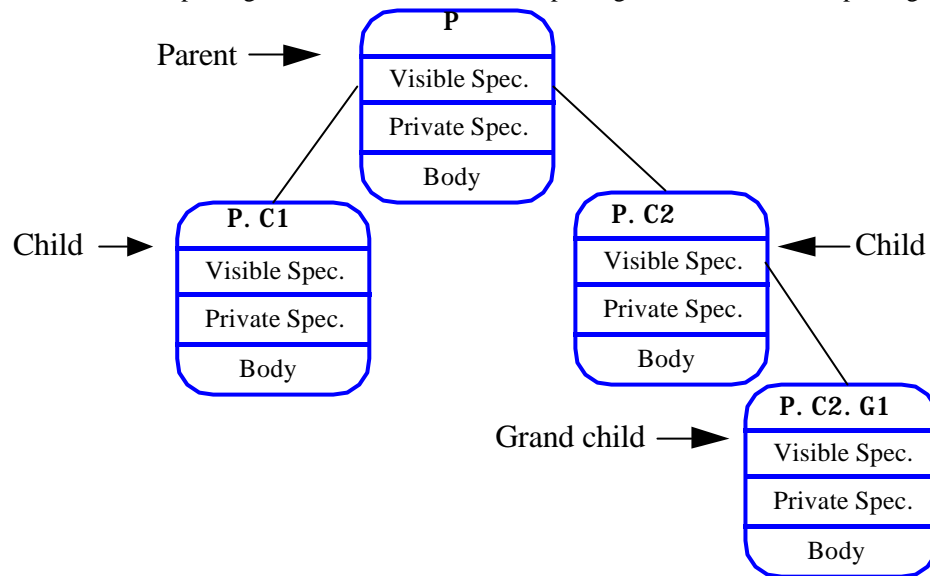


Figure 11.1 Illustration of the hierarchy of child units.

Key	Package specification	Package body
Components of a package in this case package P.	<pre> package P is   -- Visible specification. private   -- Private specification. end P;</pre>	<pre> package body P is   -- Body of package end P;</pre>

Can access in->	P	P.C1	P.C2	P.C2.G1
P.C1 Visible specification	Visible spec.	•	Visible spec. when <b>with</b> 'ed	Visible spec. when <b>with</b> 'ed
P.C1 Private specification	Visible spec. Private spec.	•	Visible spec. when <b>with</b> 'ed	Visible spec. when <b>with</b> 'ed
P.C2.G1 Visible specification	Visible spec.	Visible spec. when <b>with</b> 'ed	Visible spec.	•
P.C2.G1 Private specification	Visible spec. Private spec.	Visible spec. when <b>with</b> 'ed	Visible spec. Private spec.	•

Note: A **with** clause for a child package implies a **with** clause for all its ancestors.

## 11.3 Private child

A private child package is like a normal child package except that it is only visible within the sub-tree, which has an ancestor as its root. A private child can be used to hide implementation details from a user of the parent package. A private child package is specified by prefixing the reserved word package with the word private

### 11.3.1 Visibility rules of a private child package

If in Figure 11.1 P.C2 is a private child then the visibility of components is as illustrated in the table below.

Can access in ->	P	P.C2	P.C1
P.C2 Visible specification	Visible spec.	•	Visible spec. when <b>with</b> 'ed
P.C2 Private specification	Visible spec. Private spec.	•	Visible spec. when <b>with</b> 'ed
P.C1 Visible specification	Visible spec.	No access	•
P.C1 Private specification	Visible spec. Private spec.	No access	•

*Note:* The private part of a child package has access to all components of its parent, even the private components.

## 11.4 Child packages vs. inheritance

The following table summarizes the differences between the use of a child package and inheritance.

Ability to	Child package	Inheritance
Create a new package	✗	✓
Extend a base package by adding new procedures and functions	✓	✓
Extend a type in the base package	✗	✓ (see note)
Access private components in the base package	✓	✗
Override existing procedures and functions in the base package	✗	✓

*Note:* Must be tagged in the base class.

The danger in the use of child libraries is that they can subvert the data hiding of a class. For example, the class `Interest_account` hides the representation of, and prevents access to the `accumulated_interest`. A child library of the class `Interest_account` can allow a client of the class the ability to change or inspect this hidden variable.

## 11.5 Self-assessment

- How can the use of child libraries save time in the production of software?
- Why is a child library's public specification not allowed to access components in a parent's private specification?
- What is the difference between a normal child package and a private child package?
- What is the difference between the use of a child package and the use of inheritance to build on existing code?

## 11.6 Exercises

Construct the following:

- *Money*  
A class which manipulates amounts of money held in pounds and pence. This class should allow the following operations to be performed on an instance of the class `Money`
  - Add monetary amounts using the operator `+`.
  - Subtract monetary amounts using the operator `-`.
- *Conversion*  
A child library of the package `Class_money` which allows the conversion of an amount in pounds to dollars, francs and ECU (European Currency Unit).



# 12 Defining new operators

This chapter shows how the predefined operators in Ada can be overloaded with a new meaning.

## 12.1 Defining operators in Ada

The existing operators in Ada can be overloaded with a new meaning. These new operators have the same precedence as their existing counterparts. For example, to trace every executed integer + operation in a program, the operator + can be overloaded by a function that writes trace information to the terminal before delivering the normal integer addition. This is implemented by the following function:

```
function "+" ( F:in Integer; S:in Integer ) return Integer is
begin
  Put("[Performing "); Put(F, Width=>1 );
  Put(" + "); Put(S, Width=>1 ); Put("]");
  return Standard."+"( F, S );
end "+";
```

*Note:* To perform the inbuilt + the functional notation for the plus operation must be used. This is written as `Standard."+"( F, S )`. In Ada, the inbuilt operators are considered to belong to the package `Standard` which is automatically made visible to all program units.

Section C.4, Appendix C gives the specification for the package `standard`.

The above function can be used to trace the use of + in the following program:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  function "+" ( F:in Integer; S:in Integer ) return Integer is
  begin
    Put("[Performing "); Put(F, Width=>1 );
    Put(" + "); Put(S, Width=>1 ); Put("]");
    return Standard."+"( F, S );
  end "+";
begin
  Put("The sum of 1 + 2 is: "); Put ( 1+2 ); New_Line;
  Put("The sum of 1 + 2 is: ");
  Put( Standard."+"(1,2), Width=>1 ); New_Line;
  Put("The sum of 1 + 2 is: ");
  Put( "+"(1,2), Width=>1 ); New_Line;
end Main;
```

*Note:* As the package `Standard` is considered to be included with all program units. To achieve the effect of tracing each use of +, the overloaded function "+" has to be a nested function of the program unit.

```
The sum of 1 + 2 is: [Performing 1 + 2]3
The sum of 1 + 2 is: 3
The sum of 1 + 2 is: [Performing 1 + 2]3
```

*Note:* The way of directly using the operator + defined in the package `Standard` 'Standard."+"( 1, 2 )'.

The function notation for the use of the operator + "+"( 1, 2 ).

## 12.2 A rational arithmetic package

If precise arithmetic with rational numbers is required the Ada language can be extended by the inclusion of a package that provides a new type `Rational`. Instances of this type may be used as if they were normal numeric values such as integer.

The following extension to the language Ada is created by defining the class `Rational`. This class defines the following operations: `+`, `-`, `*` on an instance of a `Rational` number.

The responsibilities of this class are as follows::

Method	Responsibility
<code>+</code>	Delivers the sum of two rational numbers as a rational number.
<code>-</code>	Delivers the difference of two rational numbers as a rational number.
<code>*</code>	Delivers the product of two rational numbers as a rational number.
<code>/</code>	Delivers the division of two rational numbers as a rational number.

In addition the following methods are used to create a rational constant and to help output a rational number in a canonical form.

Method	Responsibility
<code>Rat_Const</code>	Creates a rational number from two <code>Integer</code> numbers.
<code>Image</code>	Returns a string image of a rational number in the canonical form 'a b/c'. For example: <pre>Put( Image( Rat_Const(3,2) ) );</pre> would print <pre>1 1/2</pre>

### 12.2.1 Ada specification of the package

The specification of the package is as follows:

```
package Class_Rational is
  type Rational is private;

  function "+" ( F:in Rational; S:in Rational ) return Rational;
  function "-" ( F:in Rational; S:in Rational ) return Rational;
  function "*" ( F:in Rational; S:in Rational ) return Rational;
  function "/" ( F:in Rational; S:in Rational ) return Rational;

  function Rat_Const( F:in Integer;
                     S:in Integer:=1 ) return Rational;
  function Image( The:in Rational ) return String;
private
  type Rational is record
    Above : Integer := 0;      --Numerator
    Below : Integer := 1;     --Denominator
  end record;
end Class_Rational;
```

Using the above package, the following code can be written:

```
with Ada.Text_IO, Class_Rational;
use  Ada.Text_IO, Class_Rational;
procedure Main is
  A,B : Rational;
begin
  A := Rat_Const( 1, 2 );
  B := Rat_Const( 1, 3 );

  Put( "a      = " ); Put( Image(A) ); New_Line;
  Put( "b      = " ); Put( Image(B) ); New_Line;
  Put( "a + b = " ); Put( Image(A+B) ); New_Line;
  Put( "a - b = " ); Put( Image(A-B) ); New_Line;
  Put( "b - a = " ); Put( Image(B-A) ); New_Line;
  Put( "a * b = " ); Put( Image(A*B) ); New_Line;
  Put( "a / b = " ); Put( Image(A/B) ); New_Line;
end Main;
```

which when run will deliver the following output:

```
a      = 1/2
b      = 1/3
a + b = 5/6
a - b = 1/6
b - a = -1/6
a * b = 1/6
a / b = 1 1/2
```

### 12.2.2 Ada implementation of the package

The internal function `sign` makes sure that only the top part of the rational number may be negative. By ensuring this form, the processing of rational numbers is simplified in later code.

```
package body Class_Rational is

function Sign( The:in Rational ) return Rational is
begin
  if The.Below >= 0 then          -- -a/b or a/b
    return The;
  else                            -- a/-b or -a/-b
    return Rational'( -The.Above, -The.Below );
  end if;
end Sign;
```

## 174 Child libraries

The internal function `simplify` reduces the rational number to its simplest form. Thus the rational number  $4/8$  is reduced to  $1/2$ .

```
function Simplify( The:in Rational ) return Rational is
  Res: Rational := The;
  D  : Positive;                                --Divisor to reduce with
begin
  if Res.Below = 0 then                          --Invalid treat as 0
    Res.Above := 0; Res.Below := 1;
  end if;
  D := 2;                                         --Divide by 2, 3, 4 ...
  while D < Res.Below loop
    while Res.Below rem D = 0 and then Res.Above rem D = 0 loop
      Res.Above := Res.Above / D;
      Res.Below := Res.Below / D;
    end loop;
    D := D + 1;
  end loop;
  return Res;
end Simplify;
```

*Note:* It is left to the reader to improve the efficiency of the algorithm used.

The standard operators of  $+$ ,  $-$ ,  $/$ , and  $*$  are overloaded to allow these standard operations to be performed between instances of rational numbers.

```
function "+" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Below := F.Below * S.Below;
  Res.Above := F.Above * S.Below + S.Above * F.Below;
  return Simplify(Res);
end "+";

function "-" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Below := F.Below * S.Below;
  Res.Above := F.Above * S.Below - S.Above * F.Below;
  return Simplify(Res);
end "-";

function "*" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Above := F.Above * S.Above;
  Res.Below := F.Below * S.Below;
  return Simplify(Res);
end "*";

function "/" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Above := F.Above * S.Below;
  Res.Below := F.Below * S.Above;
  return Simplify(Res);
end "/";
```

*Note:* Additional definitions of these standard operators would need to be provided if it was required to be able to perform operations such as `Rat_Const(1,2) + 1`.  
In this particular case a definition of  $+$  between a rational and an integer would also need to be provided.

The function `Rat_Const` is used to construct a constant rational number. The second formal parameter may be omitted when converting a whole number into a rational number.

```
function Rat_Const( F:in Integer;  
                  S:in Integer:=1 ) return Rational is  
begin  
  if F = 0 then  
    return Rational'(0,1);  
  else  
    return Simplify( Sign( Rational'( F, S ) ) );  
  end if;  
end Rat_Const;
```

*Note: A rational constant could have been created by overloading the operator / between two integers to deliver a rational number. The disadvantage of this approach is that the two distinct meanings for / must be distinguished between in a program section.*

The function `Image` returns a string representing a rational number in canonical form. The strategy used is to use the inbuilt function `Integer'Image` to convert a number into a character string. However, as this leaves a leading space for the sign character an internal function `Trim` is provided to strip off the leading character from such a string.

The nested function `To_String` delivers a string representation in canonical form of a positive rational number. By using a single case of recursion this function can deal with the case when a rational number is of the form "a b/c".

```
function Image( The:in Rational ) return String is
  Above : Integer := The.Above;
  Below : Integer := The.Below;

  function Trim( Str:in String ) return String is
  begin
    return Str( Str'First+1 .. Str'Last );
  end Trim;

  function To_String( Above, Below : in Integer )
    return String is
  begin
    if Above = 0 then                --No fraction
      return "";
    elsif Above >= Below then        --Whole number
      return Trim( Integer'Image(Above/Below) ) & " " &
        To_String( Above rem below, Below );
    else
      return Trim( Integer'Image( Above ) ) & "/" &
        Trim( Integer'Image( Below ) );
    end if;
  end To_String;

begin
  if Above = 0 then
    return "0";                      --Zero
  elsif Above < 0 then
    return "-" & To_String( abs Above, Below ); -- -ve
  else
    return To_String( Above, Below );  --+ve
  end if;
end Image;
end Class_Rational;
```

### 12.3 A bounded string class

A partial solution to overcome the fixed size limitations of Ada strings is to use a discriminated record that can hold a string of any length up to a pre-defined maximum. The responsibilities of the class `Bounded_String` which holds a variable length string is as follows:

Method	Responsibility
Operator: &	Concatenate an Ada string or a <code>Bounded_String</code> to a <code>Bounded_string</code> .
Operators : > >= < <= =	Compare instances of <code>Bounded_string</code> .
<code>To_String</code>	Convert an instance of a <code>Bounded_String</code> to an Ada string.
<code>To_Bounded_String</code>	Convert an Ada string to an instance of a <code>Bounded_string</code> .
<code>Slice</code>	Deliver a slice of a <code>Bounded_string</code> .

### 12.3.1 Overloading = and /=

The operators = and /= are provided automatically by the Ada system for comparing for equality or not equality. However, if a user redefines the = operator with a function that returns a Boolean value, the Ada system automatically provides the definition of /= as simply **not** =.

If the operator = is overloaded by a function that returns a value other than a Boolean, then the user must explicitly provide an overload definition for /= if it is to be used.

### 12.3.2 Specification of the class Bounded\_String

The Ada specification of the class Bounded\_String is shown below:

```
package Class_Bounded_String is
  type Bounded_String is private;

  function To_Bounded_String(Str:in String)
    return Bounded_String;

  function To_String(The:in Bounded_String) return String;

  function "&" (F:in Bounded_String; S:in Bounded_String)
    return Bounded_String;
  function "&" (F:in Bounded_String; S:in String)
    return Bounded_String;
  function "&" (F:in String; S:in Bounded_String)
    return Bounded_String;

  function Slice( The:in Bounded_String;
                  Low:in Positive; High:in Natural )
    return String;

  function "=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;

  function ">" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
  function ">=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
  function "<" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
  function "<=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;

private
  Max_String: constant := 80;
  subtype Str_Range is Natural range 0 .. Max_String;
  type A_Bounded_String( Length: Str_Range := 0 ) is record
    Chrs: String( 1 .. Length ); --Stored string
  end record;
  type Bounded_String is record
    V_Str : A_Bounded_String;
  end record;
end Class_Bounded_String;
```

In the specification of the class Bounded\_String a discriminated record is used. This discriminated record A\_Bounded\_String will store strings up to length MAX\_STRING characters. The discriminate length is used to specify the upper bound of the string, and has a default value of 0. An instance of Bounded\_String may be assigned another instance of Bounded\_String that may have a different discriminate value.

## 178 Child libraries

*Note: The discriminated record will usually be implemented by allocating the maximum amount of storage. Setting MAX\_STRING to 10\_000 in the package would allow for most eventualities, but would waste large amounts of storage.  
As the operator = is overloaded by a function which returns a Boolean value then the operator /= is automatically created.*

In the implementation of the package Class\_Bounded\_String shown below, the procedure To\_Bounded\_String is used to convert a 'normal' Ada string into an instance of a Bounded\_string.

```
package body Class_Bounded_String is

  function To_Bounded_String( Str:in String )
    return Bounded_String is
  begin
    return (V_Str=>(Str'Length, Str));
  end To_Bounded_String;
```

The function To\_String delivers a normal Ada string from a Bounded\_String.

```
function To_String(The:in Bounded_String) return String is
begin
  return The.V_Str.Chrs( 1 .. The.V_Str.Length );
end To_String;
```

The function Slice allows slices to be taken off an instance of a Bounded\_String.

```
function Slice( The:in Bounded_String;
                Low:in Positive; High:in Natural)
  return String is
begin
  if Low <= High and then High <= The.V_Str.Length then
    return The.V_Str.Chrs( Low .. High );
  end if;
  return " ";
end Slice;
```



The overloaded definitions of &, >, >=, <, <= allow the normal Ada comparison operators to be used with instances of a Bounded\_String. The operators for = are used by the Ada system to provide the definition of /=. The implementation for & allows concatenation between instances of a Bounded\_String and a normal Ada string. This is achieved by overloading & with three different definitions as follows:

```
function "&" ( F:in Bounded_String; S:in Bounded_String )
  return Bounded_String is
begin
  return (V_Str=>(F.V_Str.Chrs'Length + S.V_Str.Chrs'Length,
    F.V_Str.Chrs & S.V_Str.Chrs));
end "&";

function "&" ( F:in Bounded_String; S:in String )
  return Bounded_String is
begin
  return (V_Str=>(F.V_Str.Chrs'Length + S'Length,
    F.V_Str.Chrs & S ) );
end "&";

function "&" ( F:in String; S:in Bounded_String )
  return Bounded_String is
begin
  return ( V_Str=>(F'Length + S.V_Str.Chrs'Length,
    F & S.V_Str.Chrs ) );
end "&";
```

The implementation for the relational operators however, only allows comparison between instances of a Bounded\_String. Their implementation is as follows:

```
function ">" ( F:in Bounded_String; S:in Bounded_String )
  return Boolean is
begin
  return F.V_Str.Chrs > S.V_Str.Chrs;
end ">";

function ">=" ( F:in Bounded_String; S:in Bounded_String )
  return Boolean is
begin
  return F.V_Str.Chrs >= S.V_Str.Chrs;
end ">=";

function "<" ( F:in Bounded_String; S:in Bounded_String )
  return Boolean is
begin
  return F.V_Str.Chrs < S.V_Str.Chrs;
end "<";

function "<=" ( F:in Bounded_String; S:in Bounded_String )
  return Boolean is
begin
  return F.V_Str.Chrs <= S.V_Str.Chrs;
end "<=";
```

## 180 *Child libraries*

The implementation of = is as follows:

```
function "=" ( F:in Bounded_String; S:in Bounded_String )
  return Boolean is
begin
  return F.V_Str.Chrs = S.V_Str.Chrs;
end "=";

end Class_bounded_string;
```

*Note:* To compare an instance of a `Bounded_String` and an instance of an Ada string a user would have to convert the Ada string to a `Bounded_String`. For example:

```
Name : Bounded_String;
if Name > To_Bounded_String( "Brighton" ) then
```

### 12.3.3 Putting it all together

```
procedure Main is
  Town, County, Address : Bounded_String;
begin
  Town := To_Bounded_String( "Brighton" );
  County := To_Bounded_String( "East Sussex" );

  Address := Town & " " & County;

  Put( To_String(Address) ); New_Line;
  Put( Slice( County & " UK", 6, 14 ) );
  New_Line;
end Main;
```

When run, this would produce the following results:

```
Brighton East Sussex
Sussex UK
```

### 12.3.4 `Ada.Strings.Bounded` a standard library

In the standard library there is a package `Ada.Strings.Bounded` which the above class `Bounded_String` is based on. The generic library package `Ada.Strings.Bounded.Generic_bounded_length` allows the maximum length of the stored string to be defined by a user of the package. Chapter 13 describes the concepts of generics. Appendix C.8 lists the specification of the library package `Ada.Strings.Bounded`.

### 12.3.5 use type

A modified form of the **use** clause allows operators from a package to be used without having to prefix the operator with the package name. Other components however, from the package need to be prefixed with the package name when used. This modified form of the use clause is **use type** which is followed by the type name whose operators can be used without prefixing them by the package name. For example, the following program requires all components in the package `Bounded_String` except for operators to be prefixed with the package name.

```
with Ada.Text_IO, Class_Bounded_String;
use type Class_Bounded_String.Bounded_String;
procedure Main is
  Town   : Class_Bounded_String.Bounded_String :=
    Class_Bounded_String.To_Bounded_String("Brighton");
  County: Class_Bounded_String.Bounded_String :=
    Class_Bounded_String.To_Bounded_String("E Sussex");
begin
  Ada.Text_IO.Put(
    Class_Bounded_String.To_String( Town & " " & County )
  );
end Main;
```

## 12.4 Self-assessment

- What operators can be overloaded with a new meaning in Ada?
- Can a user invent new operators? For example, could a user define the monadic operator ++ to add one to an integer?
- Why might excessive use of overloading the standard operators lead to a program that is difficult to follow?
- Why is the function `rat_const` needed in the class `Rational`?
- How can a user guarantee to use the definition for the operator + in the package standard?

## 12.5 Exercises

Construct the following class:

## 182 Child libraries

- A very large integer number class .

which stores an integer number to 200 digits.

Method	Responsibility
+	Delivers the sum of two very long integer numbers as a very long integer number.
-	Delivers the difference between two very long integer numbers as a very long integer number.
VLN_const	Creates a very long integer number from an Integer.
Image	Return a string representation of a very large number.

A user of the class Class\_Very\_Large\_Number can write:

```
with Ada.Text_IO, Class_Very_Large_Number;
use Ada.Text_IO, Class_Very_Large_Number;
procedure Main is
  Max_Fibonacci : constant := 50;
  type Fibonacci_Index is range 1 .. Max_Fibonacci;
  type Fibonacci_Array is array ( Fibonacci_Index )
    of VNL;
  Fibonacci_Numbers: Fibonacci_Array;
begin
  Fibonacci_Numbers(1) := VLN_Const(1);
  Fibonacci_Numbers(2) := VLN_Const(1);
  for I in Fibonacci_Index range 3 .. Max_Fibonacci loop
    Fibonacci_Numbers( I ) := Fibonacci_Numbers( I-1 ) +
      Fibonacci_Numbers( I-2 );
  end loop;

  Put("First "); Put(Max_Fibonacci, Width=>2 );
  Put(" terms in the fibonacci sequence is"); New_Line;
  for I in Fibonacci_Array'range loop
    Put( Image( Fibonacci_Numbers(I) ) );
    New_Line;
  end loop;
end Main;
```

which would print out the first 50 terms of the Fibonacci series.

Hint:

- Use an array to store the 200 digits of the number.

# 13 Exceptions

This chapter looks at the way errors and exceptions are handled by the Ada system. Unlike many languages, Ada allows the user to capture and continue processing after an error or user-defined exception has occurred.

## 13.1 The exception mechanism

When writing code for an application it is tedious to have to keep testing for exceptional conditions such as 'Data store full'. The likely outcome is that the user will not test for the exception. Ada provides the elegant solution of allowing code to raise an exception that can be caught by a user of that code. If the user does not provide an exception handler, the exception is propagated upwards to the potential caller of the user's code. If no one has provided an exception handler, then the program will fail with a run-time message of the form 'Exception Data store full not handled'.

The following program reads in an Integer number from the user and prints the corresponding character represented by this number in the Ada character set.

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Number : Integer;           --Number read in
  Ch      : Character;        --As a character
begin
  loop
    begin
      Put("Enter character code : ");           --Ask for number
      exit when End_Of_File;                   --EOF ?
      Get( Number ); Skip_Line;                 --Read number
      Put("Represents the character [");         --Valid number
      Put( Character'Val(Number) );
      Put("]");                                --Valid character
      New_Line;
    exception
      when Data_Error =>
        Put("Not a valid Number"); Skip_Line;   --Exception
        New_Line;
      when Constraint_Error =>
        Put("Not representable as a Character"); --Exception
        New_Line;
      when End_Error =>
        Put("Unexpected end of data"); New_Line; --Exception
        exit;
    end;
  end loop;
end Main;
```

In this program the following exceptions may occur:

Exception	Explanation
Constraint_Error	An invalid value has been supplied.
Data_Error	The data item read is not of the expected type.
End_Error	During a read operation the end of file was detected.

Section B.7, Appendix B gives a full list of the exceptions that can occur during the running of an Ada program.

A user may interact with the program as shown below.

```
Enter character code : 65
Represents the character [A]
Enter character code : Invalid
Not a valid Number
Enter character code : 999
Represents the character [Not representable as a Character]
Enter character code : ^D
```

*Note: The user input is shown in bold type.  
^D represents the end of file character, which on an unix system is control – d.*

## 13.2 Raising an exception

An exception is raised by way of the **raise** statement. For example, to raise the exception `Constraint_Error` the following statement is executed:

```
raise Constraint_Error;
```

Naturally, a user-defined exception can be raised. Firstly, the exception to be raised is declared:

```
Unexpected_Condition : Exception;
```

then the exception can be raised with:

```
raise Unexpected_Condition;
```

## 13.3 Handling any exception

It is possible to capture an exception without knowing its name by the use of a **when others** clause in an exception handler. For example, the additional handler:

```
when others =>
  Put("Exception caught"); New_Line;
```

could have been included with the previous program to capture any unexpected exceptions. If information is required about the exception then the handler can include a name for the exception. For example:

```
when The_Event: others =>
  Put("Unexpected exception is ");
  Put( Exception_Name( The_Event ) ); New_Line;
```

*Note:*     The object *Event* is declared as:

```
Event : Exception_Occurrence;
```

*and is defined in the package Ada.Exceptions.*

In the above exception handler the exception is known by the name *event*. Information about the exception is obtained by using the following functions:

Function (Defined in Ada.Exceptions)	Returns as a string:
Exception_Name(event)	In upper case the exception name starting with the root library unit.
Exception_Information(event)	Detailed information about the exception.
Exception_Message(event)	A short explanation of the exception.

Other functions and procedures in Ada.Exceptions are:

Function / procedure	Action
Reraise_Occurrence(event)	A procedure which re-raises the exception event.
Raise_Exception(e,"Mess")	A procedure which raises exception e with the message "Mess".

## 13.4 The cat program revisited

The program to concatenate the contents of files previously seen in Section 3.11 can now be re-written to give a sensible error message to the user when an attempt is made to list a file that does not exist. In this program the following exception occur:

Exception	Explanation
Name_Error	File does not exist.
Status_Error	File is already open.

This new program is:

```

with Ada.Text_IO, Ada.Command_Line;
use  Ada.Text_IO, Ada.Command_Line;
procedure Cat is
  Fd : Ada.Text_IO.File_Type;      --File descriptor
  Ch : Character;                  --Current character
begin
  if Argument_Count >= 1 then
    for I in 1 .. Argument_Count loop --Repeat for each file
      begin
        Open( File=>Fd, Mode=>In_File, --Open file
              Name=>Argument(I) );
        while not End_Of_File(Fd) loop --For each Line
          while not End_Of_Line(Fd) loop --For each character
            Get(Fd,Ch); Put(Ch);        --Read / Write character
          end loop;
          Skip_Line(Fd); New_Line;      --Next line / new line
        end loop;
        Close(Fd);                     --Close file
      exception
        when Name_Error =>
          Put("cat: " & Argument(I) & " no such file" );
          New_Line;
        when Status_Error =>
          Put("cat: " & Argument(I) & " all ready open" );
          New_Line;
        end;
      end loop;
    else
      Put("Usage: cat file1 ... "); New_Line;
    end if;
  end Cat;

```

## 13.5 A stack

A stack is a structure used to store and retrieve data items. Data items are pushed onto the structure and retrieved in reverse order. This is commonly referred to as ‘first in last out’. This process is illustrated in Figure 13.1

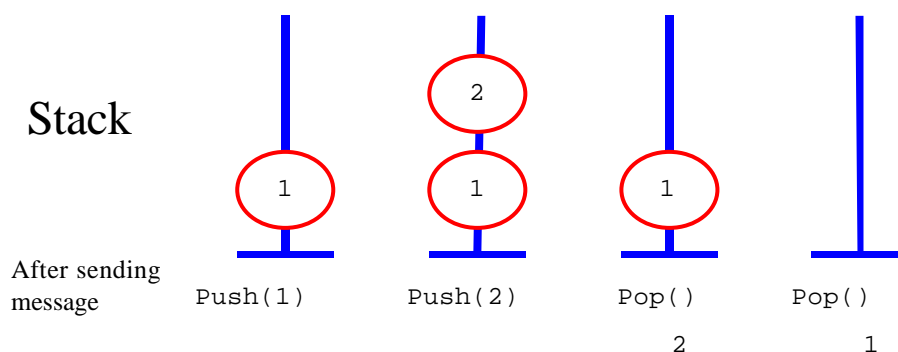


Figure 13.1 Example of operations on a stack.



A program to demonstrate the operation of a stack is developed with the aid of a class Stack. The operations Push, Pop and Reset can be performed on an instance of Stack . The responsibilities of these methods are as follows:

Method	Responsibility
Push	Push the current item onto the stack. The exception Stack_error will be raised if this cannot be done.
Pop	Return the top item on the stack, whereupon the item is removed from the stack. The exception Stack_error will be raised if this cannot be done.
Reset	Resets the stack to an initial state of empty.

The Ada specification of the class Stack is as follows:

```

package Class_Stack is
  type Stack is private;           --Copying allowed
  Stack_Error: exception;         --When error

  procedure Reset( The:in out Stack);
  procedure Push( The:in out Stack; Item:in Integer );
  procedure Pop(The:in out Stack; Item:out Integer );
private
  Max_Stack: constant := 3;
  type Stack_Index is range 0 .. Max_Stack;
  subtype Stack_Range is Stack_Index range 1 .. Max_Stack;
  type Stack_Array is array ( Stack_Range ) of Integer;

  type Stack is record
    Elements: Stack_Array;         --Array of elements
    Tos      : Stack_Index := 0;   --Index
  end record;
end Class_Stack;

```

In the specification of the class, Stack the actual representation used for the stack is an array. The array representing the stack is indexed from 1 .. Max\_Stack. However, so that an empty stack can be represented the variable holding the index to the current element in the stack TOS is allowed to hold the value 0 to represent an empty stack. Hence, in the specification of the array the following types and subtypes are used.

Type/subtype	Responsibility
Stack_Index	A type representing the index used to access the stack. As this is also used to represent an empty stack, it includes the value 0, which is not a valid index of the stack.
Stack_Range	A subtype used to represent the valid range of indexes in the stack.
Stack_array	The type used to declare the array representing the stack.

The following simple program demonstrates the operation of a stack. The program reads a line of text that consists of the following tokens:

Token	Meaning
+Number	Push Number Add Number to the stack.
-	Pop Remove the top item from the stack and print the removed item.

```

with Ada.Text_IO, Ada.Integer_Text_IO, Class_Stack;
use   Ada.Text_IO, Ada.Integer_Text_IO, Class_Stack;
procedure Main is
  Number_Stack : Stack;           --Stack of numbers
  Action       : Character;       --Action
  Number       : Integer;         --Number processed
begin
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is           --Process action
          when '+' =>
            Get( Number ); Push(Number_Stack,Number);
            Put("push number = "); Put(Number); New_Line;
          when '-' =>
            Pop(Number_Stack,Number);
            Put("Pop number = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      exception
        when Stack_Error =>
          Put("Stack_error"); New_Line;
        when Data_Error  =>
          Put("Not a number"); New_Line; Skip_Line;
        when End_Error   =>
          Put("Unexpected end of file"); New_Line; exit;
      end;
    end loop;
    Skip_Line;
  end loop;
  Reset( Number_Stack );
end Main;

```

### 13.5.1 Putting it all together

When compiled with a suitable package body the above program when run with the following data:

```
+1+2+3+4----
```

will produce the following output:

```

push number =      1
push number =      2
push number =      3
Stack_error
Pop number  =      3
Pop number  =      2
Pop number  =      1
Stack_error

```

### 13.5.2 Implementation of the stack

The implementation of the package uses the procedure `reset` to set the stack to a defined state, in this case empty.

```

package body Class_Stack is
  procedure Reset( The:in out Stack ) is
  begin
    The.Tos := 0;  --Set TOS to 0 (Non existing element)
  end Reset;

```

The exception `Stack_Error` is raised by the procedure `Push` if an attempt is made to add a new item to a full stack.

```

procedure Push( The:in out Stack; Item:in Integer ) is
begin
  if The.Tos /= Max_Stack then
    The.Tos := The.Tos + 1;           --Next element
    The.Elements( The.Tos ) := Item;  --Move in
  else
    raise Stack_Error;               --Failed
  end if;
end Push;

```

The procedure `Pop` similarly raises the exception `Stack_Error` if an attempt is made to extract an item from an empty stack.

```

procedure Pop( The:in out Stack; Item :out Integer ) is
begin
  if The.Tos > 0 then
    Item := The.Elements( The.Tos );  --Top element
    The.Tos := The.Tos - 1;           --Move down
  else
    raise Stack_Error;               --Failed
  end if;
end Pop;
end Class_Stack;

```

## 13.6 Self-assessment

- When should an exception be used?
- What happens when an exception is not caught in a user program?
- How can a program catch all exceptions which might be generated when executing a code sequence?
- Can a user program raise one of the system's exceptions such as `Constraint_error`?

## 13.7 Exercises

Construct the following class which uses exceptions:

- *Average*  
This class has the following methods:

Method	Responsibility
Add	Add a new data value.
Average	Deliver the average of the data values held.
Reset	Reset the object to its initial state.

The exception `No_Data` is raised when an attempt is made to calculate the average of zero numbers.

# 14 Generics

This chapter looks at generics that enable parameterized re-usable code to be written. The degree of re-usable ability, however, will depend on the skill and foresight of the originator.

## 14.1 Generic functions and procedures

The main problem in re-using code of previously written functions or procedures is that they are restricted to process specific types of values. For example, the function `order` developed as an exercise in Chapter 5 will only work for `Float` values. To be really useful to a programmer, this procedure should work for all objects for which a ‘greater than’ value can be defined. Ada allows the definition of generic functions or procedures. In this, the actual type(s) that are to be used are supplied by the user of the function or procedure. This is best illustrated by an example:

```
generic                                     --Specification
  type T is ( <> );                         --Any discrete type
procedure G_Order( A,B:in out T );         --Prototype ord

procedure G_Order( A,B:in out T ) is       --Implementation ord
  Tmp : T;                                 --Temporary
begin
  if A > B then                            --Compare
    Tmp := A; A := B; B := Tmp;           -- Swap
  end if;
end G_Order;                               --
```

The declaration of a generic function or procedure is split into two components: a specification part that defines the interface to the outside world, and an implementation part that defines the physical implementation. In the specification part, the type(s) that are to be used in the procedure are specified between the **generic** and the prototype line of the function or procedure. In this example a single type `T` is to be supplied. The type `T` must be one of Ada’s discrete types. The ‘(<>)’ in the declaration ‘**type T is (<>)**’ specifies this restriction. A full list of the restricted types to which a generic parameter can be constrained to are given in Section 14.2.

To use this procedure the user must first instantiate a procedure that will operate on a particular type. This is accomplished by the declaration:

```
procedure Order is new G_Order( Natural ); --Instantiate order
```

which defines a procedure `order` which will put into ascending order its two `Natural` parameters. It would, of course, be an error detected at compile-time to use the procedure `order` with any parameter of a type other than a `Natural` or a subtype of a `Natural`.

Another generic procedure `G_3Order` can be written which will order its three parameters. This new procedure uses an instantiation of the procedure `G_Order` internally.

```

generic                                --Specification
  type T is ( <> );                    --Any discrete type
procedure G_3Order( A,B,C:in out T ); --Prototype ord

with G_Order;
procedure G_3Order( A,B,C:in out T ) is --Implementation ord
  procedure Order is new G_Order( T ); --Instantiate order
begin
  Order( A, B );                      --S L -
  Order( B, C );                      --? ? L
  Order( A, B );                      --S M L
end G_3Order;                        --

```

*Note:* The generic parameter `T` can only be a member of the discrete types. This is achieved with the declaration of the type `T` as '`type T is ( <> )`'.  
The procedure `order` in the procedure `G_3Order` is an instantiation of the generic procedure `G_Order` with an actual parameter of type `T`.

Figure 14.1 illustrates the components of a generic procedure.

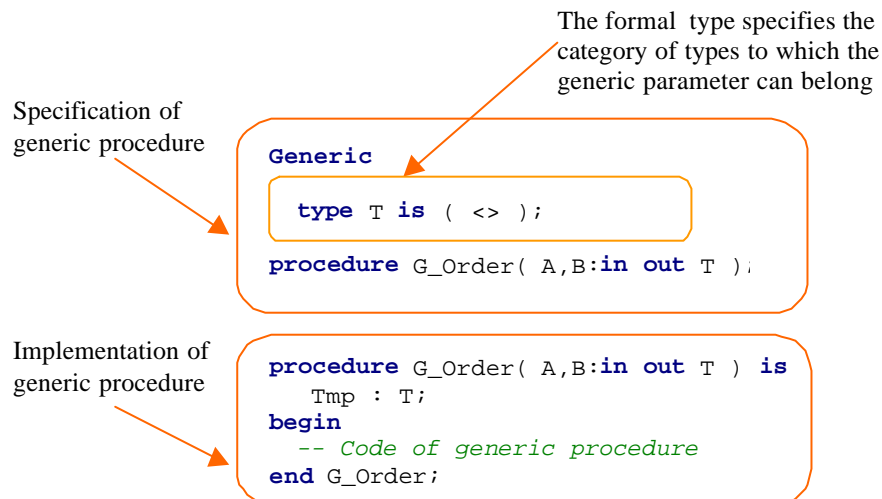


Figure 14.1 Components of a generic procedure declaration.

The above generic procedures G\_Order and G+3Order allow the following code to be written:

```
with Ada.Text_IO, Ada.Integer_Text_IO, G_3Order;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  procedure Order is new G_3Order( Natural );      --Instantiate
  Room1 : Natural := 30; --30 Square metres
  Room2 : Natural := 25; --25 Square metres
  Room3 : Natural := 20; --20 Square metres
begin
  Order( Room1, Room2, Room3 );
  Put("Rooms in ascending order of size "); New_Line;
  Put( Room1 ); New_Line;
  Put( Room2 ); New_Line;
  Put( Room3 ); New_Line;
end Main;
```

which would produce the following results when run:

```
Rooms in ascending order of size are
    20
    25
    30
```

### 14.1.1 Advantages and disadvantages of generic units

- |                      |   |
|----------------------|---|
| <i>Advantages</i>    | <ul style="list-style-type: none"> <li>• Facilitate re-use by allowing an implementor to write procedures or functions which process objects of a type determined by the user of the procedure or function.</li> </ul>  |
| <i>Disadvantages</i> | <ul style="list-style-type: none"> <li>• Extra care must be exercised in writing the procedure or function. This will undoubtedly result in a greater cost to the originator.</li> <li>• The implementation of the generic procedure, function or package may not be as efficient as a direct implementation. The compiler may generate only one sequence of code to handle all instances of the generic procedure, function or package.</li> </ul> |

## 14.2 Specification of generic component

The formal type specification constrains the actual type passed as a parameter to belong to a particular category of types. Examples of these categories are listed in the table below:

Formal type specification type T	In Ada 83	Actual parameter can belong to the following types	Note
<b>is private</b>	√	Any non limited type.	1
<b>is limited private</b>	√	Any type.	2
<b>is tagged</b>	✗	Any non limited tagged type.	1
<b>is limited tagged</b>	✗	Any tagged type.	2
<b>is</b> (<>)	√	Any discrete type, constrained type	1,5
(<>) <b>is private</b>	✗	Any discrete or indefinite non limited type.	2,3
(<>) <b>is limited private</b>	✗	Any discrete or indefinite type.	2,3
<b>is mod</b> <>	✗	Any modular type.	1
<b>is range</b> <>	√	Any integer type.	1
<b>is digits</b> <>	√	Any float type.	1
<b>is delta</b> <>	√	Any fixed ordinary type.	1
<b>is delta</b> <> <b>digits</b> <>	✗	Any fixed decimal type.	1
<b>is access</b>	√	Any access type.	4
<b>with procedure</b> ...	√	procedure matching the signature.	6
<b>with package</b> ...	✗	package matching the signature.	6

*Note 1* The formal parameter in the generic unit is restricted to a use compatible with the actual parameter.

*Note 2* The formal parameter is restricted to operations which are compatible with a limited type. Thus, assignment of, and the default comparison for equality and not equality are prohibited.

*Note 3* Cannot be used to declare an indefinite type without declaring its range. For example, the indefinite type:  
**type** String **is array** (Positive **range** <> ) of Character;  
 cannot be declared without specifying the range.

*Note 4* Access types are covered in chapter 14.  
 May also be **is access all** or **is access constant**

*Note 5* Ada 83 has the well-known problem that an indefinite type may be used as a formal parameter. If the formal parameter is used to declare an object in the body of the generic unit, then on the instantiation of the unit an error message will be generated from the body of the generic unit.

*Note 6* Used to specify a procedure, function or package that is used in the body of the generic unit.



## 14.3 Generic stack

The stack illustrated in Section 13.5 can be built as a generic package. First, the specification of the package that contains the generic components is defined:

```
generic
  type T is private;           --Can specify any type
  Max_Stack:in Positive := 3; --Has to be typed / not const
package Class_Stack is
  type Stack is tagged private;
  Stack_Error: exception;

  procedure Reset( The:in out Stack);
  procedure Push( The:in out Stack; Item:in T );
  procedure Pop( The:in out Stack; Item:out T );
private
  type Stack_Index is new Integer range 0 .. Max_Stack;
  subtype Stack_Range is Stack_Index
    range 1 .. Stack_Index(Max_Stack);
  type Stack_Array is array ( Stack_Range ) of T;

  type Stack is tagged record
    Elements: Stack_Array;      --Array of elements
    Tos      : Stack_Index := 0; --Index
  end record;
end Class_Stack;
```

*Note:* The constant `Max_Stack` must be given a type because it is passed as a generic parameter.

The implementation of the package follows the same strategy as seen in Section 13.5.2 except that the constant that defines the size of the stack is now typed. The body of the package takes this into account by converting the constant object `Max_Stack` into an object of type `Stack_Index`. The body of the package is implemented as follows:

```
package body Class_Stack is

  procedure Push( The:in out Stack; Item:in T ) is
  begin
    if The.Tos /= Stack_Index(Max_Stack) then
      The.Tos := The.Tos + 1;      --Next element
      The.Elements( The.Tos ) := Item; --Move in
    else
      raise Stack_Error;          --Failed
    end if;
  end Push;
```

## 196 Child libraries

The procedure pop returns the top item on the stack.

```
procedure Pop( The:in out Stack; Item: out T ) is
begin
  if The.Tos > 0 then
    Item := The.Elements( The.Tos );      --Top element
    The.Tos := The.Tos - 1;               --Move down
  else
    raise Stack_Error;                   --Failed
  end if;
end Pop;
```

The procedure reset resets the stack to empty.

```
procedure Reset( The:in out Stack ) is
begin
  The.Tos := 0;  --Set TOS to 0 (Non existing element)
end Reset;

end Class_Stack;
```

A generic package cannot be used directly. First an instantiation of the package must be made for a specific type. For example, to instantiate an instance of the above package Class\_Stack to provide an Integer stack, the following declaration is made:

```
with Class_Stack;
pragma Elaborate_All( Class_Stack );
package Class_Stack_Int is new Class_Stack(Integer);
```

*Note: As the size of the stack is not specified the default value of 3 is used.  
The **pragma** to cause an elaboration of the generic package. This causes the compiler to generate a specific instance of the package.*

The newly created package Class\_Stack\_Int can then be used in a program.

### 14.3.1 Putting it all together

The new package `Class_stack_int` is tested by the following program unit which is identical to the code seen in Section 13.5 except the **with**'ed and **used**'ed package is `Class_Stack_Int` rather than `Class_Stack..`

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Stack_Int;
use  Ada.Text_IO, Ada.Integer_Text_IO, Class_Stack_Int;
procedure Main is
  Number_Stack : Stack;           --Stack of numbers
  Action       : Character;       --Action
  Number       : Integer;         --Number processed
begin
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is           --Process action
          when '+' =>
            Get( Number ); Push(Number_Stack,Number);
            Put("push number = "); Put(Number); New_Line;
          when '-' =>
            Pop(Number_Stack,Number);
            Put("Pop number = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      end loop;
      Skip_Line;
    end loop;

    exception
      when Stack_Error =>
        Put("Stack_error"); New_Line;
      when Data_Error  =>
        Put("Not a number"); New_Line; Skip_Line;
      when End_Error   =>
        Put("Unexpected end of file"); New_Line; exit;
    end;
  end loop;

  Reset( Number_Stack );
end Main;
```

When run with the following data:

```
+1+2+3+4----
```

the following results will be produced:

```
push number =      1
push number =      2
push number =      3
Stack_error
Pop number  =      3
Pop number  =      2
Pop number  =      1
Stack_error
```

### 1.1.2 Implementation techniques for a generic package

The implementation of a generic package is usually performed by one of the following mechanisms:

- A new package is generated for each unique instantiation of the generic package. This is sometimes referred to as the macro implementation.
- A single code body is used which can cater for different formal types.

## 14.4 Generic formal subprograms

When specifying a generic formal type, the compiler must know how to perform operations on an instance of this type. For example, if the formal type is specified as `private`, then any Ada private type can be used. The implementor of the body of the generic package may wish to use instances of this type in a comparison. For example, as part of a logical expression:

```
if Instance_Of_Formal_Type > Another_Instance_Of_Formal_type then
...
end if;
```

For this to be allowed, the type passed must allow, in this case, for the ">" operation to be performed between instances of the generic type. To enforce this contract the specification part of the generic procedure, function or package must include a generic formal parameter for the ">" logical operation. Remember, the type passed may not have ">" defined between instances of the type. For example, the class `Account` does not provide any comparison operators between instances of an `Account`.

The following generic procedure which orders its formal parameters of type `private` is defined with a formal subprogram specification for ">".

```
generic
  type T is private;           --Specification
  with function ">" ( A, B:in T ) --Any non limited type
    return Boolean is <>;       --Need def for >
procedure G_Order( A,B:in out T ); --Prototype G_Order

procedure G_Order( A,B:in out T ) is --Implementation G_Order
  Tmp : T;
begin
  if A > B then                 --Compare
    Tmp := A; A := B; B := Tmp; --Swap
  end if;
end G_Order;                   --
```

The generic formal subprogram:

```
with function ">" ( A, B:in T )
  return Boolean is <>;           --Need def for >
```

specifies that on instantiation of the package, a definition for ">" between instances of the formal parameter `T` must be provided. The `<>` part of the generic formal subprogram specifies that this formal parameter has as a default value of the current definition for ">" at the point of instantiation of the generic.

Thus, an instantiation of a procedure order to order two `Natural` values would be:

```
with G_Order;
  procedure Order is new G_Order( Natural );    --Instantiate
```

*Note: The default value for the function ">" is the current definition for ">" between `Natural`s at the point of instantiation. If the operator ">" has not been overloaded it will be the intrinsic function defined in `Standard` for ">" . Naturally, if the operator ">" is not defined between instances of the formal type `T`, a compile-time error message will be generated.*

If the generic formal subprogram had been of the form:

```
with function ">" ( A, B:in T )
  return Boolean;    --Need def for >
```

then the formal parameter does not have a default value and therefore an actual parameter for a function with signature **function** ( a, b:in T ) must be specified on the instantiation. In this case the instantiation of `order` would be:

```
with G_Order;
  procedure Order is new G_Order( Natural, ">" );    --Instantiate
```

*Note: If the function ">" has not been overloaded then the function used will be the intrinsic function for ">" in the package `Standard`. If the instantiation had been :*

```
with G_Order;
  procedure Order is new G_Order( Natural, "<" );
```

*Then the items would be ordered in descending order.*

### 14.4.1 Example of the use of the generic procedure G\_3Order

A program to order the height of three people is shown below. In this program, a record of type `Person` is created for each of the individuals. A specific instance of the generic procedure `G_3Order` is created to order these records into ascending height order.

```
with Ada.Text_IO, G_3Order;
use Ada.Text_IO;
procedure Main is
  Max_Chars : constant := 7;
  type Gender is ( Female, Male );
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name : String( 1 .. Max_Chars ); --Name as a String
    Height : Height_Cm := 0;         --Height in cm.
    Sex : Gender;                    --Gender of person
  end record;

  function Gt( First, Second:in Person ) return Boolean is
  begin
    return First.Height > Second.Height;
  end Gt;

  procedure Order is new G_3Order( Person, Gt ); --Instantiate
  Person1 : Person := ( "Corinna", 171, Female );
  Person2 : Person := ( "Miranda", 74, Female );
  Person3 : Person := ( "Mike", 183, Male );

begin
  Order( Person1, Person2, Person3 );
  Put( "List of people in ascending height order are" ); New_Line;
  Put( Person1.Name ); New_Line;
  Put( Person2.Name ); New_Line;
  Put( Person3.Name ); New_Line;
end Main;
```

which when run would print:

```
List of people in ascending height order are
Miranda
Corinna
Mike
```

### 14.4.2 Summary

The following table summarizes the effect of different subprogram specifications for a formal parameter.

Generic formal subprogram	Explanation
<code>with function "&gt;" ( a, b:in T ) return Boolean is &lt;&gt;;</code>	Has a default value of the current definition of ">" at the point of instantiation of the generic subprogram.
<code>with function "&gt;" ( a, b:in T ) return Boolean;</code>	Takes the value of the formal parameter ">" at the point of instantiation of the generic subprogram.
<code>with procedure exec is exec;</code>	Takes the value of the formal parameter at the point of definition of the generic subprogram.

## 14.5 Sorting

Some of the simplest sorting algorithms are based on the idea of a bubble sort. In an ascending order bubble sort, consecutive pairs of items are compared, and arranged if necessary into their correct ascending order. The effect of this process is to move the larger items to the end of the list. However, in a single pass through the list only the largest item not already in the correct position will be guaranteed to be moved to the correct position. The process of passing through the list exchanging consecutive items is repeated until all the items in the list are in the correct order. For example, the following list of numbers is to be sorted into ascending order:

20	10	17	18	15	11
----	----	----	----	----	----

The first pass of the bubble sort compares consecutive pairs of numbers and orders each pair into ascending order. This is illustrated in Figure 14.2 below.

20		10		10		10		10		10
10		20		17		17		17		17
17		17		20		18		18		18
18		18		18		20		15		15
15		15		15		15		20		11
11		11		11		11		11		20

Figure 14.2 The first pass of the bubble sort.

Each pass through the list of numbers moves the larger numbers towards the end of the list and the smaller numbers towards the start of the list. However, only one additional number in the list is guaranteed to be in the correct position. The result of cumulative passes through the list of numbers is illustrated in the table below.

List of numbers	Commentary
20 10 17 18 15 11	The original list.
10 17 18 15 11 20	After the 1st pass through the list.
10 17 15 11 18 20	After the 2nd pass through the list.
10 15 11 17 18 20	After the 3rd pass through the list.
10 11 15 17 18 20	After the 4th pass through the list.

The process is repeated until there have been no swaps during a single pass through the list. Thus, after the 4th pass an additional pass through the list will be made in which no changes will occur. This indicates that the list is sorted.

### 14.5.1 Efficiency

This variation on the bubble sort is not a very efficient algorithm, as in the worse case it will take  $n$  passes through the list to rearrange the data into ascending order, where  $n$  is the size of the list. Each pass through the list will result in  $n-1$  comparisons.

The big O notation is used to give an approximation of the order of an algorithm. For this modified bubble sort the order (number of comparisons) will be approximately  $O(n^2)$ . For a small amount of data this is not important, but if  $n$  is large then the number of comparisons will be very large, and hence the time taken to sort the data will be lengthy.

## 14.6 A generic procedure to sort data

A generic sort procedure using the above variation of the bubble sort algorithm has the following specification:

```
generic
  type T          is private;           --Any non limited type
  type Vec_Range is (<>);               --Any discrete type
  type Vec        is array( Vec_Range ) of T;
  with function ">"( First, Second:in T ) return Boolean is <>;
procedure Sort( Items:in out Vec );
```

The generic formal parameters for the procedure Sort are:

Formal parameter	Description
<b>type T is private;</b>	The type of data item to be sorted.
<b>type Vec_Range is (&lt;&gt;);</b>	The type of the index to the array.
<b>type Vec is array( Vec_Range ) of T;</b>	The type of the array to be sorted.
<b>with function "&gt;"(First,Second:in T) return Boolean is &lt;&gt;;</b>	A function that the user of the generic procedure provides to compare pairs of data items.

The implementation of the generic procedure is:

```
procedure Sort( Items:in out Vec ) is
  Swaps : Boolean := True;
  Tmp    : T;
begin
  while Swaps loop
    Swaps := False;
    for I in Items'First .. Vec_Range'Pred(Items'Last) loop
      if Items( I ) > Items( Vec_Range'Succ(I) ) then
        Swaps := True;
        Tmp := Items( Vec_Range'Succ(I) );
        Items( Vec_Range'Succ(I) ) := Items( I );
        Items( I ) := Tmp;
      end if;
    end loop;
  end loop;
end Sort;
```

*Note: Passes through the data are repeated until there are no more swaps.  
The use of 'Succ delivers the next index. Remember the array might have an index of an enumeration type, so + cannot be used.*



### 14.6.1 Putting it all together

The following program illustrates the use of the generic procedure `sort` to sort a list of characters into ascending order.

```
with Ada.Text_IO, Sort;
use  Ada.Text_IO;
procedure Main is

  type Chs_Range is range 1 .. 6;
  type Chs       is array( Chs_Range ) of Character;

  procedure Sort_Chgs is new Sort (
    T      => Character,
    Vec_Range => Chs_Range,
    Vec     => Chs,
    ">"     => ">" );
  Some_Chgs : Chs := ( 'q', 'w', 'e', 'r', 't', 'y' );
begin
  Sort_Chgs( Some_Chgs );
  for I in Chs_Range loop
    Put( Some_Chgs( I ) ); Put( " " );
  end loop;
  New_Line;
end Main;
```

*Note:* The actual parameters used in the instantiation of the procedure `sort_chgs`.

When run, this will print:

```
e q r t w y
```

### 14.6.2 Sorting records

A program to sort an array of records is shown below. In this program each record represents a person's name and height. First, the declaration of the type `Person` which is:

```
with Ada.Text_IO, Sort;
use  Ada.Text_IO;
procedure Main is
  Max_Chgs : constant := 7;
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name   : String( 1 .. Max_Chgs ); --Name as a String
    Height : Height_Cm := 0;          --Height in cm.
  end record;
  type People_Range is (First, Second, Third, Forth );
  type People       is array( People_Range ) of Person;
```

## 204 *Child libraries*

Then the declaration of two functions: the function `Cmp_Height` that returns true if the first person is taller than the second and the second function `Cmp_Name` that returns true if the first person's name collates later in the alphabet than the second.

```
function Cmp_Height(First, Second:in Person) return Boolean is
begin
    return First.Height > Second.Height;
end Cmp_Height;

function Cmp_Name( First, Second:in Person ) return Boolean is
begin
    return First.Name > Second.Name;
end Cmp_Name;
```

Two instantiations of the generic procedure `sort` are made, the first to sort people into ascending height order, the second to sort people into ascending name order.

```
procedure Sort_People_Height is new Sort (
    T      => Person,
    Vec_Range => People_Range,
    Vec     => People,
    ">"     => Cmp_Height );

procedure Sort_People_Name is new Sort (
    T      => Person,
    Vec_Range => People_Range,
    Vec     => People,
    ">"     => Cmp_Name );
```

The body of the program which orders the friends into ascending height and name order is:

```
Friends : People := ( ("Paul   ", 146 ), ("Carol  ", 147 ),
                      ("Mike   ", 183 ), ("Corinna", 171 ) );
begin
    Sort_People_Name( Friends );                --Name order
    Put( "The first in ascending name order is  " );
    Put( Friends( First ).Name ); New_Line;
    Sort_People_Height( Friends );              --Height order
    Put( "The first in ascending height order is " );
    Put( Friends( First ).Name ); New_Line;
end Main;
```

which when run will print:

```
The first in ascending name order is Carol
The first in ascending height order is Paul
```

## 14.7 Generic child library

The stack seen in Section 13.5 can be extended to include the additional methods of:

Method	Responsibility
Top	Return the top item of the stack without removing it from the stack.
Items	Return the current numbers of items in the stack.

An efficient implementation is to access the private instance attributes of the class `Stack` directly. This can be done by creating a child package of the generic package `Class_Stack`. However, as the parent class is generic, its child package must also be generic. The specification of this generic child package is as follows:

```
generic
package Class_Stack.Additions is
  function Top( The:in Stack ) return T;
  function Items( The:in Stack ) return Natural;
private
end Class_Stack.Additions;
```

*Note:* As the child can see the components of the parent, it can also see any generic types.  
The implementation of the class is then:

```
package body Class_Stack.Additions is

  function Top( The:in Stack ) return T is
  begin
    return The.Elements( The.Tos );
  end Top;

  function Items( The:in Stack ) return Natural is
  begin
    return Natural(The.Tos);
  end Items;

end Class_Stack.Additions;
```

A generic child of a package is considered to be declared within the generic parent. Thus, to instantiate an instance of the parent and child the following code is used:

```
with Class_Stack;
pragma Elaborate_All( Class_Stack );
package Class_Stack_Pos is new Class_Stack(Positive,10);

with Class_Stack_Pos, Class_Stack.Additions;
pragma Elaborate_All( Class_Stack_Pos, Class_Stack.Additions );
package Class_Stack_Pos_Additions is
  new Class_Stack_Pos.Additions;
```

*Note:* The name of the instantiated child package is an Ada identifier.

### 14.7.1 Putting it all together

The following program tests the child library:

```
with Ada.Text_Io, Ada.Integer_Text_Io,
      Class_Stack_Pos, Class_Stack_Pos_Additions;
use   Ada.Text_Io, Ada.Integer_Text_Io,
      Class_Stack_Pos, Class_Stack_Pos_Additions;
procedure Main is
  Numbers : Stack;
begin
  Push( Numbers, 10 );
  Push( Numbers, 20 );
  Put( "Top item " ); Put( Top( Numbers ) ); New_Line;
  Put( "Items     " ); Put( Items( Numbers ) ); New_Line;
end Main;
```

which when run gives these results:

```
Top item      20
Items         2
```

## 14.8 Inheriting from a generic class

The class Stack seen in Section 13.5 and its generic child seen in Section 14.7 can be extended to include the additional method of:

Method	Responsibility
Depth	Return the maximum depth that the stack reached.

The specification for the new class Better\_Stack is:

```
with Class_Stack, Class_Stack.Additions;
generic
  type T is private;
  Max_Stack:in Positive := 3; --Has to be typed / not const
package Class_Better_Stack is
  package Class_Stack_T is new Class_Stack(T,Max_Stack);
  package Class_Stack_T_Additions is new Class_Stack_T.Additions;

  type Better_Stack is new Class_Stack_T.Stack with private;

  procedure Push( The:in out Better_Stack; Item:in T );
  function  Max_Depth( The:in Better_Stack ) return Natural;
private
  type Better_Stack is new Class_Stack_T.Stack with record
    Depth : Natural := 0;
  end record;
end Class_Better_Stack;
```

*Note:* Be aware of the instantiation of the base class Stack and its generic child within the body of the inheriting class.

The procedure push is overloaded so that it can record the maximum depth reached.

The implementation of this inherited class is:

```
package body Class_Better_Stack is

  procedure Push( The:in out Better_Stack; Item:in T ) is
    D : Natural;
  begin
    Class_Stack_T.Push( Class_Stack_T.Stack(The), Item );
    D := Class_Stack_T_Additions.Items(Class_Stack_T.Stack(The) );
    if D > The.Depth then
      The.Depth := The.Depth + 1;
    end if;
  end Push;

  function Max_Depth( The:in Better_Stack ) return Natural is
  begin
    return The.Depth;
  end Max_Depth;

end Class_Better_Stack;
```

### 14.8.1 Putting it all together

An instantiation of the class Better\_stack for Positive numbers is created with the declaration:

```
with Class_Better_Stack;
pragma Elaborate_All( Class_Better_Stack );
package Class_Better_Stack_Pos is
  new Class_Better_Stack(Positive,10);
```

This is then used in a small test program of the new class as follows:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Better_Stack_Pos;
use Ada.Text_IO, Ada.Integer_Text_IO, Class_Better_Stack_Pos;
procedure Main is
  Numbers : Better_Stack;
  Res      : Positive;
begin
  Put("Max depth "); Put( Max_Depth( Numbers ) ); New_Line;
  Push( Numbers, 10 );
  Push( Numbers, 20 );
  Put("Max depth "); Put( Max_Depth( Numbers ) ); New_Line;
  Push( Numbers, 20 );
  Put("Max depth "); Put( Max_Depth( Numbers ) ); New_Line;
  Pop( Numbers, Res );
  Put("Max depth "); Put( Max_Depth( Numbers ) ); New_Line;
  null;
end Main;
```

which when run produces the following results:

```
Max depth  0
Max depth  2
Max depth  3
Max depth  3
```

## 14.9 Self-assessment

- How do generic functions and packages help in producing re-usable code?
- Why can an implementor specify the possible types that can be used as a generic parameter to their package, procedure or function?
- What mechanism(s) prevent a user supplying an inappropriate type as a generic parameter, for example, an instance of the class `Account` to a generic sort procedure? This would be inappropriate as the comparison operator `>` is not defined between instances of an `Account`.

## 14.10 Exercises

Construct the following procedure:

- *Sort (Better)*  
Modify the sort package so that a pass through the data does not consider items that are already in the correct order.

Construct the following classes:

- *Store*  
A store for data items which has as its generic parameters the type of the item stored and the type of the index used. The generic specification of the class is:

```
generic
  type Store_index   is private;      --
  type Store_element is private;      --
package Class_store is
  type Store is limited private;      -- NO copying
  Not_there, Full : exception;
  procedure add    ( the:in out Store;
                    index:in Store_index;
                    item:in Store_element);
  function deliver( the:in Store;
                    index:in Store_index )
    return Store_element;
private
  --
end Class_store;
```

- *Better Store*  
By adding to the class `store`, provide a class which will give a user of the class information about how many additional items may be added before the store fills up.

# 15 Dynamic memory allocation

This chapter shows how storage can be allocated and de-allocated arbitrarily by a program. An undisciplined use of this facility can lead to programs that are difficult to debug and fail in unpredictable ways.

## 15.1 Access values

Ada allows the access value of an object to be taken. An access value is a pointer or reference to an object which can be manipulated and used to access the original object. An access value is usually implemented as the physical address of the object in memory. For example, the declaration shown below elaborates storage for an object which is to hold an integer value.

```
People    : aliased Integer;
```

*Note:* In the declaration of `people` the prefix **aliased** denotes that an access value of the object `people` may be taken. If the prefix is omitted the access value of the object may not be taken.

The storage for `people` can be visualized as illustrated in Figure 15.1.

People 24

Figure 15.1 Storage for an instance of an Integer object.

To store a value into the Integer object, a normal assignment statement is used:

```
People    := 24;
```

whilst the declaration:

```
type P_Integer is access all Integer;  
  
P_People : P_Integer;
```

uses the access type `P_Integer` to declare an object `P_People` to hold an access value for an integer object. In the declaration of the access type `P_Integer` the keyword **all** signifies that read and write access may be made to the object described by the access value. The following code assigns to `p_people` the access value of `people`:

```
P_People := People'Access;    --Access value for people
```

*Note:* The attribute `'Access` is used which delivers from an object its access value. `Access` is used both as a keyword and as an attribute name. The attribute `'Access` can only be used when the object will exist for all the life-time of the access value. See Section 15.7 for a more detailed explanation of the consequences of this requirement.

## 210 Polymorphism

The storage for P\_People can be visualized as illustrated in Figure 15.2.

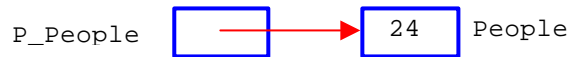


Figure 15.2 Storage for P\_People after it has been assigned the access value of people.

### 15.1.1 Access to an object via its access value

To access an object via its access value requires the use of `.all` which de-references the access value. This may be thought of as an indirection operator. For example, the following program accesses the object `people` by using the access value for `People` stored in the object `P_People`.

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  type P_Integer is access all Integer;
  People : aliased Integer;
  P_People : P_Integer;
begin
  People := 24;
  P_People := People'Access;      --Access value for people
  Put("The number of people is : "); Put( P_People.all );
  New_Line;
End Main;
```

*Note:* In the declaration of `P_integer`, **access all** signifies that read and write access may be made to the object via its access value.

which when run, would produce:

```
The number of people is :      24
```

The ideas described above have their origins in low-level assembly language programming where the address of an item may be freely taken and manipulated. Ada provides a more disciplined way of implementing these concepts.

### 15.1.2 Lvalues and rvalues

In working with access values it is convenient to think about the lvalue and rvalue of an object. The lvalue is the access value of the object, whilst the rvalue is the contents of the object. For example, in the statement:

```
value := amount;
```

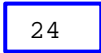


the object `amount` will deliver its contents, whilst the object `value` will deliver its access value so that the rvalue of `amount` may be assigned to it. The names lvalue and rvalue are an indication of whether the object is on the left or the right-hand side of an assignment statement.



In a program it is usual to deal with the contents or rvalue of an object. The access value of an object is its lvalue. For example, after the following fragment of code has been executed:

```
declare
  type P_Integer is access all Integer;
  type P_P_Integer is access all P_Integer;
  P_P_People : P_P_Integer;
  P_People   : aliased P_Integer;
  People     : aliased Integer;
begin
  People     := 42;
  P_People   := People'Access;
  P_P_People := P_People'Access;
end;
```

the following expressions will deliver the contents of the object people:

Expression	Diagram
People	
P_People.all	
P_P_People.all.all	

In a similar way the following statements will assign 42 to the object people.

Statement	Explanation
People := 42;	Straight-forward assignment.
P_People.all := 42;	Single level of indirection.
P_P_People.all.all := 42;	Double level of indirection.

### 15.1.3 Read only access

Access to an object via an access value may be restricted to read only by replacing **all** with **constant** in the declaration of the access type. For example, in the following fragment of code, only read access is allowed to People when using the access value held in P\_People.

```
declare
  type P_Integer is access constant Integer;
  People   : aliased Integer;
  P_People : P_Integer;
begin
  People   := 24;
  P_People := People'Access;      --Access value for people
  Put("The number of people is : "); Put( P_People.all );
  New_Line;
end;
```

## 15.2 Dynamic allocation of storage

The process described so far has simply used existing storage; the real power of access values accrue when storage is claimed dynamically from a storage pool. In Ada terminology an allocator is used to allocate storage dynamically from a storage pool. For example, storage can be allocated dynamically by using the allocator **new** as follows:

```
declare
  Max_Chars : constant := 7;
  type Gender is ( Female, Male );
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name      : String( 1 .. Max_Chars ); --Name as a String
    Height    : Height_Cm := 0;           --Height in cm.
    Sex       : Gender;                  --Gender of person
  end record;
  type P_Person is access Person;        --Access type
  P_Mike : P_Person;
begin
  P_Mike := new Person'( "Mike  ", 183, Male);
end;
```

*Note:* As the storage for a *Person* is always allocated dynamically from a specific storage pool, an access type that declares an object to hold an access value of a *Person* may be declared without the keyword **all** or **constant**.

The expression:

```
new Person'( "Mike  ", 183, Male);
```

returns the access value to dynamically allocated storage for a person initialized with the values given in the record aggregate. This could also have been written as:

```
P_Mike := new Person;
P_Mike.all := Person'( "Mike  ", 183, Male);
```

One way of managing dynamically allocated storage is to form a daisy chain of the allocated storage. The usual approach when implementing this technique is to include with the record component a value which can hold the access value of the next item in the daisy chain. The end of the chain is indicated by the value **null**. The value **null** is special as it is considered the null value for any access type. The Ada system will guarantee that no allocated access value can ever have the value **null**.

The code below forms a daisy chain of two items of storage which represent individual people.

```

declare
  Max_Chars : constant := 7;
  type Gender is ( Female, Male );
  type Height_Cm is range 0 .. 300;
  type Person;
  type P_Person is access Person;
  type Person is record
    Name : String( 1 .. Max_Chars );
    Height : Height_Cm := 0;
    Sex : Gender;
    Next : P_Person;
  end record;
  People : P_Person;
begin
  People := new Person'( "Mike ", 183, Male, null );
  People.Next := new Person'( "Corinna", 171, Female, null );
end;

```

*Note: P\_Person and Person are mutually dependent upon each other as both contain a reference to each other. To fit in with the rule that all items must be declared before they can be used, Ada introduces the concept of a tentative declaration. This is used when Person is defined as 'type Person;'. The full declaration of person is defined a few lines further down.*

The resultant data structure is illustrated in Figure 15.3.

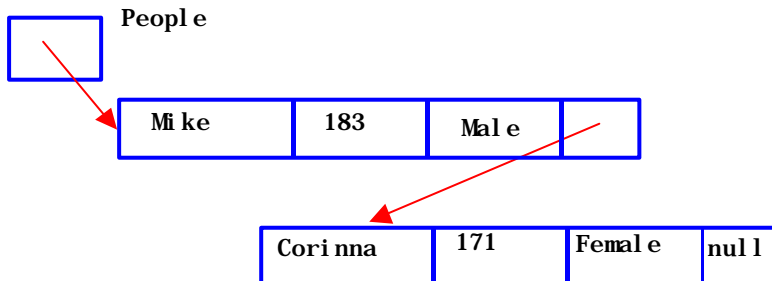


Figure 15.3 Daisy chain of two people.

In the above code the daisy chain of people could have been created with the single assignment statement:

```

People := new Person'( "Mike ", 183, Male,
  new Person'( "Corinna", 171, Female, null ) );

```

## 214 Polymorphism

An iterative procedure to print the names of people represented in this chain is:

```
procedure Put( Crowd: in P_Person ) is
  Cur : P_Person := Crowd;
begin
  while Cur /= null loop
    Put( Cur.Name ); Put( " is " );
    Put( Integer(Cur.Height), Width=>3 ); Put( "cm and is " );
    if Cur.Sex = Female then
      Put( "female" );
    else
      Put( "male" );
    end if;
    New_Line;
    Cur := Cur.Next;
  end loop;
end Put;
```

whilst a recursive version of the above procedure is written as::

```
procedure Put( Cur: in P_Person ) is
begin
  if Cur /= null then
    Put( Cur.Name ); Put( " is " );
    Put( Integer(Cur.Height), Width=>3 ); Put( "cm and is " );
    if Cur.Sex = Female then
      Put( "female" );
    else
      Put( "male" );
    end if;
    New_Line;
    Put( Cur.Next );
  end if;
end Put;
```

When executed:

```
declare
  -- Declarations omitted for brevity
  People : P_Person;
begin
  People := new Person'( "Mike  ", 183, Male,
                        new Person'( "Corinna", 171, Female, null) );
  Put( People );
end;
```

with either of the above procedures Put the code would produce the following results:

```
Mike    is 183cm and is male
Corinna is 171cm and is female
```

when called to print people.

### 15.2.1 Problems with dynamically allocated storage

The use of dynamically allocated storage can result in errors which can be difficult to track down in a program. Some of the potential problems associated with dynamic storage allocation are tabled below:

Problem	Result
Memory leak	The storage that is allocated is not always returned to the system. For a program which executes for a long time, this can result in eventual out of memory error messages.
Accidentally using the same storage twice for different data items.	This will result in corrupt data in the program and probably a crash which is difficult to understand.
Corruption of the chained data structure holding the data.	Most likely a program crash will occur some time after the corruption of the data structure.
Time taken to allocate and de-allocate storage is not always constant.	There may be unpredictable delays in a real-time system. However, a worst case Figure can usually be calculated.

### 15.3 Returning dynamically allocated storage

In Ada there are two processes used for returning dynamically allocated storage to the system. These are:

- The Ada run-time system implicitly returns storage once the type that was used to elaborate the storage goes out of scope.
- The programmer explicitly calls the storage manager to release dynamically allocated storage which is no longer active. This returned storage is then immediately available for further allocation in the program.

The advantages and disadvantages of the two processes described above are as follows:

Process	Advantages	Disadvantages
Storage reclamation implicitly managed by the system.	No problem about de-allocating active storage.	May result in a program consuming large amounts of storage even though its actual use of storage is small. In extreme cases this may prevent a program from continuing to run.
Storage de-allocation explicitly initiated by a programmer.	Prevents inactive storage consuming program address space.	If the programmer makes an error in the de-allocation then this may be very difficult to track down.

*Note:* The process of explicitly returning storage to the run-time system is described in Section 15.4.

As storage de-allocation can be an error-prone process, the best strategies are to either:

- Let the Ada run-time system do the de-allocation automatically for you.
- Hide allocation and de-allocation of storage in a class. The methods of the class can then be tested in isolation. The fully tested class can then be incorporated into a program.

## 15.3.1 Summary: access all, access constant, access

The following table summarizes the choice and restrictions that apply to the use of access values.

Note	Declaration (T is an Integer type)	Example of use
1	<pre>type P_T is access all T; a_t : aliased T; a_pt: P_T;</pre>	<pre>a_pt := a_t'Access; a_pt.all := 2;</pre>
2	<pre>type P_T is access constant T; a_t : aliased constant T := 0; a_pt: P_T;</pre>	<pre>a_pt := a_t'Access; Put( a_pt.all );</pre>
3	<pre>type P_T is access T; a_pt: P_T;</pre>	<pre>a_pt := new T; a_pt.all := 2;</pre>

Note 1: Used when it is required to have both read and write access to `a_t` using the access value held in `a_pt`. The storage described by `a_t` may also be dynamically created using an allocator.

Note 2: Used when it is required to have only read access to `a_t` using the access value held in `a_pt`. The storage described by `a_t` may also be dynamically created using an allocator.

Note 3: Used when the storage for an instance of a `T` is allocated dynamically. Access to an instance of `T` can be read or written to using the access value obtained from `new`.

This form may only be used when an access value is created with an allocator (`new T`).

## 15.4 Use of dynamic storage

The Stack package shown in Section 14.3 could be rewritten using dynamic storage allocation. In rewriting this package, the user interface to the package has not been changed. Thus, a user of this package would not need to modify their program.

There is however, one important difference and this is that as the implementation of the stack uses linked storage as the implementation stands it cannot be correctly copied. To prevent a user of the package from attempting to copy an instance of a stack the type `Stack` is created as a limited type.

```
generic
  type Stack_Element is private;
package Class_Stack is
  type Stack is limited private;
  Stack_Error : exception;

  procedure Push( The:in out Stack; Item:in Stack_Element );
  procedure Pop(The:in out Stack; Item :out Stack_Element );
  procedure Reset( The:in out Stack );
private
  type Node;
  type P_Node is access Node;
  pragma Controlled( P_Node );
  type Node is record
    Item : Stack_Element;
    P_Next : P_Node;
  end record;

  type Stack is record
    P_Head : P_Node := null;
  end record;
end Class_Stack;
```

Note: The compiler directive `pragma Controlled( P_Node )` to inform the compiler that the programmer will explicitly perform the storage de-allocation for data allocated with the type `P_Node`.

The package body is as follows:

```
with Unchecked_Deallocation;
pragma Elaborate_All( Unchecked_Deallocation );
package body Class_Stack is

  procedure Dispose is
    new Unchecked_Deallocation( Node, P_Node );
```

The procedure `Dispose` is an instantiation of the generic package `Unchecked_Deallocation` that returns space back to the heap. The parameters to the generic package `Unchecked_Deallocation` are of two types: firstly, the type of the object to be disposed, and secondly, the access type for this object.

*Note: This procedure does little error checking. It is important not to dispose of storage which is still active.*

The empty list is represented by the `p_head` containing the `null` pointer. The `null` pointer is used to indicate that currently the object `P_Head` does not point to any storage. This can be imagined as Figure 15.4.



Figure 15.4 A location containing the `null` access value or pointer.

When an item (in this case an Integer) has been added to the stack it will look like Figure 15.5.

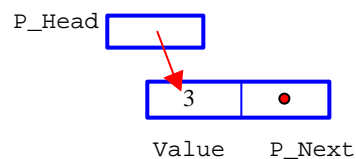


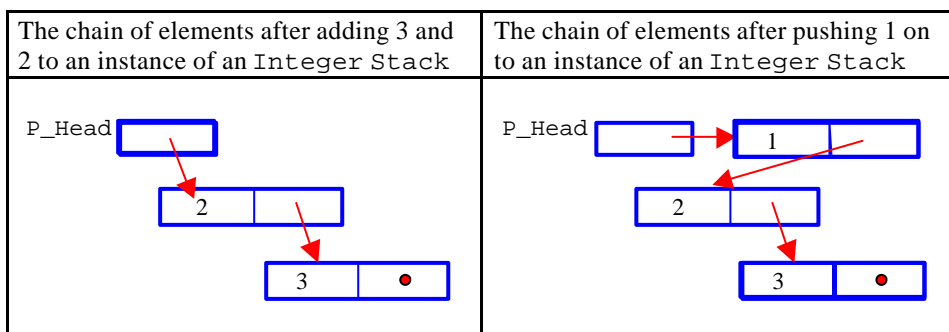
Figure 15.5 A stack containing one element.

To access the component value the `.` operator is used.

*Note: The compiler will generate the appropriate code to reference value. In this case it will involve a de-referencing through the pointer held in `P_Head`.*

```
P_Head.Item = 3;
```

The function `Push` creates a new element and chains this into a linked list of elements which hold the items pushed onto the stack.



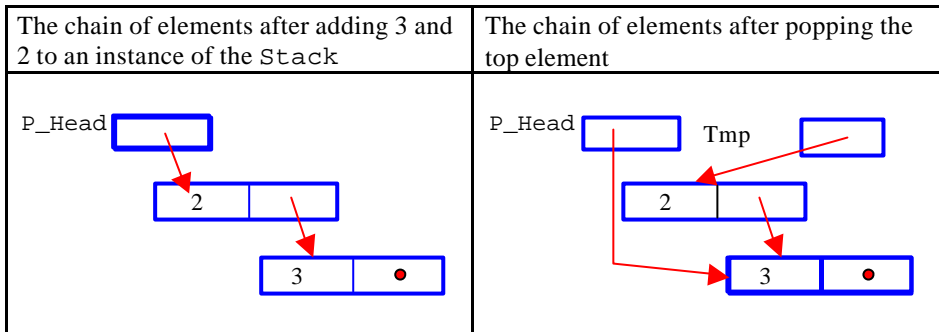
## 218 Polymorphism

```

procedure Push( The:in out Stack; Item:in Stack_Element ) is
    Tmp : P_Node;                                --Allocated node
begin
    Tmp := new Node'(Item=>Item, P_Next=>The.P_Head);
    The.P_Head := Tmp;
end Push;

```

Pop extracts the top item from the stack, and then releases the storage for this element back to the system.



```

procedure Pop( The:in out Stack; Item :out Stack_Element ) is
    Tmp : P_Node;                                --Free node
begin
    if The.P_Head /= null then                    --if item then
        Tmp := The.P_Head;                        --isolate top node
        Item := The.P_Head.Item;                  --extract item stored
        The.P_Head := The.P_Head.P_Next;          --Relink
        Dispose( Tmp );                           --return storage
    else
        raise Stack_Error;                        --Failure
    end if;
end Pop;

```

The procedure reset pops all existing elements from the stack. Remember the procedure pop releases the storage for the held item.

```

procedure Reset( The:in out Stack ) is
    Tmp : Stack_Element;
begin
    while The.P_Head /= null loop                --Re-initialize stack
        Pop( The, Tmp );
    end loop;
end Reset;

end Class_Stack;

```



### 15.4.1 Putting it all together

The following code tests the implementation of the previously compiled `Class_Stack`. Firstly an instance of an Integer stack is instantiated.

```
with Class_Stack;
pragma Elaborate_All( Class_Stack );
package Class_Stack_Int is new Class_Stack(Integer);
```

Then this package, is used in the following program that tests the stack implementation.

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Stack_Int;
use Ada.Text_IO, Ada.Integer_Text_IO, Class_Stack_Int;
procedure Main is
  Number_Stack : Stack;           --Stack of numbers
  Action       : Character;       --Action
  Number       : Integer;         --Number processed
begin
  Reset( Number_Stack );         --Reset stack to empty
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is           --Process action
          when '+' =>
            Get( Number ); Push(Number_Stack,Number);
            Put("push number = "); Put(Number); New_Line;
          when '-' =>
            Pop(Number_Stack,Number);
            Put("Pop number = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      exception
```

```
        when Stack_Error =>
          Put("Stack_error"); New_Line;
        when Data_Error =>
          Put("Not a number"); New_Line;
        when End_Error =>
          Put("Unexpected end of file"); New_Line; exit;
      end;
    end loop;
    Skip_Line;
  end loop;

  Reset( Number_Stack );
end Main;
```

When run with the data:

```
+1+2+3+4-----
```

## 220 Polymorphism

this program will produce the following results:

```
push number = 1
push number = 2
push number = 3
push number = 4
Pop number = 4
Pop number = 3
Pop number = 2
Pop number = 1
Pop: Exception Stack_error
```

This is essentially the same driver code as used on the previous implementation of a stack. This time, however, the stack is using dynamically allocated storage.

### 15.5 Hiding the structure of an object (opaque type)

So far, even though a client of a class cannot access the instance attributes of an instance of the class the client can still see the type and names of the instance attributes used in the object in the specification of the class. The instance attributes in an object can be hidden by moving the data declarations to the implementation part of the class. The specification part of the class will now contain an access value to a data structure whose contents are defined in the implementation part of the class.

If the specification part of the class no longer defines how much storage is to be used then the storage for an object must be allocated dynamically. The reason for this is that the compiler will not know how much storage to allocate for an object. Remember that the implementation part may have been separately compiled. The specification part of the class defines an access value which points to the storage for the object's instance attributes. For example, the class for a bank account can now be defined as:

```
with Ada.Finalization;
use  Ada.Finalization;
package Class_Account is
type Account is new Limited_Controlled with private;
  subtype Money is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;
  procedure Initialize( The:in out Account );
  procedure Finalize ( The:in out Account );
  procedure Deposit  ( The:in out Account; Amount:in Pmoney );
  procedure Withdraw ( The:in out Account; Amount:in Pmoney;
                        Get:out Pmoney );
  function  Balance  ( The:in Account ) return Money;
end Class_Account;
```

```
private
type Actual_Account;           --Details In body
type P_Actual_Account is access all Actual_Account;
type Account is new Limited_Controlled with record
  Acc : P_Actual_Account;      --Hidden in body
end record;
end Class_Account;
```

*Note:* Apart from the user-defined initialization and finalization, the procedure and function specification is the same as seen in Section 6.3.4.  
The declaration for the type `Actual_Account` is tentative.  
The base type of the class is `Limited_Controlled`. Thus assignments of instances of `Account` are prevented..

A benefit of the approach taken above is that a client of the class only needs to relink with any new implementation code even though the implementor of the class has changed the data in the object.

The implementation of the revised class Account is as follows:

```
with Unchecked_Deallocation;
package body Class_Account is

  pragma Controlled( P_Actual_Account ); -- We do deallocation
  type Actual_Account is record          --Hidden declaration
    Balance_Of : Money := 0.00;          --Amount in account
  end record;
```

The code for Initialize allocates the storage for the object automatically when an instance of the class is created. The body of finalize releases the storage when the object goes out of scope.

```
procedure Dispose is
  new Unchecked_Deallocation(Actual_Account, P_Actual_Account);

procedure Initialize( The:in out Account ) is
begin
  The.Acc := new Actual_Account;          --Allocate storage
end Initialize;

procedure Finalize ( The:in out Account ) is
begin
  if The.Acc /= null then                  --Release storage
    Dispose(The.Acc); The.Acc:= null;     --Note can be called
  end if;                                 -- more than once
end Finalize;
```

The code for Deposit, Withdraw and Balance are similar to the previous implementation of Account.

```
procedure Deposit ( The:in out Account; Amount:in Pmoney ) is
begin
  The.Acc.Balance_Of := The.Acc.Balance_Of + Amount;
end Deposit;
```

```
procedure Withdraw( The:in out Account; Amount:in Pmoney;
  Get:out Pmoney ) is
begin
  if The.Acc.Balance_Of >= Amount then
    The.Acc.Balance_Of := The.Acc.Balance_Of - Amount;
    Get := Amount;
  else
    Get := 0.00;
  end if;
end Withdraw;

function Balance( The:in Account ) return Money is
begin
  return The.Acc.Balance_Of;
end Balance;

end Class_Account;
```

Note: The automatic de-referencing of The.

## 222 Polymorphism

### 15.5.1 Putting it all together

The revised version of the class `Account` can be used in the same way as previously. In this program, the procedure `Statement` seen earlier in Section 6.3.2 is used to print details about an individual account.

```
with Ada.Text_IO, Class_Account, Statement;
use   Ada.Text_IO, Class_Account;
procedure Main is
  My_Account:Account;
  Obtain    :Money;
begin
  Statement( My_Account );

  Put("Deposit £100.00 into account"); New_Line;
  Deposit( My_Account, 100.00 );
  Statement( My_Account );

  Put("Withdraw £80.00 from account"); New_Line;
  Withdraw( My_Account, 80.00, Obtain );
  Statement( My_Account );

  Put("Deposit £200.00 into account"); New_Line;
  Deposit( My_Account, 200.00 );
  Statement( My_Account );
end Main;
```

which when run, would produce:

```
Mini statement: The amount on deposit is £ 0.00

Deposit _100.00 into account
Mini statement: The amount on deposit is £100.00

Withdraw _80.00 from account
Mini statement: The amount on deposit is £20.00

Deposit _200.00 into account
Mini statement: The amount on deposit is £220.00
```

*Note: The class `Account` allows the assignment of an instance of an `Account`. The consequences of this are that two objects will share the same storage. Section 17.4 explores and discusses this in detail and shows a solution to the problem.*

### 15.5.2 Hidden vs. visible storage in a class

The main benefit of this approach is that a client of the class does not need to recompile their code when the storage structure of the class is changed. The client code only needs to be relinked with the new implementation code. This would usually occur when a new improved class library is provided by a software supplier. Naturally, this assumes that the interface with the library stays the same.

The pros and cons of the two approaches are:

Criteria	Hidden storage	Visible storage
Compilation efficiency	Fewer resources required, as only a recompile of the class and then a re-link need to be performed.	Greater as all units that use the class need to be recompiled.
Run-time efficiency	Worse as there is the dynamic storage allocation overhead.	No extra run-time overhead.
Client access to data components of an object.	None	None

*Note: The extra cost of re-compiling and re-linking all units of a program may be marginal when compared with just re-linking.*

### 15.6 Access value of a function

The access value of a function or procedure may also be taken. This allows a function or procedure to be passed as a parameter to another function or procedure. For example, the procedure `apply` applies the function passed as a parameter to all elements of the array. The implementation of this function is shown in the fragment of code below:

```

type P_Fun is access function(Item:in Float) return Float;
type Vector is array ( Integer range <> ) of Float;

procedure Apply( F:in P_Fun; To:in out Vector ) is
begin
  for I in To'range loop
    To(I) := F( To(I) );
  end loop;
end Apply;

```

*Note: The de-referencing is done automatically when the function  $F$  is called. This could have been done explicitly with `F.all( To(I) )`. This explicit de-referencing is, however, required if the called function or procedure has no parameters.*

The first parameter to the procedure `apply` can be any function which has the signature:

```
function(Item:in Float) return Float;
```

Two such functions are:

```

function Square( F:in Float ) return Float is
begin
  return F * F;
end Square;

function Cube( F:in Float ) return Float is
begin
  return F * F * F;
end Cube;

```

## 224 Polymorphism

### 15.6.1 Putting it all together

Using the above declarations the following program can be written:

```
with Ada.Text_IO, Ada.Float_Text_IO;
use  Ada.Text_IO, Ada.Float_Text_IO;
procedure Main is
  type P_Fun is access function(Item:in Float) return Float;
  type Vector is array ( Integer range <> ) of Float;

  -- Body of the procedures apply, square and float

  procedure Put( Items:in Vector ) is
  begin
    for I in Items'Range loop
      Put( Items(I), Fore=>4, Exp=>0, Aft=>2 ); Put(" ");
    end loop;
  end Put;
begin
  Numbers := (1.0, 2.0, 3.0, 4.0, 5.0);
  Put("Square list :");
  Apply( Square'access, Numbers );
  Put( Numbers ); New_Line;
  Numbers := (1.0, 2.0, 3.0, 4.0, 5.0);
  Put("cube list  :");
  Apply( Cube'access, Numbers );
  Put( Numbers ); New_Line;
end Ex2;
```

which when run, will produce the following results:

Square list :	1.00	4.00	9.00	16.00	25.00
cube list :	1.00	8.00	27.00	64.00	125.00

*Note: A program is not allowed to take the address of a predefined operator such as Standard. "+". This is to ease compiler implementation.*

## 15.7 Attributes 'Access and 'Unchecked\_Access

The access value of an object may only be taken if the object is declared at the same lexical level or lower than the type declaration for the access value. If it is not, then a compile time error message will be generated when an attempt is made to take the object's access value. This is to prevent the possibility of holding an access value for an object which does not exist. For example, in the following program:

```

procedure Main is
  Max_Chars : constant := 7;
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name    : String( 1 .. Max_Chars );  --Name as a String
    Height  : Height_Cm := 0;           --Height in cm.
  end record;
  Mike    : aliased Person := Person("Mike  ",156);
begin
  declare
    type P_Person is access all Person;  --Access type
    P_Human: P_Person;
  begin
    P_Human:= Mike'access;                --OK
    declare
      Clive : aliased Person := Person("Clive  ", 160 );
    begin
      P_Human := Clive'Access;
    end;
    Put( P_Human.Name ); New_Line;        --Clive no longer exists
    P_Human := Mike'access;              --Change to Mike
  end;
end Main;

```

a compile time error message is generated for the line:

```
P_Human := Clive'Access;                -- Compile time error
```

as the object Clive does not exist for all the scope of the type P\_Person. In fact, there is a serious error in the program, as when the line:

```
Put( P_Human.Name ); New_Line;          -- Clive no longer exists
```

is executed, the storage that p\_human points to does not exist. Remember the scope of clive is the **declare** block.

In some circumstances the access value of an object declared in an inner block to the access type declaration is required. If this is so, then the compiler checking can be subverted or overridden by the use of 'Unchecked\_Access. Thus, the following code can be written:

```

procedure Main is
  -- Declaration of Person, P_Person, Mike etc.
begin
  P_Human:= Mike'Access;                -- OK
  declare
    Clive : aliased Person := Person("Clive  ", 160 );
  begin
    P_Human := Clive'Unchecked_Access;
    Put( P_Human.Name ); New_Line;      -- Clive
  end;
  P_Human:= Mike'Access;                --Change to Mike
  Put( P_Human.Name ); New_Line;      --Mike
end Main;

```

Of course the compiler can no longer help the programmer in detecting possible inconsistencies.

## 15.8 Self-assessment

- What is an access type? What is an access value?
- How is dynamic storage allocated?
- What mechanisms are available to return dynamically allocated storage?
- Why is dynamic storage allocation often considered a potential problem area in a program?
- How do you pass a procedure as a parameter to another procedure?
- Why is it essential to be able to call the procedure `Finalize` in the class `Account` more than once on the same object?
- What is the difference between `'Access` and `'Unchecked_Access`?

## 15.9 Exercises

Construct the following:

- *Store*  
A store for data items which has as its generic parameters the type of the item stored and the type of the index used. The generic specification of the class is:

```
generic
  type Store_index    is private;      --
  type Store_element is private;      --
package Class_store is
  type Store is limited private;      -- NO copying
  Not_there, Full : exception;

  procedure add    ( the:in out Store;
                    index:in Store_index;
                    item:in Store_element);
  function deliver( the:in Store;
                    index:in Store_index )
    return Store_element;
private
  --
end Class_store;
```

The implementation of the store uses a linked structure.

- *Queue*  
The class `Queue` implements a data structure in which items are added to the rear of the queue and extracted from the front. Implement this generic class using dynamic storage allocation.



# 16 Polymorphism

In the processes described so far, when a message is sent to an object, the method executed has been determinable at compile-time. This is referred to as static binding. If the type of an object that a message is sent to is not known until run-time, the binding between the method and the message is dynamic. Dynamic binding leads to polymorphism, which is when a message sent to an object causes the execution of a method that is dependent on the type of the object.

## 16.1 Rooms in a building

A partial classification of the different types of accommodation found in an office building is shown in Figure 16.1.

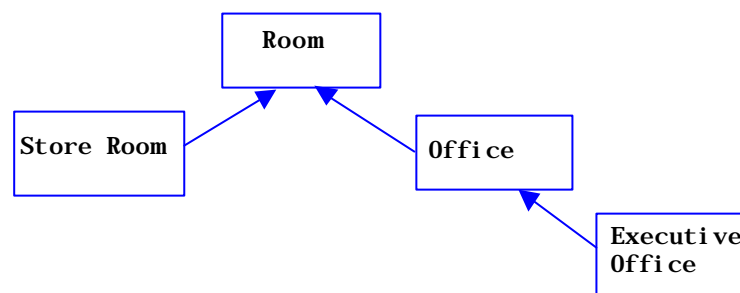


Figure 16.1 Partial classification of types of accommodation in a building.

The type of accommodation in each part of the building can be modelled using the Ada inheritance mechanism. First, a class `Room` to describe a general room is created. This class is then used as the base class for a series of derived classes that represent more specialized types of room. For example, an executive office is a more luxurious office, perhaps with wall-to-wall carpets and an outside view.

Each class derived from the class `Room`, including `Room`, has a function `describe` which returns a description of the accommodation.

A program is able to send the message `describe` to an instance of any type of accommodation and have the appropriate code executed. This is accomplished with function name overloading.

Figure 16.2 illustrates the call of a function `describe` on an instance of a class derived from `Room`.

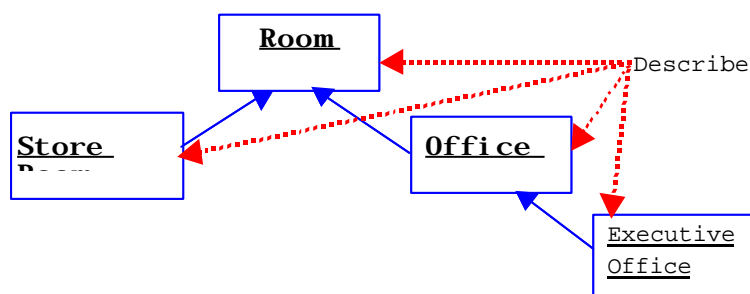


Figure 16.2 Call of `describe` on any instance of an object derived from `Room`.

**16.1.1 Dynamic binding**

Ada introduces the concept of a class-wide type to describe a tagged type and any type derived from the tagged type. The class-wide type is treated as an unconstrained type and is written `T'Class`, where `T` is a tagged type. For example, using the above hierarchy of types:

Class wide type	Can describe an instance of the following types
<code>Room'Class</code>	<code>Room</code> , <code>Office</code> , <code>Executive_Office</code> or <code>Store_Room</code>
<code>Office'Class</code>	<code>Office</code> or <code>Executive_Office</code>
<code>Executive_Office'Class</code>	<code>Executive_Office</code>
<code>Store_Room'Class</code>	<code>Store_Room</code>

*Note:* This is a departure from Ada's normal strict type checking, as any derived type of `T` can be implicitly converted to `T'Class`.

A class-wide type `T'Class` is considered to be an unconstrained type which must be initialized and is then only allowed to hold instances of the initialization type.

When a message such as `describe` is sent to an object of the class-wide type `Room'Class` the compiler does not know at compile-time which method to execute. The decision as to which method to execute must be determined at run-time by inspecting the object's tag. An object's tag describes its type. The mechanism of dynamic binding between an object and the message sent to it is referred to in Ada terminology as a run-time dispatch.

The object's tag can be explicitly examined using the attribute `'Tag`. For example:

```
if W422'Tag = W414'Tag then
  Put("Areas are the same type of accommodation");
  New_Line;
end if;
```

**16.2 A program to maintain details about a building**

A program that maintains details about a building stores individual details about rooms and offices. The details stored about a room include a description of its location. For an office, the details stored are all those for a room, plus the number of people who will occupy the room. The program will be required to give details about the individual areas in the building that may be a room or an office.

The responsibilities of a `Room` are as follows:

Method	Responsibility
<code>Initialize</code>	Store a description of the room.
<code>Describe</code>	Deliver a string containing a description of the room.
<code>Where</code>	Deliver the room's number.

This can be implemented as an Ada class specification as follows:

```
with B_String; use B_String;
package Class_Room is
  type Room is tagged private;

  procedure Initialize( The:in out Room; No:in Positive;
                      Mes:in String );
  function Where( The:in Room ) return Positive;
  function Describe( The:in Room ) return String;
private
  type Room is tagged record
    Desc : Bounded_String;      --Description of room
    Number: Positive;           --Room number
  end record;
end Class_Room;
```

Note: The package *B\_string* is an instantiation of the package *Ada.Strings.Bounded*. For example:

```
with Ada.Strings.Bounded;
use Ada.Strings.Bounded;
package B_String is new Generic_Bounded_Length( 80 );
```

The implementation of the class is:

```
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
package body Class_Room is

  procedure Initialize( The:in out Room;
                      No:in Positive; Mes:in String ) is
  begin
    The.Desc := To_Bounded_String( Mes );
    The.Number := No;
  end Initialize;

  function Where( The:in Room ) return Positive is
  begin
    return The.Number;
  end Where;

  function Describe( The:in Room ) return String is
    Num : String( 1 .. 4 );      --Room number as string
  begin
    Put( Num, The.Number );
    return Num & " " & To_String(The.Desc);
  end Describe;
end Class_Room;
```

The responsibilities of an Office are those of the Room plus:

Method	Responsibility
Initialize	Store a description of the office plus the number of occupants.
Describe	Returns a String describing an office.
No_Of_People	Return the number of people who occupy the room.

## 230 Polymorphism

The specification for a class Office extends the specification for a class Room as follows:

```
with Class_Room; use Class_Room;
package Class_Office is
  type Office is new Room with private;

  procedure Initialize( The:in out Office; No:in Positive;
                      Desc:in String; People:in Natural );
  function Deliver_No_Of_People(The:in Office) return Natural;
  function Describe( The:in Office ) return String;
private
  type Office is new Room with record
    People : Natural := 0;           --Occupants
  end record;
end Class_Office;
```

In the implementation of the class Office the procedure Initialize calls the inherited Initialize from class Room to store the description of the office. Remember the storage for the description in class Room is inaccessible to the class Office.

```
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
package body Class_Office is

  procedure Initialize( The:in out Office; No:in Positive;
                      Desc:in String; People:in Natural ) is
  begin
    Initialize( The, No, Desc );
    The.People := People;
  end Initialize;
```

The function Deliver\_No\_Of\_People returns the number of people who occupy the office.

```
function Deliver_No_Of_People( The:in Office ) return Natural is
begin
  return The.People;
end Deliver_No_Of_People;
```

The function describe is overloaded with a new meaning. In the implementation of the method Describe, a call is made to the method Describe in the class Room. To call the function Describe in the class Room, the function is passed an instance of Office viewed as a Room. This is referred to as a view conversion — the view changes, not the object. If this had not been done, a recursive call to describe in the class Office would have been made.

```
function Describe( The:in Office ) return String is
  No : String( 1 .. 4 );    --the.people as string
begin
  Put( No, The.People );
  return Describe( Room(The) ) &
    " occupied by" & No & " people";
end Describe;
end Class_Office;
```

### 16.2.1 Putting it all together

The above classes can be combined into a program which prints details about various rooms or offices in a building. The program is as follows:

```
with Ada.Text_IO, Class_Room, Class_Office;
use  Ada.Text_IO, Class_Room, Class_Office;
procedure Main is
  W422 : Room;
  W414 : Office;
```

The procedure `About` can take either an instance of a `Room` or an `Office` as a parameter. This is achieved by describing the parameter as `Room'Class`. The parameter of type `Room'Class` will match an instance of `Room` plus an instance of any type which is derived directly or indirectly from a `Room`.

In the procedure `About` the call to the function `Describe( Place )` is not resolvable at compile-time, as the object `place` may be either a `Room` or an `Office`. At run-time when the type of `Place` is known to the system, this call can be resolved. Thus, either `Describe` in the class `Room` or `Describe` in the class `Office` will be called.

```
procedure About( Place:in Room'Class ) is
begin
  Put( "The place is" ); New_Line;
  Put( " " & Describe( Place ) ); --Run time dispatch
  New_Line;
end About;
```

*Note: One way to implement the dynamic binding is for the object to contain information about which function `Describe` is to be called. This information can then be interrogated at run-time to allow the appropriate version of the function `Describe` to be executed. By careful optimization, the overhead of dynamic binding can be limited to a few machine cycles.*

The body of the test program `Main` is:

```
begin
  Initialize( W414, 414, "4th Floor west wing", 2 );
  Initialize( W422, 422, "4th Floor east wing" );

  About( W422 );           --Call with a room
  About( W414 );           --Call with an Office

end Main;
```

*Note: The call to `about` with an instance of a `Room` and an `Office` is possible because the type of the formal parameter is `Room'Class`.*

## 232 Polymorphism

When run, this program will produce the following output:

```
The place is
    422 4th Floor east wing
The place is
    414 4th Floor west wing occupied by    2 people
```

### 16.3 Run-time dispatch

For run-time dispatching to take place:

- The function or procedure must have a tagged type as a formal parameter.
- In the call of the function or procedure the actual parameter corresponding to the tagged type must be an instance of a class-wide type.

For example, the call to the function `describe(place)` in the procedure `about` will be a dispatching call.

If the actual class-wide type can represent two or more different typed objects which have procedures or functions with the same parameters, except the parameter for the class type, then polymorphism can take place. For example, in class `Room` and class `Office`, the function `describe` has the same signature except for the parameter with the class type.

In class <code>Room</code>	<b>function</b> <code>Describe(The:in Room) return String;</code>
In class <code>Office</code>	<b>function</b> <code>Describe(The:in Office) return String;</code>

The signatures of the functions and procedures in the class `Room` and the class `Office` are:

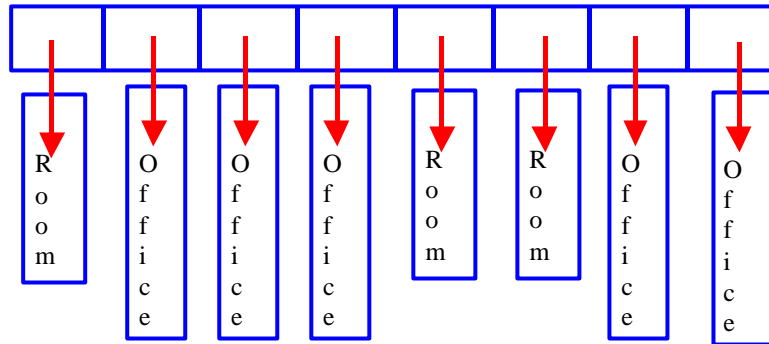
In class <code>Room</code>	In class <code>Office</code>
<code>Initialize(Room,String)</code>	<code>Initialize(Office,String)</code>
<code>Describe(Room) -&gt; String</code>	<code>Describe(Office) -&gt; String</code>
	<code>No_Of_People(Room) -&gt; Integer</code>
	<code>Initialize(Office,String,Natural)</code>

*Note: The function `initialize(Office,String)` in class `Office` is inherited from the class `Room`.*

### 16.4 Heterogeneous collections of objects

The real benefits of polymorphism accrue when a heterogeneous collection of related items is created. For example, a program which maintains details about accommodation in a building could use an array to hold objects which represent the different types of accommodation. Unfortunately, this technique cannot be implemented directly in Ada, as the size of individual members of the collection may vary. The solution in Ada is to use an array of pointers to the different kinds of object which represent the accommodation. In Ada, a pointer is referred to as an access value. An access value is usually implemented as the physical address in memory of the referenced object. An array of access values to objects of type `Room` and `Office` is illustrated in Figure 16.3.

Figure 16.3 Heterogeneous collection of Rooms and Offices.



### 16.4.1 An array as a heterogeneous collection

A heterogeneous collection of different types of accommodation can be modelled in an array. The array will contain for each type of accommodation a pointer to either an instance of a Room or an instance of an Office. For example:

```
type P_Room      is access all Room'Class;
type Rooms_Array is array (1 .. 15) of P_Room;
```

*Note:* *P\_Room is an access type which can declare an object which can hold the access value for a Room or any type derived from a Room.*

*The keyword **null** is a predefined value to indicate that the object contains no access value.*

The heterogeneous collection is then built using an array of P\_Room. The object's access value is used when entering a description of the accommodation into the heterogeneous array. For example, to enter details about room 414 into the heterogeneous array, the following code can be used.

```
declare
  P      : P_Room;
  Accommodation: Rooms_Array;
begin
  P := new Room;
  Initialize( P.all, 422, "4th Floor east wing" );
  Accommodation(1) := P;
end;
```

This inelegant code sequence is required as the instance attributes of the class Room are hidden and hence the construct:

Accommodation(1) := new Office( 414, "4th Floor east wing" ,2 );  
cannot be used.

### 16.4.2 Additions to the class Office and Room

To simplify later code, the classes Room and Office are extended to include additional methods to return an access value to an initialized object. These additional methods are:

Method	Responsibility
Build_Room	Deliver an access value to a dynamically created Room.
Build_Office	Deliver an access value to a dynamically created Office.

## 234 Polymorphism

A child package is used to implement this extension to the classes. The specification for the child package of Room is:

```
package Class_Room.Build is
  type P_Room is access all Room'Class;

  function Build_Room( No:in Positive;
                      Desc:in String ) return P_Room;
end Class_Room.Build;
```

whilst its implementation is:

```
package body Class_Room.Build is

  function Build_Room( No:in Positive;
                      Desc:in String ) return P_Room is
    P : P_Room;
  begin
    P := new Room; Initialize( P.all, No, Desc );
    return P;
  end Build_Room;

end Class_Room.Build;
```

The specification for the child package of Office is:

```
with Class_Room, Class_Room.Build;
use   Class_Room, Class_Room.Build;
package Class_Office.Build is

  function Build_Office( No:in Positive; Desc:in String;
                       People:in Natural ) return P_Room;
end Class_Office.Build;
```

The implementation of the package is:

```
package body Class_Office.Build is

  type P_Office is access all Office;

  function Build_Office( No:in Positive; Desc:in String;
                       People:in Natural ) return P_Room is
    P : P_Office;
  begin
    P := new Office; Initialize( P.all, No, Desc, People );
    return P.all'access;
  end Build_Office;
end Class_Office.Build;
```

*Note: The function Build\_Office returns an access value to a Room.*



## 16.5 A building information program

A class Building, which is used as a container to store and retrieve details about the accommodation in a building, has the following responsibilities:

Method	Responsibility
Add	Add a description of a room.
About	Return a description of a specific room.

The Ada specification for the class Building is:

```
with Class_Room, Class_Room.Build;
use   Class_Room, Class_Room.Build;
package Class_Building is

    type Building is tagged private;

    procedure Add( The:in out Building; Desc:in P_Room );
    function About(The:in Building; No:in Positive) return String;

private
    Max_Rooms : constant := 15;
    type Rooms_Index is range 0 .. Max_Rooms;
    subtype Rooms_Range is Rooms_Index range 1 .. Max_Rooms;
    type Rooms_Array is array (Rooms_Range) of P_Room;

    type Building is tagged record
        Last      : Rooms_Index := 0;  --Last slot allocated
        Description : Rooms_Array;      --Rooms in building
    end record;
end Class_Building;
```

The procedure Add adds new data to the next available position in the array.

```
package body Class_Building is

    procedure Add( The:in out Building; Desc:in P_Room ) is
    begin
        if The.Last < Max_Rooms then
            The.Last := The.Last + 1;
            The.Description( The.Last ) := Desc;
        else
            raise Constraint_Error;
        end if;
    end Add;
```

*Note: The exception Constraint\_Error is raised if there is no more free space in the array.*

## 236 Polymorphism

The function `about` uses a linear search to find the selected room number. If the room number does not exist, the returned string contains the text "Sorry room not known".

```
function About(The:in Building; No:in Positive) return String is
begin
  for I in 1 .. The.Last loop
    if Where(The.Description(I).all) = No then
      return Describe(The.Description(I).all);
    end if;
  end loop;
  return "Sorry room not known";
end About;
end Class_Building;
```

*Note: Chapter 17 explores more sophisticated container implementations.  
The appending of `.all` to an access value causes the object described by the access value to be delivered.*

### 16.5.1 Putting it all together

The classes `Room`, `Office` and `Building` can be used to build a program to allow visitors to a building to find out details about individual rooms. The program is split into two procedures. The first procedure declares and sets up details about individual rooms in the building.

```
with Ada.Text_IO,Ada.Integer_Text_IO,Class_Room,Class_Room.Build,
      Class_Office, Class_Office.Build, Class_Building;
use   Ada.Text_IO,Ada.Integer_Text_IO,Class_Room,Class_Room.Build,
      Class_Office, Class_Office.Build, Class_Building;
procedure Set_Up( Watts:in out Building ) is
begin
  Add( Watts, Build_Office( 414, "4th Floor west wing", 2 ) );
  Add( Watts, Build_Room ( 422, "4th Floor east wing" ) );
end Set_Up;
```

The second procedure interrogates the object `watts` to find details about individual rooms.

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Building, Set_Up;
use   Ada.Text_IO, Ada.Integer_Text_IO, Class_Building;
procedure Main is
  Watts : Building;           --Watts Building
  Room_No : Positive;         --Queried room
begin
  Set_Up( Watts );           --Populate building
  loop
    begin
      Put( "Inquiry about room: " );  --Ask
      exit when End_Of_File;
      Get( Room_No ); Skip_Line;      --User response
      Put( About( Watts, Room_No ) ); --Display answer
      New_Line;
    exception
      when Data_Error =>
        Put("Please retype the number"); --Ask again
        New_Line; Skip_Line;
    end;
  end loop;
end Main;
```

*Note: The program does not release the storage for the descriptions of the individual rooms and offices.*

An example interaction using the program would be as follows:

```
Inquiry about room: 414
414 4th Floor west wing occupied by 2 people
Inquiry about room: 422
422 4th Floor east wing
Inquiry about room: 999
Sorry room not known
^D
```

Note: The user's input is indicated by **bold** type.

## 16.6 Fully qualified names and polymorphism

When using polymorphism and fully qualified names, the base class package name is used to qualify the polymorphic function or procedure. For example, the class `Building` specification could have been written as:

```
with Class_Room, Class_Room.Build;
package Class_Building is

  type Building is tagged private;
  procedure Add( The:in out Building;
                 Desc:in Class_Room.Build.P_Room );
  function About(The:in Building; No:in Positive) return String;

private
  Max_Rooms : constant := 15;
  type Rooms_Index is range 0 .. Max_Rooms;
  subtype Rooms_Range is Rooms_Index range 1 .. Max_Rooms;
  type Rooms_Array is array (Rooms_Range) of
    Class_Room.Build.P_Room;

  type Building is tagged record
    Last      : Rooms_Index := 0;  --Last slot allocated
    Description : Rooms_Array;     --Rooms in building
  end record;
end Class_Building;
```

:

## 238 Polymorphism

Whilst the implementation would have been written as:

```
package body Class_Building is

  procedure Add( The:in out Building;
                 Desc:in Class_Room.Build.P_Room ) is
  begin
    if The.Last < Max_Rooms then
      The.Last := The.Last + 1;
      The.Description( The.Last ) := Desc;
    else
      raise Constraint_Error;
    end if;
  end Add;

  function About(The:in Building; No:in Positive) return String is
  begin
    for I in 1 .. The.Last loop
      if Class_Room.Where(The.Description(I).all) = No then
        return Class_Room.Describe(The.Description(I).all);
      end if;
    end loop;
    return "Sorry room not known";
  end About;
end Class_Building;
```

Note:     The call of the function *Describe* is written as:  
          Class\_Room.Describe(The.Description(I).all).

## 16.7 Program maintenance and polymorphism

To modify the above program so that details about executive offices in the building are also displayed would involve the following changes:

- The creation of a new derived class *Executive\_office*.
- The modification of the procedure *set\_up* so that details of the executive offices in the building are added to the collection object *watts*.

No other components of the program would need to be changed. In carrying out these modifications, the following points are evident:

- Changes are localized to specific parts of the program.
- The modifier of the program does not have to understand all the details of the program to carry out maintenance.
- Maintenance will be easier.

Thus, if a program using polymorphism is carefully designed, there can be considerable cost saving when the program is maintained/updated.

## 16.8 Downcasting

Downcasting is the conversion of an instance of a base class to an instance of a derived class. This conversion is normally impossible as extra information needs to be added to the base type object to allow it to be turned into an instance of a derived type. However, in a program it is possible to describe the access value of an instance of a derived type as the access value of the base type. This will usually occur when a heterogeneous collection is created. The data members of a heterogeneous collection, though consisting of many different types, are each defined as an access value of the base type of the collection.

The conversion from a base type to a derived type must, of course be possible. For example, the following code copies the offices in the heterogeneous array `accommodation` into the array `Offices`.

```
with Ada.Text_IO,Ada.Integer_Text_IO,Class_Room, Class_Room.Build,
     Class_Office, Class_Office.Build, Ada.Tags;
use   Ada.Text_IO,Ada.Integer_Text_IO,Class_Room, Class_Room.Build,
     Class_Office, Class_Office.Build, Ada.Tags;
procedure Main is
  Max_Rooms : constant := 3;
  type      Rooms_Index is range 0 .. Max_Rooms;
  subtype   Rooms_Range is Rooms_Index range 1 .. Max_Rooms;
  type      Rooms_Array is array ( Rooms_Range ) of P_Room;
  type      Office_Array is array ( Rooms_Range ) of Office;
  Accommodation : Rooms_Array;    --Rooms and Offices
  Offices       : Office_Array;   --Offices only
  No_Offices    : Rooms_Index;
```

```
begin
  Accommodation(1):=Build_Office(414, "4th Floor west wing", 2);
  Accommodation(2):=Build_Room (518, "5th Floor east wing");
  Accommodation(3):=Build_Office(403, "4th Floor east wing", 1);

  No_Offices := 0;
  for I in Rooms_Array'range loop
    if Accommodation(I).all'Tag = Office'Tag then
      No_Offices := No_Offices + 1;
      Offices(No_Offices) := Office(Accommodation(I).all); --
    end if;
  end loop;

  Put("The offices are:" ); New_Line;
  for I in 1 .. No_Offices loop
    Put( Describe( Offices(I) ) ); New_Line;
  end loop;

end Main;
```

*Note: The use of 'Tag to allow the selection of objects of type Office.*

This when run, will give the following results:

```
The offices are:
414 4th Floor west wing occupied by    2 people
403 4th Floor east wing occupied by    1 people
```

### 16.8.1 Converting a base class to a derived class

It is possible to convert a base class to a derived class by adding the extra data attributes to an instance of the base class. However, for this to be performed the programmer must have access to the base class components. The implication of this is that the encapsulation of the base class has been broken. In the example below, an instance of `Account` is converted to an instance of an `Account_Ltd`.

```

with Ada.Tags;
use   Ada.Tags;
procedure Main is
  Withdrawals_In_A_Week : constant Natural := 3;
  subtype Money          is Float;
  type Account           is tagged record
    Balance_Of : Money := 0.00;      --Amount in account
  end record;
  type Account_Ltd is new Account with record
    Withdrawals : Natural := Withdrawals_In_A_Week;
  end record;
  Normal       : Account;
  Restricted   : Account_Ltd;
begin
  Normal      := ( Balance_Of => 20.0 );
  Restricted := ( Normal with 4 );
  Restricted := ( Normal with Withdrawals => 4 );
end Main;

```

*Note:*     **with** is used to extend normal, an instance of an Account, into restricted, an instance of an Account\_Ltd.

The components may be named:

*restricted* := ( normal **with** withdrawals => 4 );

If there are no additional components, **with null record** is used to form the extension.

## 16.9 The observe-observer pattern

A danger in writing any program is that input and output of data values become entangled in the body of the code of the program. When this happens the program becomes less easy to maintain and will require major changes if the format of the input or output changes. By separating the input and output from the functionality of the program allows a cleaner solution to be formulated.

The model-view paradigm in essence consists of:

- An observer: An object that has responsibility for displaying a representation of another object.
- The observed: An object that has one or more observers who will display the state of the object.

In the game of noughts and crosses, for example, the observed object would be the object representing the board. The observer would be an object that has responsibility for displaying a representation of the board onto the computer screen. There could be several implementations of the observer, depending on how the information is to be displayed. For example, the observer may be implemented to display the board as a:

- Textual representation: When a console application is written.
- As a graphical image: When an application with a graphical interface is developed.

This separation of responsibility is illustrated in Figure 16.4. In which the observed object `oxo` is interrogated by the object `oxo_observer` so that it can display a graphical representation of the noughts and crosses board on the computer screen.

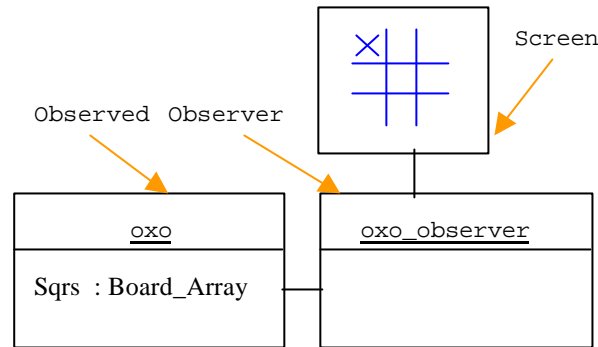


Figure 16.4 Oxo observer and observed oxo board.

The observed object `oxo` is unaware of how its representation will be displayed, whilst the observer object `oxo_observer` is unaware of how the observed object `oxo` represents and manipulates the board.

### 16.9.1 The Observer's responsibilities

An observer class for an object is required to inherit from the type `Observer` and override the method `Update` with code that will display the state of the observed object.

Method	Responsibility
Update	Display the state of the observed object.

An implementor of an `Observer` class overrides the method `Update` with a method that displays an appropriate representation of the observed object passed as a formal parameter to the method. The method `Update` is called when the state of the object being observed changes and an update of the representation of the objects state is required.

The Ada specification of this responsibility is:

```

type Observer      is tagged private;

procedure Update( The:in Observer; What:in Observable'Class );

```

*Note: The object being observed (What) is passed as its base representation. This object will need to be converted back into its true type before it can be interrogated.*

### 16.9.2 The responsibilities of the observable object

An observable class for an object is required to inherit from the type `Observable` so that the following methods may be made available to it.

Method	Responsibility
Add_Observer	Add an observer to the observable object.
Delete_Observer	Removes an observer.
Notify_Observers	If the object has changed, tells all observers to update their view of the object.
Set_Changed	Sets a flag to indicate that the object has changed.

## 242 Polymorphism

The Ada specification of this responsibility is:

```
type Observable is tagged private;

type P_Observer is access all Observer'Class;

procedure Add_Observer ( The:in out Observable;
                        O:in P_Observer );
procedure Delete_Observer( The:in out Observable;
                          O:in P_Observer );
procedure Notify_Observers( The:in out Observable'Class );
procedure Set_Changed( The:in out Observable );
```

*Note: Observers are manipulated by using their access values, rather than the instance of the observer object directly.*

### 16.9.3 Putting it all together

The complete class specifications for the observer and observable classes are combined into a single package as follows:

```
package Class_Observe_Observer is
  type Observable is tagged private;
  type Observer is tagged private;

  type P_Observer is access all Observer'Class;

  procedure Add_Observer ( The:in out Observable;
                          O:in P_Observer );
  procedure Delete_Observer( The:in out Observable;
                              O:in P_Observer );
  procedure Notify_Observers( The:in out Observable'Class );
  procedure Set_Changed( The:in out Observable );

  procedure Update( The:in Observer; What:in Observable'Class );
private
  Max_Observers : constant := 10;
  subtype Viewers_Range is Integer range 0 .. Max_Observers;
  subtype Viewers_Index is Viewers_Range range 1 .. Max_Observers;
  type Viewers_Array is array( Viewers_Index ) of P_Observer;
  type Observable is tagged record
    Viewers : Viewers_Array := ( Others => null );
    Last : Viewers_Range := 0;
    State_Changed : Boolean := True;
  end record;

  type Observer is tagged null record;
end Class_Observe_Observer;
```

*Note: Unfortunately as the methods of the classes Observer and Observable are mutually interdependent it is not possible to easily separate this package into two distinct packages.*

The implementation of the package Class\_Observe\_Observer is as follows:

```
package body Class_Observe_Observer is
```



The method `Add_Observer` adds the access value of an observer to the array `Viewers`. The exception `Constraint_Error` is raised if this operation cannot be accomplished due to lack of space.

```

procedure Add_Observer( The:in out Observable;
                        O:in P_Observer ) is
begin
    for I in 1 .. The.Last loop                --Check for empty slot
        if The.Viewers( I ) = null then
            The.Viewers( I ) := O;                --Populate
            return;
        end if;
    end loop;
    if The.Last >= Viewers_Index'Last then --Extend
        raise Constraint_Error;                -- Not enough room
    else
        The.Last := The.Last + 1;                -- Populate
        The.Viewers( The.Last ) := O;
    end if;
end Add_Observer;

```

The inverse method `Delete_Observer` removes an observer for the observed object.

```

procedure Delete_Observer( The:in out Observable;
                           O:in P_Observer ) is
begin
    for I in 1 .. The.Last loop                --For each observer
        if The.Viewers( I ) = O then            --Check if to be removed
            The.Viewers( I ) := null;
        end if;
    end loop;
end Delete_Observer;

```

When the model of the object (the observed) has changed and it is required to redisplay a representation of it the method `Notify_Observers` is called. This calls the `Update` method in each observer.

```

procedure Notify_Observers( The:in out Observable'Class ) is
begin
    for I in 1 .. The.Last loop                --For each observer
        if The.Viewers( I ) /= null then        -- call it's
            Update( The.Viewers( I ).all, The ); -- update method
        end if;
    end loop;
    The.State_Changed := True;                --
end Notify_Observers;

```

*Note: The second parameter is the observed object that is to be displayed by the Update method.*

The method `Set_Changed` simply records that the state of the observed object has changed.

```

procedure Set_Changed( The:in out Observable ) is
begin
    The.State_Changed := True;
end Set_Changed;

```

## 244 Polymorphism

The method Update is overridden by an observed with appropriate code to display the state of the observable object. This object is passed as the second parameter to the method.

```
procedure Update( The:in Observer; What:in Observable'Class ) is
begin
    null;                                --Should be overridden
end Update;

end Class_Observe_Observer;
```

*Note:* The parameter What is of type Observable'Class so that re-dispatching can take place.

## 16.10 Using the observe-observer pattern

The following is an implementation of the game of noughts and crosses using the observe-observer pattern.

### 16.10.1 The observed board object

The class Board that implements the board for the game of noughts and crosses is now defined as:

```
with Class_Observe_Observer;
use Class_Observe_Observer;
package Class_Board is

    type Board      is new Observable with private;
    type Game_State is ( Win, Playable, Draw );

    procedure Add( The:in out Board; Pos:in Integer;
                  Piece:in Character );
    function Valid( The:in Board; Pos:in Integer ) return Boolean;
    function State( The:in Board ) return Game_State;
    function Cell( The:in Board; Pos:in Integer ) return Character;
    procedure Reset( The:in out Board );
private
    subtype Board_Index is Integer range 1 .. 9;
    type Board_Array is array( Board_Index ) of Character;
    type Board is new Observable with record
        SqrS : Board_Array := ( others => ' ');  --Initialize
        Moves : Natural := 0;
    end record;
end Class_Board;
```

*Note:* Apart from inheriting from the class Observable, this code is identical to that seen in Section 8.4.1.

The implementation of the class Board is defined in the body of the package Class\_Board as follows:

```
package body Class_Board is
```

The procedure add adds a counter either the character 'X' or 'O' to the board.

```
procedure Add( The:in out Board; Pos:in Integer;
              Piece:in Character ) is
begin
    The.SqrS( Pos ) := Piece;
end Add;
```

The function `valid` returns **true** if the square selected is not occupied by a previously played counter.

```
function Valid(The:in Board; Pos:in Integer) return Boolean is
begin
    return Pos in Board_Array'Range and then
        The.Sqrs( Pos ) = ' ';
end Valid;
```

The function `Cell` returns the contents of a cell on the noughts and crosses board. This method is used to interrogate the state of the board, without having to know how the state is stored. Using this method printing of the state of the board can be separated from the code that manipulates the board.

```
function Cell(The:in Board; Pos:in Integer) return Character is
begin
    return The.Sqrs( Pos );
end Cell;
```

The procedure `Reset` sets the state of the board back to its initial state.

```
procedure Reset( The:in out Board ) is
begin
    The.sqrs := ( others => ' ');    --All spaces
    The.moves := 0;                --No of moves
end reset;
```

The function `state` returns the current state of the board.

```
function State( The:in Board ) return Game_State is
    subtype Position is Integer range 1 .. 9;
    type Win_Line is array( 1 .. 3 ) of Position;
    type All_Win_Lines is range 1 .. 8;
    Cells: constant array ( All_Win_Lines ) of Win_Line :=
        ( (1,2,3), (4,5,6), (7,8,9), (1,4,7),
          (2,5,8), (3,6,9), (1,5,9), (3,5,7) ); --All win lines
    First : Character;
begin
    for Pwl in All_Win_Lines loop    --All Pos Win Lines
        First := The.Sqrs( Cells(Pwl)(1) ); --First cell in line
        if First /= ' ' then        -- Looks promising
            if First = The.Sqrs(Cells(Pwl)(2)) and then
                First = The.Sqrs(Cells(Pwl)(3)) then return Win;
            end if;
        end if;
    end loop;
    if The.Moves >= 9 then          --Check for draw
        return Draw;                -- Board full
    else
        return Playable;            -- Still playable
    end if;
end State;

end Class_Board;
```

**16.10.2 An observer for the class Board**

The specification for the class `Display_Board` that will display a representation of the board contains the single method `Update` that has responsibility for displaying a representation of the board on a text output device.

```
with Class_Observe_Observer, Class_Board, Ada.Text_IO;
use Class_Observe_Observer, Class_Board, Ada.Text_IO;
package Class_Display_Board is

    type Display_Board is new Observer with private;

    procedure Update( The:in Display_Board; B:in Observable'Class );
private
    type Display_Board is new Observer with record
        null;
    end record;
end Class_Display_Board;
```

The implementation of the class `Display_Board` is shown below. The main point of interest is the conversion of the second parameter `B` into an instance of a `Board`. This is required as the object is passed as an instance of the class `Observable`. This down conversion will be checked to make sure that this object is an instance of `Board`.

```
package body Class_Display_Board is

    procedure Update( The:in Display_Board; B: in Observable'Class ) is
    begin
        for I in 1 .. 9 loop
            Put( Cell( Board(B), I ) );      --Its really a Board
            case I is                       --after printing counter
                when 3 | 6 =>                -- print Row Separator
                    New_Line; Put( "-----" ); --
                    New_Line;
                when 9 =>                    -- print new line
                    New_Line;
                when 1 | 2 | 4 | 5 | 7 | 8 => -- print Col separator
                    Put( " | " );
            end case;
        end loop;
    end Update;

end Class_Display_Board;
```

**16.10.3 The driver code for the program of nought and crosses**

The driver program for the game of noughts and crosses follows the same style as seen earlier in Section 8.4.5.

```
with Class_Board, Class_Display_Board,
    Ada.Text_IO, Ada.Integer_Text_IO, Class_Observe_Observer;
use   Class_Board, Class_Display_Board,
    Ada.Text_IO, Ada.Integer_Text_IO, Class_Observe_Observer;
procedure Main is
  Player : Character;           --Either 'X' or 'O'
  Game   : Board;              --An instance of Class Board
  Move   : Integer;            --Move from user
begin
  Player := 'X';                --Set player
```

An instance of an observer of the board is passed to the method Add\_Observer so that it can be displayed.

```
Add_Observer( Game, new Display_Board );
```

*Note: As a pointer to the object is required, a dynamic instance of the object is created for simplicity of code. Naturally, this could have been done by taking the access value of a non-dynamic instance.*

The code for the logic of the game asks each player in turn for a move, when a player has entered a valid move, the method Notify\_Observers is used to request a display of the new state of the board.

```
while State( Game ) = Playable loop      --While playable
  Put( Player & " enter move (1-9) : "); -- move
  Get( Move ); Skip_Line;                -- Get move
  if Valid( Game, Move ) then            --Valid
    Add( Game, Move, Player );           -- Add to board
    Notify_Observers( Game );
    case State( Game ) is                --Game is
      when Win =>
        Put( Player & " wins" );
      when Playable =>
        case Player is                  --Next player
          when 'X' => Player := 'O'; -- 'X' => 'O'
          when 'O' => Player := 'X'; -- 'O' => 'X'
          when others => null;          --
        end case;
      when Draw =>
        Put( "It's a draw " );
      end case;
    New_Line;
  else
    Put( "Move invalid" ); New_Line;     --for board
  end if;
end loop;
New_Line(2);
end Main;
```

## 16.11 Self-assessment

- What is the difference between static and dynamic binding?
- What is an object's tag?
- What is a heterogeneous collection of objects? How are heterogeneous collections of objects created and used in Ada?
- What is a view conversion? Why are view conversions required?

- How does the use of polymorphism help in simplifying program maintenance?
- Can you convert a derived class to a base class? Can you convert a base class to a derived class? Are these conversions safe? Explain your answer.

## 16.12 Exercises

Construct the following:

- The class `Executive_Office` which will extend a normal office by including biographical details about the occupants. For example, 'Ms Miranda Smith, Programming manager'.
- A new information program for a building which will include details about rooms, offices and executive offices. You should try and re-use as much code as possible.
- A program to record transactions made on different types of bank account. For example, the program should be able to deal with at least the following types of account:
  - A deposit account on which interest is paid.
  - An account on which no interest is paid and the user is not allowed to be overdrawn.

# 17 Containers

This chapter describes the implementation and use of container objects. A container object is a store for objects created in a program. The container will allow a programmer a greater flexibility in storing data items than Ada's array construct.

## 17.1 List object

A list is a container on which the following operations may be performed:

- Insert a new object into the list at any point.
- Delete an existing object from the list.
- Iterate through the objects held in the list in either a forward or reverse direction.

*Note: The number of items held in the list is dependent purely on available storage.*

The list object is based on pointer semantics. A 'pointer' is used in this context as an iterator which steps through the elements of the list. For example, a list of three integers and an iterator on the list is illustrated in Figure 17.1.

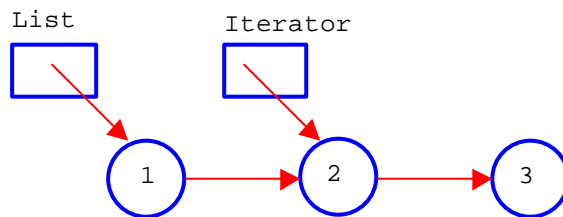


Figure 17.1 A list and its iterator.

Figure 17.2 shows the same list after inserting 99 before the current position of the iterator.

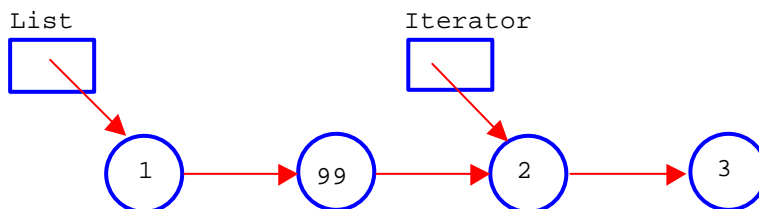


Figure 17.2 After inserting 99 into the container.

A demonstration program to show the capabilities of the list is illustrated below. In this demonstration program, a list is filled with the numbers 1 .. 10. The strategy for filling the list is to insert the numbers in reverse order into the list. The insert function inserts an item before the current position of the iterator.

After the list has been filled, it is then printed, using the iterator to move from the first item in the list to the list.

```

with Class_List;
pragma Elaborate_All( Class_List );
package Class_List_Nat is new Class_List(Natural);

with Class_List_Nat, Class_List.Iterator;
pragma Elaborate_All( Class_List_Nat, Class_List.Iterator );
package Class_List_Nat_Iterator is new Class_List_Nat.Iterator;

with Ada.Text_Io, Ada.Integer_Text_Io, Class_List_Nat,
     Class_List_Nat_Iterator;
use   Ada.Text_Io, Ada.Integer_Text_Io,
     Class_List_Nat, Class_List_Nat_Iterator;
procedure Main is
  Numbers      : List;
  Numbers_It   : List_Iter;
  Value        : Integer;
begin
  Value := 1;
  While Value <= 10 loop
    Last( Numbers_It, Numbers );           --Set iterator Last
    Next( Numbers_It );                   --Move beyond last
    Insert( Numbers_It, Value );           --Insert before
    value := Value + 1;                   --Increment
  end loop;

```

```

First(Numbers_It,Numbers);                --Set to start
while not Is_End( Numbers_It ) loop       --Not end of list
  Put( Deliver(Numbers_It) , Width=>3);    -- Print
  Next( Numbers_It );                     --Next item
end loop;
New_Line;
end Main;

```

*Note:* When an instance of this list is created there is no need to specify the number of items that will be held in the list.

The list `Numbers` has an iterator `Numbers_It` which is set initially to point to the last element of the list by:

```

Last( Numbers_It, Numbers );              --Set iterator Last

```

The iterator is then moved beyond the last item so that a new number may be inserted at the end of the list.

```

Next( Numbers_It );                       --Move beyond last
Insert( Numbers_It, Value );               --Insert before

```

In printing the numbers in the list the iterator `Numbers_It` is moved through the elements of the list. The function `Is_End` delivers true when the end of the list is reached. The procedure `Next` moves the iterator to the potentially next item in the list. If the current item 'pointed at' by the iterator in the list is the last item then a call to `Is_End` will now deliver true. The current item that the iterator is 'pointing at' is delivered using the function `Deliver`.

The code to print the contents of the list is:

```

First(Numbers_It,Numbers);                --Set to start
while not Is_End( Numbers_It ) loop       --Not end of list
  Put( Deliver(Numbers_It) , Width=>3);    -- Print
  Next( Numbers_It );                     --Next item
end loop;

```



This mirrors closely the mechanism used to access sequentially the elements of an array in Ada.

The list is implemented as a generic class `list` and its generic child `Iterator`. The two classes allow the elaboration of:

- The object `numbers` which is the list of natural numbers.
- The object `num_it`, an iterator which is used to step through the objects held in the list.

### 17.1.1 List vs. array

Criteria	List	Array
The number of items held can be increased at run-time.	√	×
Deletion of an item leaves no gap when the items are iterated through.	√	×
Random access is very efficient.	×	√

*Note:* Ada array's bounds are fixed once the declaration is elaborated.

## 17.2 Methods implemented in a list

The methods that are implemented in an instance of the class `list` are as follows:

Method	Responsibility
<code>Initialize</code>	Initialize the container.
<code>Finalize</code>	Finish using the container object.
<code>Adjust</code>	Used to facilitate a deep copy.
<code>=</code>	Comparison of a list for equality.

*Note:* A full explanation of `adjust` can be found in Section 17.3.

Whilst the methods that are implemented in an instance of the class `List_Iter` are:

Method	Responsibility
<code>Initialize</code>	Initialize the iterator.
<code>Finalize</code>	Finish using the iterator object.
<code>Deliver</code>	Deliver the object held at the position indicated by the iterator.
<code>First</code>	Set the current position of the iterator to the first object in the list.
<code>Last</code>	Set the current position of the iterator to the last object in the list.
<code>Insert</code>	Insert into the list an object before the current position of the iterator.
<code>Delete</code>	Remove and dispose of the object in the list which is specified by the current position of the iterator.
<code>Is_end</code>	Deliver true if the iteration on the container has reached the end.

<code>Next</code>	Move to the next item in the container and make that the current position.
<code>Prev</code>	Move to the previous item in the container and make that the current position.

### 17.2.1 Example of use

The following program illustrates the use of a list. In this program natural numbers are read and inserted in ascending order into a list. The contents of the list are then printed.

## 252 Containers

The strategy used for inserting individual numbers in ascending order into the list is as follows:

- Search through the list to find the position of the first number in the list that has a value greater than the number to be inserted.
- The new number is then inserted into the list before this number. Remember that insertions are always done before the current item.

```
with Class_List;
pragma Elaborate_All( Class_List );
package Class_List_Nat is new Class_List(Natural);

with Class_List_Nat, Class_List.Iterator;
pragma Elaborate_All( Class_List_Nat, Class_List.Iterator );
package Class_List_Nat_Iterator is new Class_List_Nat.Iterator;

with Ada.Text_IO, Ada.Integer_Text_IO, Class_List_Nat,
     Class_List_Nat_Iterator;
use Ada.Text_IO, Ada.Integer_Text_IO,
     Class_List_Nat, Class_List_Nat_Iterator;
```

```
procedure Main is
  Numbers      : List;
  Numbers_It   : List_Iter;
  Num, In_List : Natural;
begin
  First( Numbers_It, Numbers );           --Setup iterator

  while not End_Of_File loop              --While data
    while not End_Of_Line loop
      Get(Num);                           --Read number
      First(Numbers_It, Numbers);          --Iterator at start
      while not Is_End( Numbers_It ) loop --scan through list
        In_List := Deliver(Numbers_It);
        exit when In_List > Num;           --Exit when larger no.
        Next( Numbers_It );               --Next item
      end loop;
      Insert( Numbers_It, Num );           -- before curent number
    end loop;
    Skip_Line;                            --Next line
  end loop;
```

The list is printed out by the following code:

```
Put("Numbers sorted are: ");
First(Numbers_It, Numbers);           --Set at start
while not Is_End( Numbers_It ) loop
  In_List := Deliver( Numbers_It );    --Current number
  Put( In_List ); Put(" ");           -- Print
  Next( Numbers_It );                 --Next number
end loop;
New_Line;
end Main;
```

Which when run with the following data:

```
10 8 6 2 4
```

will produce the following results:

Numbers sorted are:	2	4	6	8	10
---------------------	---	---	---	---	----

### 17.3 Specification and implementation of the list container

The specification of the container list is split between a parent package `Class_list` which contains details of the container and a child package `Class_list.Iterator` which contains details of the iterator. The specification for the container is:

```
with Ada.Finalization, Unchecked_Deallocation;
use Ada.Finalization;
generic
  type T is private;
package Class_List is
  type List is new Controlled with private;

  procedure Initialize( The:in out List );
  procedure Initialize( The:in out List; Data:in T );
  procedure Finalize( The:in out List );
  procedure Adjust( The:in out List );
  function "=" ( F:in List; S:in List ) return Boolean;
private
  type Node;
  type P_Node is access all Node;
  type Node is record
    Prev : P_Node;
    Item : T;
    Next : P_Node;
  end record;
  type List is new Controlled with record
    First_Node : aliased P_Node := null;
    Last_Node : aliased P_Node := null;
  end record;
end Class_List;
```

The implementation of the List container object uses a linked list to hold the data items. This data structure will allow for a possibly unlimited number of data items to be added as well as the ability to add or remove items from any point in the list. When an instance of the container `colours` holds three items (Red, Green, Blue) the data structure representing the data would be as shown in Figure 17.3.

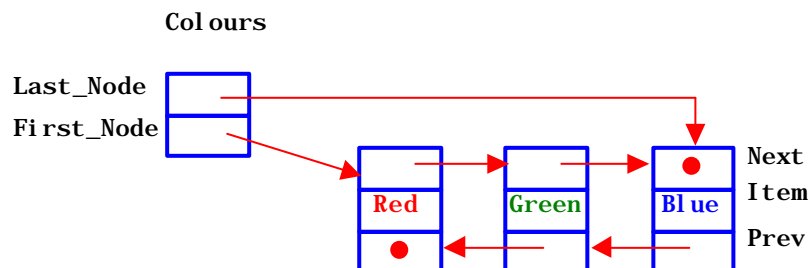


Figure 17.3 Object `colours` holding the three colours Red, Green and Blue.

*Note:* The object `Colours` holds a pointer to the root of the linked list. Adding items to the list is performed by the iterator.

## 254 Containers

The implementation of the class `list` is shown below. In the implementation the internal procedure `Release_Storage` is used to release all of the storage of the list. This procedure uses the internal procedure `Dispose_Node` to actually release individual elements of storage.

```
package body Class_List is

  procedure Dispose_Node is
    new Unchecked_Deallocation( Node, P_Node );

  procedure Release_Storage( The:in out List ) is
    Cur : P_Node := The.First_Node; --Pointer to curr node
    Tmp : P_Node;                    --Node to dispose
  begin
    while Cur /= null loop          --For each item in list
      Tmp := Cur;                   --Item to dispose
      Cur := Cur.Next;              --Next node
      Dispose_Node( Tmp );          --Dispose of item
    end loop;
  end Release_Storage;
```

The overloaded procedures `Initialize` set up either an empty list or a list of one item. The first version of `Initialize` will be automatically called whenever an instance of `List` is elaborated. Remember that `List` is a controlled type.

```
procedure Initialize( The:in out List ) is
begin
  The.First_Node := null; --Empty list
  The.Last_Node  := null; --Empty list
end Initialize;

procedure Initialize( The:in out List; Data:in T ) is
begin
  The.First_Node := new Node'(null, Data, null);
  The.Last_Node  := The.First_Node;
end Initialize;
```

The procedure `Finalize`, which is called when an instance of a `List` goes out of scope, releases any storage used in holding objects in the list. This process is decomposed into the internal procedure `Release_Storage` which performs the actual return of the storage as it iterates along the linked list.

```
procedure Finalize( The:in out List ) is
begin
  if The.First_Node /= null then
    Release_Storage( The );
    The.First_Node := null;
  end if;
end Finalize;
```

When an instance of class `List` is assigned, only the direct storage contained in the **record** list will be copied. This will not physically duplicate the storage contained in the list, but only copy the pointers to the list. When a controlled object is assigned, the procedure `Adjust` is called after the assignment has been made. The procedure `Adjust` is used to perform any additional actions required on an assignment. The exact effect of assigning a controlled object is as follows:

Assignment of controlled objects	Actions that take place on assignment
<code>A := B;</code>	<code>Anon := B;</code> <code>Adjust( Anon );</code> <code>Finalize( A );</code> <code>A := Anon;</code> <code>Adjust( a );</code> <code>Finalize( Anon );</code>

Action on assignment	Commentary
<code>Anon := B</code>	Make a temporary anonymous copy anon.
<code>Adjust(Anon);</code>	Adjustments required to be made after copying the direct storage of the source object B to Anon.
<code>Finalize(A);</code>	Finalize the target of the assignment.
<code>A := Anon;</code>	Perform the physical assignment of the direct components of the anon object.
<code>Adjust(A);</code>	Adjustments required to be made after copying the direct storage of the Anon object.
<code>Finalize(Anon);</code>	Finalize the anonymous object Anon.

*Note: If the object's storage does not overlap, which will be the usual case, then the compiler may implement the following optimization:*

*`Finalize(A); A := B; Adjust(A);`*

*Look at the effect of the assignment `A := A;` to see why this optimization may not be performed when the object's storage overlaps.*

*If the source and target are the same, then the operation may be skipped.*

The procedure `Adjust` is used to create a new duplicate copy of the storage in the list. The contents of the target in the assignment are updated to point to this newly created copy. Hence, there are now two identical copies of the linked list:

- The linked list in the Target.
- The linked list in the Source .

```

procedure Adjust( The:in out List ) is
    Cur : P_Node := The.First_Node;  --Original list
    Lst : P_Node := null;             --Last created node
    Prv : P_Node := null;             --Previously created node
    Fst : P_Node := null;             --The first node

    begin
        while Cur /= null loop
            Lst := new Node'( Prv, Cur.Item, null );
            if Fst = null then Fst := Lst; end if;
            if Prv /= null then Prv.Next := Lst; end if;
            Prv := Lst;
            Cur := Cur.Next;           --Next node
        end loop;
        The.First_Node := Fst;        --Update
        The.Last_Node := Lst;
    end Adjust;

```

When comparing two lists, the physical storage of the list needs to be compared, rather than the access values which point to the storage for the list. Remember, two lists may contain equal contents yet be represented by different physical lists.

```
function "=" ( F:in List; S:in List ) return Boolean is
  F_Node : P_Node := F.First_Node; --First list
  S_Node : P_Node := S.First_Node; --Second list
begin
  while F_Node /= null and S_Node /= null loop
    if F_Node.Item /= S_Node.Item then
      return False; --Different items
    end if;
    F_Node := F_Node.Next; S_Node := S_Node.Next;
  end loop;
  return F_Node = S_Node; --Both NULL if equal
end "=";
```

Note: When = is overloaded, /= is also overloaded with the definition of **not** =. This is true for = which returns a Boolean value.

### 17.3.1 The list iterator

The specification for the iterator, which is implemented as a child package of the package Class\_List, is:

```
generic
package Class_List.Iterator is

  type List_Iter is limited private;

  procedure First( The:in out List_Iter; L:in out List );
  procedure Last( The:in out List_Iter; L:in out List );

  function Deliver( The:in List_Iter) return T;
  procedure Insert( The:in out List_Iter; Data:in T );
  procedure Delete( The:in out List_Iter );
  function Is_End( The:in List_Iter ) return Boolean;
  procedure Next( The:in out List_Iter );
  procedure Prev( The:in out List_Iter );
private
  type P_P_Node is access all P_Node;
  type List_Iter is record
    Cur_List_First: P_P_Node := null; --First in chain
    Cur_List_Last : P_P_Node := null; --Last in chain
    Cur_Node      : P_Node   := null; --Current item
  end record;
end Class_List.Iterator;
```

Note: The child package *Class\_List.Iterator* must be generic as its parent is generic.

When the iterator `Colours_It` has been set at the start of the list `Colours`, the resulting data structure is as illustrated in Figure 17.4.

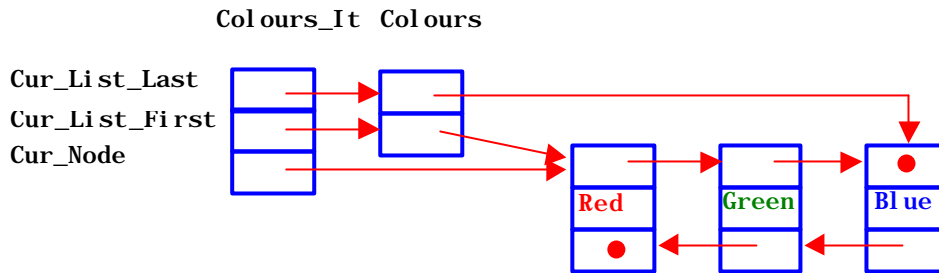


Figure 17.4 The interrelationship between the two objects `colours` and `colours_it`.

*Note:* The iterator holds pointers to the root, current and previous positions in the container.

In the implementation of the iterator for the list, the procedure, `first` and `last` set pointers in the iterator to the first or last object in the list respectively.

```
package body Class_List.Iterator is
  procedure Dispose_Node is
    new Unchecked_Deallocation( Node, P_Node );
  procedure First( The:in out List_Iter; L:in out List ) is
  begin
    The.Cur_Node      := L.First_Node;      --Set to first
    The.Cur_List_First:= L.First_Node'Unchecked_Access;
    The.Cur_List_Last := L.Last_Node'Unchecked_Access;
  end First;
  procedure Last( The:in out List_Iter; L:in out List ) is
  begin
    The.Cur_Node      := L.Last_Node;      --Set to last
    The.Cur_List_First:= L.First_Node'Unchecked_Access;
    The.Cur_List_Last := L.Last_Node'Unchecked_Access;
  end Last;
```

*Note:* The use of 'Unchecked\_Access to deliver the access value of the positions of the first and last access values of items in the list.

The access values of the first and last nodes in the list are recorded in the iterator so that they may be updated should an insertion or deletion take place at the start or the end of the list.

The procedure `Deliver` returns a copy of the current item pointed at by the iterator.

```
function Deliver( The:in List_Iter ) return T is
begin
  return The.Cur_Node.Item; --The current item
end Deliver;
```

*Note:* It is an error to try and deliver the contents of a non-existent element of the list.

The code for `Insert` is complex due to the necessity of handling insertion at different places in the linked list. In particular, the list's access values to the physical storage of the list will need to be updated. Remember, the iterator only knows about the current position in the list.

In the implementation of `Insert` there are four distinct cases to handle when a data item is inserted. This is summarized in the table below:

Position	Commentary
On an empty list	Will need to update the list's access values <code>Cur_List_First</code> and <code>Cur_List_Last</code> as well as update the current position <code>Cur_Node</code> in the iterator.
Beyond the last item in the list	Will need to update the list's access value <code>Cur_List_Last</code> as well as update the current position <code>Cur_Node</code> in the iterator.
Before the first item	Will need to update the list's access value <code>Cur_List_First</code> .
In the middle of the list	No updating required to the list's access values nor the current position of the iterator.

The implementation of the insert procedure is as follows:

```

procedure Insert( The:in out List_Iter; Data:in T ) is
    Tmp      : P_Node;
    Cur      : P_Node := The.Cur_Node;  --Current element
    First    : P_P_Node := The.Cur_List_First;
    Last     : P_P_Node := The.Cur_List_Last;
begin
    if Cur = null then                --Empty or last item
        if First.all = null then      -- Empty list
            Tmp := new Node'( null, Data, null );
            First.all := Tmp;
            Last.all  := Tmp;
            The.Cur_Node := Tmp;
        else                          -- Last
            Tmp := new Node'( Last.all, Data, null );
            Last.all.Next := Tmp;
            Last.all      := Tmp;
            The.Cur_Node := Tmp;
        end if;
    else
        Tmp := new Node'( Cur.Prev, Data, Cur );
        if Cur.Prev = null then        --First item
            First.all := Tmp;
        else
            Cur.Prev.Next := Tmp;
        end if;
        Cur.Prev := Tmp;
    end if;
end Insert;

```



In the implementation of `Delete` there are two different pointers to fix: the forward pointer and the previous pointer in the linked list. Each of these cases leads to further specializations depending on whether the object deleted is the first, last or middle object in the list.

```

procedure Delete( The:in out List_Iter) is
  Cur   : P_Node := The.Cur_Node;  --Current element
  First : P_P_Node := The.Cur_List_First;
  Last  : P_P_Node := The.Cur_List_Last;
begin
  if Cur /= null then                --Something to delete
    if Cur.Prev /= null then          --Fix forward pointer;
      Cur.Prev.Next := Cur.Next;    -- Not first in chain
    else
      First.all := Cur.Next;        -- First in chain
      if First.all = null then
        Last.all := null;          -- Empty list
      end if;
    end if;
    if Cur.Next /= null then        --Fix backward pointer;
      Cur.Next.Prev := Cur.Prev;    -- Not last in chain
    else
      Last.all := Cur.Prev;          -- Last in chain
      if Last.all = null then
        First.all := null;          -- Empty list
      end if;
    end if;
  end if;

```

```

    if Cur.Next /= null then          --Fix current pointer
      The.Cur_Node := Cur.Next;      -- next
    elsif Cur.Prev /= null then
      The.Cur_Node := Cur.Prev;      -- previous
    else
      The.Cur_Node := null;          -- none empty list
    end if;
    Dispose_Node( Cur );              --Release storage
  end if;
end Delete;

```

The function `Is_End` returns true when the iterator is moved beyond the end of the list, or beyond the start of the list.

```

function Is_End( The:in List_Iter ) return Boolean is
begin
  return The.Cur_Node = null;        --True if end
end Is_End;

```

The procedure `Next` and `Prev` move the iterator on to the next / previous item in the list. If the iterator is not currently pointing at an item, the iterator is unmodified. The end of the list is indicated by the iterator pointing to a `null` value. By inspecting the list this case can be distinguished from the case of an empty list.

```

procedure Next( The:in out List_Iter ) is
begin
    if The.Cur_Node /= null then           --
        The.Cur_Node := The.Cur_Node.Next;   --Next
    end if;
end Next;

procedure Prev( The:in out List_Iter ) is
begin
    if The.Cur_Node /= null then           --
        The.Cur_Node := The.Cur_Node.Prev;   --Previous
    end if;
end Prev;

end Class_List.Iterator;

```

*Note: If you move the iterator to beyond the first element with `prev` then it is your responsibility to reset the iterator's position. The class `list` will consider the position at the end of the list.*

### 17.3.2 Relationship between a list and its iterator

The list on which the iterator navigates must be writable. This is because the iterator may be used to insert or delete an item in the list. Another solution would have been to have two distinct iterators for read and write operations on the list.

## 17.4 Limitations of the list implementation

A limitation of this implementation is that a list object is physically duplicated when it is assigned. This is referred to as a deep copy of an object. A deep copy of an object can involve the use of considerable time and storage space.

There are two options for the implementation of assignment. These options are summarized in the table below:

Type of copy	Commentary
Deep copy	The whole physical data structure is duplicated.
Shallow copy	Only the pointer held directly in the object is duplicated.

For example, consider the data structure `Original` representing a list of colours held as a linked list. This is illustrated in Figure 17.5 which shows the memory layout for the list container `Original` which holds the three colours Red, Green and Blue.

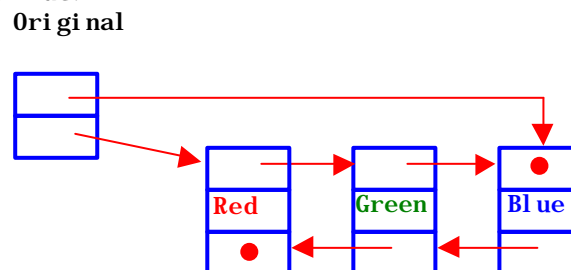


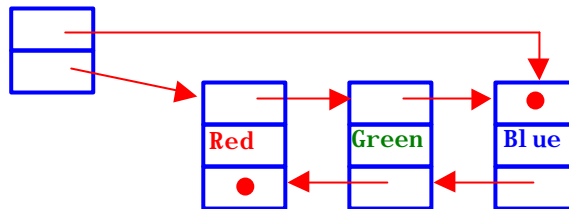
Figure 17.5 Illustration of the memory layout for a linked list of three colours.

A deep copy of this structure:

```
List original, copy;
copy := original;    -- Deep copy
```

would give the memory layout as illustrated in Figure 17.6.

Original



Copy

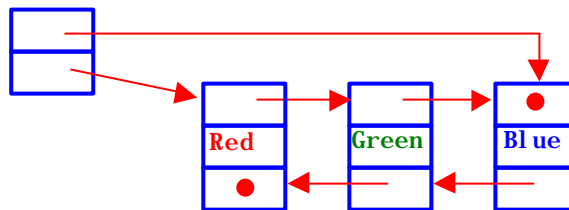


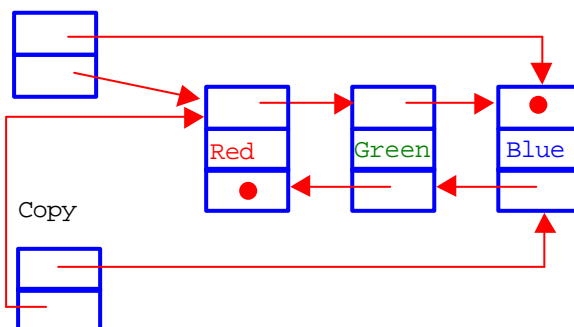
Figure 17.6 Effect of the deepCopy := Original;.

A shallow copy :

```
List original, copy;
copy := original;    -- Shallow copy
```

would produce the memory layout as illustrated in Figure 17.7.

Original



Copy



Figure 17.7 Effect of the shallow copy copy := original;

The major problem with the shallow copy is that serious errors will occur in a program if a change to the original data structure is made. This is because the copy will change as well. Worse, if the storage for the Original object is released, then the object Copy will be unsafe as it now points to storage that is no longer considered active by the Ada run-time system.

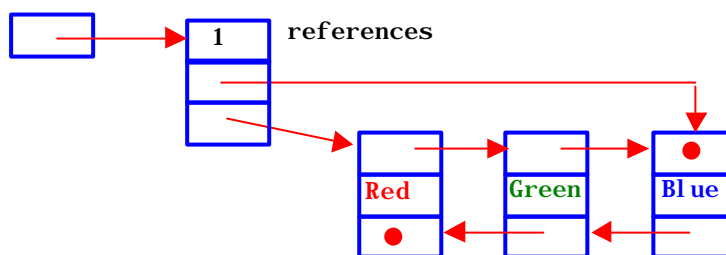
Ada's assignment statement performs a bit by bit copy of the source to the destination, which is a shallow copy. If a deep copy is required, then the assignment operator must be overloaded with a new meaning. The solution taken in the class `List` was to make assignment and comparison of a container perform a deep copy, and a shallow equality operation.

## 17.5 Reference counting

One solution to the problems encountered with a shallow copy is to implement a reference counting scheme. In a reference counting scheme, an additional component is held which is the number of active references to the data structure. A consequence of using this scheme is that additional code needs to be executed on an assignment. For example, the previously described list would be stored as illustrated in Figure 17.8.

Figure 17.8 A reference counted list.

**Original**



*Note:* The root of the list now contains the number of references that are made to this data structure.

When a shallow copy is made, for example, with the assignment:

```
copy := original;
```

the resulting data structure will be as illustrated in Figure 17.9.

**Original**

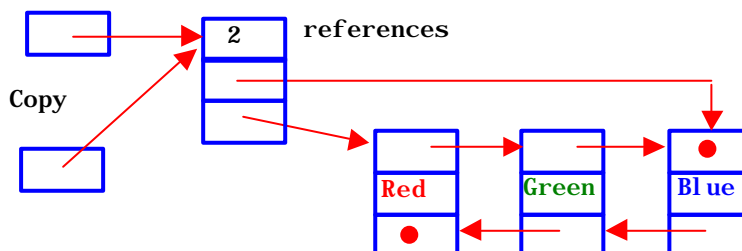


Figure 17.9 Two objects sharing the same physical storage.

The actions that take place for the shallow copy `'copy := original;'` are:

- If the objects overlap (share the same storage):

Action on assignment	Commentary
Anon := Original;	Perform the assignment: 'Anon := Original;'
Adjust(Anon);	Increment the reference count for the object Anon.
Finalize(Copy);	Decrement by one the reference count for the object Copy. If this is now zero, release the storage that the object Copy points to.
Copy := Anon;	Perform the assignment: 'Copy := Original;'
Adjust(Copy);	Increment the reference count for the object Copy.
Finalize(Anon);	Decrement by one the reference count for the object Anon. If this is now zero release the storage that the object Anon points to.

- If the objects do not overlap, the following optimization may be performed:

Action on assignment	Commentary
Finalize(Original);	Decrement by one the reference count for the object Original. If this is now zero, release the storage that the object Original points to.
Copy := Original;	Perform the assignment: 'Copy := Original;'
Adjust(Copy);	Increment the reference count for the object Copy.

*Note: The compiler may generate no code if the target and the source are the same.*

When a reference counted item is passed as a parameter to a function or procedure by value, for example:

Procedure	Call of procedure
<pre> <b>procedure</b> Put(L:in List) <b>is</b> <b>begin</b>   .... <b>end</b> Put; </pre>	<pre> <b>declare</b>   Colours : List; <b>Begin</b>   Put( Colours ); <b>end;</b> </pre>

then only the following actions are performed:

Put( Colours )	Commentary
L := Colours;	Perform the assignment of colours to the formal parameter: 'L := colours;'
Adjust(L);	Increment the reference count for the actual parameter L.

*Note: When the procedure put is exited, finalize will be called on the formal parameter l.*

## 17.6 Implementation of a reference counting scheme

A wrapper class for an object that will implement a reference counting has the following major methods:

Method	Responsibility
Initialize	Initialize an object contained in the reference counting wrapper object.
Deliver	Deliver the contents of the object wrapped by the reference counting object.
Deliver_Ref	Deliver the access value of the object wrapped by the reference counting object.
Unique	Make the target of an assignment a unique copy.

An example of the use of an instantiation `Class_Account_Rc` of this generic class is shown below. Firstly, two wrapped instances of the class `Account` are elaborated.

```
Original, Copy : Class_Account_Rc.Object;
```

The normal assignment operation now delivers a shallow copy:

```
Copy := Original;                                --Shallow copy
```

To make the object `Copy` not share storage with the object `Original` the method `Unique` is used as follows:

```
Unique( Copy );                                --Deep copy
```

### 17.6.1 Ada specification

The Ada specification for this generic class is as follows:

```
with Ada.Finalization;
use Ada.Finalization;
generic
  type T is private;           --The type
  Null_Value:in T;             --Identity element
package Class_Object_Rc is
  type Object is new Controlled with private;
  type P_T is access all T;

  procedure Initialize( The:in out Object );
  procedure Initialize( The:in out Object; Data:in T );
  procedure Finalize( The:in out Object );
  procedure Adjust( The:in out Object );
  function Deliver( The:in Object) return T;
  function Deliver_Ref( The:in Object) return P_T;
  procedure Unique( The:in out Object);
private
  procedure Build_Storage ( The:in out Object; Value:in T );
  procedure Release_Storage( The:in out Object );

  type Descriptor;
  type P_Descriptor is access all Descriptor;

  type Descriptor is record
    Refs    : Natural;          --References to this data
    Object  : aliased T;         --The physical data
  end record;

  type Object is new Controlled with record
    P_Desc : P_Descriptor:= null; --Descriptor for a number
  end record;

end Class_Object_Rc;
```

So that an object can be reference counted, a new class is instantiated with parameters of the object's type and the null value for the type. For example, to reference count instances of the class `Account`, the following instantiation would be made:

```
with Class_Account;
package Pack_Consts is
  Null_Account: Class_Account.Account;
end Pack_Consts;

with Pack_Consts, Class_Object_Rc, Class_Account;
package Class_Rc_Account is
  new Class_Object_Rc(Class_Account.Account,
    Pack_Consts.Null_Account);
```

*Note:* In this instantiation, the null value for the type is simply an instance of `Account`, which contains a zero balance.

**17.6.2 Ada implementation**

The implementation of the class is shown below. In this, the internal procedure `Build_Storage` is responsible for allocating a new storage element. Whilst the internal procedure `Release_Storage` is responsible for releasing storage when its reference count is zero.

```
with Unchecked_Deallocation;
package body Class_Object_Rc is
  procedure Build_Storage ( The:in out Object; Value:in T ) is
  begin
    The.P_Desc := new Descriptor'(1,Value);
  end Build_Storage;

  procedure Dispose is
    new Unchecked_Deallocation( Descriptor, P_Descriptor );

  procedure Release_Storage( The:in out Object ) is
  begin
    The.P_Desc.Refs := The.P_Desc.Refs-1;
    if The.P_Desc.Refs = 0 then
      Dispose( The.P_Desc );
    else
      null;
    end if;
  end Release_Storage;
end;
```

The procedure `Initialize` builds storage for a record which holds the data item and the reference count for the data item. Initially this reference count will be zero.

```
procedure Initialize( The:in out Object ) is
begin
  Build_Storage( The, Null_Value );
end Initialize;

procedure Initialize( The:in out Object; Data:in T ) is
begin
  Build_Storage( The, Data );
end Initialize;
```

The procedure `finalize` is decomposed into the procedure `Release_Storage` which releases the storage for the managed data item only when the reference count goes to zero.

```
procedure Finalize( The:in out Object ) is
begin
  if The.P_Desc /= null then
    Release_Storage( The );
    The.P_Desc := null;
  end if;
end Finalize;
```



The procedure `Adjust` is automatically called whenever an assignment or implied assignment of a controlled object takes place. The reason for this is that when an assignment of an object managed by this class is made, there are now two references to the object. The procedure `adjust` has the responsibility of managing this process, which it does by increasing the reference count to the object by 1.

```
procedure Adjust( The:in out Object ) is
begin
  The.P_Desc.Refs := The.P_Desc.Refs+1;
end Adjust;
```

Remember, when an assignment of a controlled object is made, the following sequence of events occurs:

Assignment of controlled objects	Actions that take place
A := B;	<pre>Anon := B; Adjust( Anon ); Finalize( A ); A := Anon; Adjust( A ); Finalize( Anon );</pre>

*Note:* When the storage for the source and target do not overlap, this process can be optimized by the compiler to:

```
Finalize(A); A := B; Adjust(A);
```

The function `Deliver` returns a copy of the managed object

```
function Deliver( The:in Object) return T is
begin
  return The.P_Desc.Object;
end Deliver;
```

whilst the function `deliver_ref` returns an access value to the managed object.

```
function Deliver_Ref( The:in Object) return P_T is
begin
  return The.P_Desc.Object'access;
end Deliver_Ref;
```

The procedure `unique` converts the object managed by the class into a unique copy. This may involve a deep copy of the managed object.

```
procedure Unique( The:in out Object) is
  Tmp : P_Descriptor;
begin
  if The.P_Desc.Refs > 1 then
    The.P_Desc.Refs := The.P_Desc.Refs-1;
    Tmp := new Descriptor'(1,The.P_Desc.Object);
    The.P_Desc := Tmp;
  end if;
end Unique;
end Class_Object_Rc;
```

### 17.6.3 Putting it all together

The program below illustrates the use of the package `Class_Object_Rc` to provide a reference counted Account class. Firstly, the generic package `Class_Account_Rc` is used to create the specific package `Class_Account_Rc`.

```
with Class_Account;
package Pack_Consts is
  Null_Account: Class_Account.Account;
end Pack_Consts;

with Pack_Consts, Class_Object_Rc, Class_Account;
package Class_Account_Rc is
  new Class_Object_Rc(Class_Account.Account,
    Pack_Consts.Null_Account);
```

Then the class `Account_Rc` is used in the following example program to illustrates a deep and a shallow copy.

```
with Ada.Text_IO, Ada.Float_Text_IO,
     Class_Account, Class_Account_Rc, Statement;
use Ada.Text_IO, Ada.Float_Text_IO,
     Class_Account, Class_Account_Rc;
procedure Main is
  Original, Copy : Class_Account_Rc.Object;
begin
  Deposit( Deliver_Ref(Original).all, 100.00 );
  Put("copy := original; (Shallow copy)"); New_Line;
  Copy := Original;                                --Shallow copy
  Statement( Deliver_Ref(Original).all );           --The same object
  Statement( Deliver_Ref(Copy).all );               -- " "
  Put("Make copy unique (Deep copy if necessary)"); New_Line;
  Unique( Copy );                                   --Deep copy
  Deposit( Deliver_Ref(Copy).all, 20.00 );           --copy only
  Statement( Deliver_Ref(Original).all );           --Unique object
  Statement( Deliver_Ref(Copy).all );               -- " "
end Main;
```

*Note: **all** is used to de-reference the access value returned by the function `Deliver_Ref`.  
The procedure `Statement` is the previously seen procedure in Section 6.3.2 that prints a mini-statement for a bank account.*

When the above program is compiled and run the output produced is as follows::

```
Mini statement: The amount on deposit is £100.00
Mini statement: The amount on deposit is £100.00
Make copy unique (Deep copy if necessary)
Mini statement: The amount on deposit is £100.00
Mini statement: The amount on deposit is £120.00
```

## 17.7 A set

Using as a base class the class `List`, a class to represent a set can be easily created using inheritance. The class `Set` has the following responsibilities:

Method	Responsibility
<code>Put</code>	Display the contents of the set.
<code>+</code>	Form the union of two sets.
<code>Set_Const</code>	Return a set with a single member.
<code>Members</code>	Return the numbers of items in the set.

The Ada specification for the class `Set` is as follows:

```
with Class_List, Class_List.Iterator;
pragma Elaborate_All( Class_List, Class_List.Iterator );
generic
  type T is private;
  with procedure Put( Item:in T ) is <>;
  with function ">" (First,Second:in T ) return Boolean is <>;
  with function "<" (First,Second:in T ) return Boolean is <>;
package Class_Set is
  type Set is private;
  procedure Put( The:in Set );
  function "+"( F:in Set; S:in Set ) return Set;
  function Set_Const( Item: in T ) return Set;
  function Members( The:in Set ) return Positive;
private
  package Class_List_T is new Class_List(T);
  package Class_List_T_Iterator is new Class_List_T.Iterator;
  type Set is new Class_List_T.List with record
    Elements : Natural := 0;           --Elements in set
  end record;
end Class_Set;
```

*Note:* On instantiation of the class, a procedure `put` and definitions for `>` and `<` must be provided, either explicitly or implicitly.

In the implementation of the class, the procedure `put` lists in a canonical form the elements of the set.

```
with Ada.Text_IO;
use Ada.Text_IO;
package body Class_Set is
  use Class_List_T, Class_List_T_Iterator;

  procedure Put( The:in Set ) is
    It : List_Iter;
    C_The : List := List(The);
  begin
    Put("("); First( It, C_The );
    for I in 1 .. The.Elements loop
      Put( Deliver(It) ); Next( It );
      if I /= The.Elements then Put(","); end if;
    end loop;
    Put(")");
  end Put;
```

A simple merging process is used to form the union of two sets.

```

function "+" ( F:in Set; S:in Set ) return Set is
  Res_It      : List_Iter;
  F_It,S_It   : List_Iter;
  Res         : Set;
  F_List, S_List: List;
begin
  F_List := List(F); S_List := List(S);
  First( F_It, List(F_List) );
  First( S_It, List(S_List) );
  First( Res_It, List(Res) );

  while (not Is_End(F_It)) or (not Is_End(S_It)) loop
    if Is_End(F_It) then
      Next(Res_It); Insert(Res_It, Deliver(S_It));
      Next(S_It);
    elsif Is_End(S_It) then
      Next(Res_It); Insert(Res_It, Deliver(F_It));
      Next(F_It);
    elsif Deliver(F_It) < Deliver(S_It) then
      Next(Res_It); Insert(Res_It, Deliver(F_It));
      Next(F_It);
    elsif Deliver(F_It) > Deliver(S_It) then
      Next(Res_It); Insert(Res_It, Deliver(S_It));
      Next(S_It);
    elsif Deliver(F_It) = Deliver(S_It) then
      Next(Res_It); Insert(Res_It, Deliver(F_It) );
      Next(F_It); Next(S_It);
    end if;
    Res.Elements := Res.Elements + 1;
  end loop;
  return Res;
end "+";

```

*Note: The copying of an instance of a set to a list object so that it can be manipulated.*

The procedure Set\_Const returns a set with a single element, whilst the function members returns the number of elements in the set.

```

function Set_Const( Item: in T ) return Set is
  Res : Set;
begin
  Initialize( Res, Item ); Res.Elements := 1;
  return Res;
end Set_Const;

function Members( The:in Set ) return Positive is
begin
  return The.Elements;
end Members;

end Class_Set;

```

### 17.7.1 Putting it all together

The program below illustrates the use of a set to record the ingredients in a sandwich.

```

package Pack_Types is
  type Filling is ( Cheese, Onion, Ham, Tomato );
end Pack_Types;

with Ada.Text_Io, Pack_Types;
use Ada.Text_Io, Pack_Types;
procedure Put_Filling( C:in Filling ) is
begin
  Put( Filling'Image( C ) );
end Put_Filling;

with Pack_Types, Class_Set, Put_Filling;
use Pack_Types;
pragma Elaborate_All( Class_Set );
package Class_Set_Sandwich is
  new Class_Set( T => Pack_Types.Filling, Put => Put_Filling );

with Pack_Types, Ada.Text_Io, Ada.Integer_Text_Io, Class_Set_Sandwich;
use Pack_Types, Ada.Text_Io, Ada.Integer_Text_Io, Class_Set_Sandwich;
procedure Main is
  Sandwich : Class_Set_Sandwich.Set;
begin
  Sandwich := Sandwich + Set_Const(Cheese);
  Sandwich := Sandwich + Set_Const(Onion);
  Put("Contents of sandwich are : ");
  Put( Sandwich ); New_Line;
  Put("Number of ingredients is : ");
  Put( Members(Sandwich) ); New_Line;
  null;
end Main;

```

*Note: The instantiation of the class Class\_Set\_Sandwich uses the default definitions of > and < taken from the environment.  
An instantiation of a class which is inherited from Controlled must be at the library level.*

which when run, will produce the following results:

```

Contents of sandwich are : (CHEESE,ONION)
Number of ingredients is :          2

```

## 17.8 Self-assessment

- What is the purpose of an iterator?
- When inheriting from Controlled, the user can provide the following procedures:  
Finalize, Initialize, and Adjust.  
What is the purpose of these procedures?
- If A and B are controlled objects, what happens when the assignment:  
A := B;  
is made?
- What is the difference between a deep and a shallow copy?

## 272 Containers

- What are the semantics of an assignment in Ada for the following assignments:
  - (a) The assignment of an instance of an `Integer`?
  - (b) The assignment of an instance of a linked list?
- With the container `Class_List`, what would be the effect of using an iterator to the container when the storage for the container object has gone out of scope?
- What should happen to the iterator when an item is added to a container on which it is iterating?

## 17.9 Exercises

Construct the following class:

- `Class_Better_Set`  
A class to implement a better set. A set is an ordered collection of unique items. The operations allowed on a set are:
  - Forming the intersection of two sets.
  - Forming the union of two sets.
  - Forming the set difference of two sets.
  - Testing if an element is a member of the set.

# 18 Input and output

This chapter describes how input and output of objects other than Float, Integer or Character may be performed.

## 18.1 The input and output mechanism

In Ada input and output operations are strongly typed. This can cause initial problems as only a mechanism for inputting or outputting instances of:

- |   |                      |                     |
|---|----------------------|---------------------|
| • | Character and String | Ada.Text_Io         |
| • | Integer              | Ada.Integer_Text_io |
| • | Float                | Ada.Float_Text_Io   |

are explicitly provided. The full definition of these packages is given in Appendix C.

The package Ada.Text\_Io contains generic packages for outputting Float, Integer, String, Fixed or Enumeration types. For example, to output instances of the following types:

```
type Memory is range 0 .. Max_Mem;           -- Integer
type Cpu     is (I64, I32, PowerPc);          -- Enum
type Mips     is digits 8 range 0.0 .. Max_Mips; -- Float
type Clock    is delta 0.01 range 0.0 .. Max_Clock; -- Fixed
```

the following packages would need to be instantiated:

```
package Class_Mem_Io is new Ada.Text_Io.Integer_Io(Memory);
package Class_Cpu_Io is new Ada.Text_Io Enumeration_Io(Cpu);
package Class_Mips_Io is new Ada.Text_Io.Float_Io(Mips);
package Class_Clock_Io is new Ada.Text_Io.Fixed_Io(Clock);
```

*Note: Each of the generic packages has as its generic parameter the type that is to be output.*

**18.1.1 Putting it all together**

The above generic packages are used in the following program that prints out details about the internal specification of a CPU:

```
with Ada.Text_Io;
use   Ada.Text_Io;
procedure Main is
  Max_Mem      : constant := 4096;           --Mb
  Max_Mips      : constant := 12000.0;       --Mips
  Max_Clock     : constant := 4000.0;        --Clock

  type Memory is range 0 .. Max_Mem;         --Integer
  type Cpu     is (I64, I32, PowerPc);       --Enum
  type Mips     is digits 8 range 0.0 .. Max_Mips; --Float
  type Clock    is delta 0.01 range 0.0 .. Max_Clock; --Fixed

  Mc_Mem      : Memory;      --Main memory
  Mc_Cpu      : Cpu;         --Type of CPU
  Mc_Mips      : Mips;        --Raw MIPS
  Mc_Clock     : Clock;       --Clock frequency

  package Class_Mem_Io is new Ada.Text_Io.Integer_Io(Memory);
  package Class_Cpu_Io is new Ada.Text_Io Enumeration_Io(Cpu);
  package Class_Mips_Io is new Ada.Text_Io.Float_Io(Mips);
  package Class_Clock_Io is new Ada.Text_Io.Fixed_Io(Clock);
```

The body of the procedure illustrated below writes out details about the computer.

```
begin
  declare
    use Class_Mem_Io, Class_Mips_Io, Class_Clock_Io, Class_Cpu_Io;
  begin
    Mc_Mem := 512;      Mc_Cpu := I64;
    Mc_Mips := 3000.0;  Mc_Clock := 1000.0;

    Put("Memory:"); Put( Mc_Mem ); New_Line;
    Put("CPU   "); Put( Mc_Cpu ); New_Line;
    Put("Mips   "); Put( Mc_Mips ); New_Line;
    Put("Clock  "); Put( Mc_Clock ); New_Line;

    Put("Memory:"); Put( Mc_Mem, Width=>3); New_Line;
    Put("CPU   "); Put( Mc_Cpu, Width=>7, Set=>Upper_Case );
    New_Line;
    Put("Mips   "); Put( Mc_Mips, Fore=>3, Aft=>2, Exp=>0 );
    New_Line;
    Put("Clock  "); Put( Mc_Clock, Fore=>3, Aft=>2, Exp=>0 );
    New_Line;
  end;
end Main;
```

When compiled and run the output from this program is as follows:

```
Memory: 512
CPU    :I64
Mips   : 3.0000000E+03
Clock  : 1000.00
Memory:512
CPU    :I64
Mips   :3000.00
Clock  :1000.00
```



## 18.2 Reading and writing to files

The following program copies input typed in at a terminal to the file named `file.txt`. The object `Fd` is associated with the newly created file `file.txt` and is used as a file descriptor in all writing to the text file.

```
with Text_Io;
use Text_Io;
procedure Main is
  Fd      : Text_Io.File_Type;           --File descriptor
  File_Name: constant String:= "file.txt";--Name
  Ch      : Character;                  --Character read
begin
  Create( File=>Fd, Mode=>Out_File, Name=>File_Name );
  while not End_Of_File loop            --For each Line
    while not End_Of_Line loop          --For each character
      Get(Ch); Put(Fd, Ch);             --Read / Write character
    end loop;
    Skip_Line; New_Line(Fd);            --Next line / new line
  end loop;
  Close(Fd);
exception
  when Name_Error =>
    Put("Cannot create " & File_Name ); New_Line;
end Main;
```

*Note:* The exception `Name_Error` is generated if the file cannot be created.

The data in the file `file.txt` is read by the following program, which copies the contents of the file to the terminal:

```
with Text_Io;
use Text_Io;
procedure Main is
  Fd      : Text_Io.File_Type;           --File descriptor
  File_Name: constant String:= "file.txt";--Name
  Ch      : Character;                  --Character read
begin
  Open( File=>Fd, Mode=>In_File, Name=>File_Name );
  while not End_Of_File(Fd) loop        --For each Line
    while not End_Of_Line(Fd) loop      --For each character
      Get(Fd, Ch); Put(Ch);             --Read / Write character
    end loop;
    Skip_Line(Fd); New_Line;            --Next line / new line
  end loop;
  Close(Fd);
exception
  when Name_Error =>
    Put("Cannot open " & File_Name ); New_Line;
end Main;
```

*Note:* The exception `Name_Error` is generated if the file cannot be opened.

## 276 *Input and output*

The following program appends instances of `Number`, one of the integer types to the end of the file `file.txt`:

```
with Text_Io;
use Text_Io;
procedure Main is
  type Number is range 1 .. 10;
  Fd          : Text_Io.File_Type;           --File descriptor
  File_Name: constant String:= "file.txt";--Name
  package Pack_Number_Io is new Text_Io.Integer_Io( Number );
begin
  Open( File=>Fd, Mode=>Append_File, Name=>File_Name );
  for I in Number loop
    Pack_Number_Io.Put( Fd, I ); New_Line( Fd );
  end loop;
  Close(Fd);
exception
  when Name_Error =>
    Put("Cannot append to " & File_Name ); New_Line;
end Main;
```

## 18.3 Reading and writing binary data

Any instance of a type may be read and written to a file using the package `Ada.Sequential_io`. By using this package, binary images of objects may be read and written. For example, the following code writes instances of the data structure `Person` to the file `people.txt`. Firstly the `Package_Types` defines the data structure `Person` used to represent an individual person.

```
package Pack_Types is
  Max_Chars : constant := 10;
  type Gender is ( Female, Male );
  type Height_Cm is range 0 .. 300;

  type Person is record
    Name      : String( 1 .. Max_Chars ); --Name as a String
    Height    : Height_Cm := 0;           --Height in cm.
    Sex       : Gender;                  --Gender of person
  end record;

  type Person_Index is range 1 .. 3;
  subtype Person_Range is Person_Index;
  type Person_Array is array ( Person_Range ) of Person;
end Pack_Types;
```

Then the following example program writes instance of Person to the file `people.txt`.

```
with Text_Io, Pack_Types, Sequential_Io;
use Text_Io, Pack_Types;
procedure Main is
  File_Name: constant String:= "people.txt";--Name
  People   : Person_Array;
  package Io is new Sequential_Io( Person );
begin
  declare
    Fd      : Io.File_Type;          --File descriptor
  begin
    People(1) := (Name=>"Mike", Sex=>Male, Height=>183);
    People(2) := (Name=>"Corinna", Sex=>Female, Height=>171);
    People(3) := (Name=>"Miranda", Sex=>Female, Height=>74);
    Io.Create( File=>Fd, Mode=>Io.Out_File, Name=>File_Name );
    for I in Person_Range loop
      Io.Write( Fd, People(I) );
    end loop;
    Io.Close(Fd);
  exception
    when Name_Error =>
      Put( "Cannot create " & File_Name ); New_Line;
  end;
end Main;
```

*Note:* The package `Ada.Text_Io` is used to provide the definition of the exception `Name_Error`. When using the generic package `Ada.Sequential_Io`, the procedures `read` and `write` are used to perform the input and output operations.

To read back the data written to the file `people.txt` the following example program is used:

```
with Text_Io, Pack_Types, Sequential_Io;
use Text_Io, Pack_Types;
procedure Main7 is
  File_Name: constant String:= "people.txt";--Name
  People   : Person_Array;
  package Io is new Sequential_Io( Person );
begin
  declare
    Fd      : Io.File_Type;          --File descriptor
  begin
    Io.Open( File=>Fd, Mode=>Io.In_File, Name=>File_Name );
    for I in Person_Range loop
      Io.Read( Fd, People(I) );
      Put( People(I).Name );
      Put( Height_Cm'Image( People(I).Height ) );
      if People(I).Sex = Male then
        Put( " Male" );
      else
        Put( " Female" );
      end if;
      New_Line;
    end loop;
    Io.Close(Fd);
  exception
    when Name_Error =>
      Put( "Cannot open " & File_Name ); New_Line;
  end;
end Main7;
```

## 278 *Input and output*

Which when run would print the following results:

Mike	183 Male
Corinna	171 Female
Miranda	74 Female

### 18.4 Switching the default input and output streams

It is possible to switch the default input or output stream to another stream using the following procedures in the package `Ada.Text_Io`. The effect is to change the source or sink from which input and output will come from or go to when using the normal input and output procedures `put` and `get` without a `File` parameter.

Procedure	Sets the default
<code>Set_Input ( File:in File_Type )</code>	Input file descriptor.
<code>Set_OutPut( File:in File_Type )</code>	Output file descriptor.
<code>Set_Error ( File:in File_Type )</code>	Error file descriptor.

As the file descriptor is of type limited private, it may not be directly assigned. However, an access value of the file descriptor can be saved. The following functions return an access value of the standard input and output file descriptors:

Function	Returns the access
<code>Standard_Input return File_Access;</code>	Value of the input file descriptor.
<code>Standard_Output return File_Access;</code>	Value of the output file descriptor.
<code>Standard_Error return File_Access;</code>	Value of the error file descriptor.

#### 18.4.1 Putting it all together

The program first saves an access value to the original default input stream. Then it switches the default input stream to be the file `file.txt`. After reading and printing the contents of this file the default, input stream is switched back to its original value and the contents of the stream are read and written out.

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
  Fd      : Ada.Text_Io.File_Type;      --File descriptor
  P_St_Fd : Ada.Text_Io.File_Access;    --Access value of Standard
  Ch      : Character;                  --Current character
begin
  P_St_Fd := Standard_Input;            --Access value of standard fd
  Open( File=>Fd, Mode=>In_File, Name=>"file.txt" );
  Set_Input( Fd );
  while not End_Of_File loop            --For each Line
    while not End_Of_Line loop          -- For each character
      Get(Ch); Put(Ch);                  -- Read / Write character
    end loop;
    Skip_Line; New_Line;                 -- Next line / new line
  end loop;
  Close(Fd);                             --Close file
  Set_Input( P_St_Fd.all );
  while not End_Of_File loop            --For each Line
    while not End_Of_Line loop          --For each character
      Get(Ch); Put(Ch);                  --Read / Write character
    end loop;
    Skip_Line; New_Line;                 --Next line / new line
  end loop;
end Main;
```

Note: Notice how **all** has been used to de-reference the access value when using the procedure `Set_Input`.

## 18.5 Self-assessment

- What is the purpose of the package `Ada.Sequential_io`?
- How can you detect if a file does not exist in Ada?
- How might input and output in Ada be simplified for the novice user?
- How can you write an instance of a record to a file?

## 18.6 Exercises

Construct the following program:

- *Copy*  
A program to copy a file. A user should be able to run your program by typing:

```
new_copy old_file new_file
```

- *Upper case*  
A program to convert a file in upper and lower case to all upper case. A user should be able to run your program by typing:

```
to_upper_case file
```

*Note: The program should create an intermediate file then delete the original file and rename the intermediate file to the original file name. This operation should be safe.*

# 19 Persistence

This chapter shows how to create persistent objects. A persistent object will have a life-time beyond the life-time of the program that created it.

## 19.1 A persistent indexed collection

The life-time of an object in Ada depends on its declaration, but its life-time will never exist beyond that of the program. For an object to exist beyond the life-time of an individual execution of a program requires the object's state to be saved to disk, allowing the object's state to be restored in another program. The above process makes the object persistent. Normally this process is visible to a programmer.

For example, a program to print the IDC (International Dialling Code) for countries selected by a user could use a persistent object to hold IDC details for individual countries. These details could be amended by the user of the program and the changes would be retained for subsequent re-running of the program.

A program of this kind could use the persistent object `Tel_List` that is an instance of the class `Pic` (Persistent Indexed Collection). The class `Pic` implements a persistent indexed collection of data items. The index can be an arbitrary value as can the data stored with the index.

The responsibilities of the class `PIC` are as follows:

Method	Responsibility
Initialize	Initialize the object. When the object is initialized with an identity, the state of the named persistent object is restored into the object.
Finalize	If the object has an identity, save the state of the object under this name.
Add	Add a new data item to the object.
Extract	Extract the data associated with an index.
Update	Update the data associated with an index.
Set_Name	Set the identity of the object.
Get_Name	Return the identity of the object.

The package `Pack_types` contains string definitions for the `Country` and the `IDC`.

```
package Pack_Types is
  subtype Country is String(1 .. 12); --Country
  subtype Idc     is String(1 .. 6);  --International Dialling Code
end Pack_Types;
```

The Class `Tel_List` is an instantiation of the generic class `Pic`.

```
with Class_Pic, Pack_Types;
use Pack_Types;
pragma Elaborate_All( Class_Pic );
package Class_Tel_List is new Class_Pic( Country, Idc, ">" );
```

*Note: The generic class `Pic` is described in Section 19.2.*

A program to implement this telephone aid would be as follows:

```

with Ada.Text_IO, Pack_Types, Class_Tel_List;
use Ada.Text_IO, Pack_Types, Class_Tel_List;
procedure Main is
  Tel_List : Pic;
  Action    : Character;
  Name      : Country;
  Tel       : Idc;
begin
  Initialize( Tel_List, "tel_list.per" );
  while not End_Of_File loop
    begin
      Get( Action );                                --Action to perform
      case Action is
        when '+' =>                                --Add
          Get( Name ); Get( Tel );
          Add( Tel_List, Name, Tel );
        when '=' =>                                --Extract
          Get( Name );
          Extract( Tel_List, Name, Tel );
          Put( "IDC for " ); Put( Name );
          Put( " is " ); Put( Tel ); New_Line;
        when '*' =>                                --Update
          Get( Name ); Get( Tel );
          Update( Tel_List, Name, Tel );
        when others =>                             --Invalid action
          null;
      end case;
    exception
      when Not_There =>                             --Not there
        Put( "Name not in directory" ); New_Line;
      when Mainists =>                               --Exists
        Put( "Name already in directory" ); New_Line;
    end;
    Skip_Line;
  end loop;
end Main;

```

*Note:* When the object *Tel\_List* is initialized it is named with an identity which is the file used to hold its saved state. Using this file the state of the object can be restored so allowing the object to have life beyond a single program run.

## 282 Persistence

### 19.1.1 Putting it all together

When compiled with the class `Tel_list` and the package `Pack_types` an example interaction using the program would be as follows:

```
=UK                      -- Previously stored
IDC for UK               is 44
New Zealand 64         -- Add IDC for New Zealand
Sweden      46         -- Add IDC for Sweden
Portugal    3510       -- Invalid IDC
=Portugal                -- Lookup IDC for Portugal
IDC for Portugal         is 3510
Portugal      351      -- Try to add new IDC for Portugal
Name already in directory
*Portugal      351      -- Correct invalid IDC
=Portugal                -- Lookup IDC for Portugal
IDC for Portugal         is +351
```

*Note:* The user's input is indicated by **bold** type.  
The actions allowed are:

- + Add country and IDC to collection
- = Extract IDC for country
- \* Change IDC for existing country

### 19.1.2 Setting up the persistent object

The following program creates the initial persistent collection:

```
with Ada.Text_IO, Class_Tel_List;
use  Ada.Text_IO, Class_Tel_List;
procedure Main is
  Tel_List : Pic;
begin
  Put("Creating Telephone list"); New_Line;
  Set_Name( Tel_List, "tel_list.per" );
  Add( Tel_List, "Canada", "+1", " " );
  Add( Tel_List, "USA", "+1", " " );
  Add( Tel_List, "Netherlands", "+31", " " );
  Add( Tel_List, "Belgium", "+32", " " );
  Add( Tel_List, "France", "+33", " " );
  Add( Tel_List, "Gibraltar", "+350", " " );
  Add( Tel_List, "Ireland", "+353", " " );
  Add( Tel_List, "Switzerland", "+41", " " );
  Add( Tel_List, "UK", "+44", " " );
  Add( Tel_List, "Denmark", "+45", " " );
  Add( Tel_List, "Norway", "+47", " " );
  Add( Tel_List, "Germany", "+49", " " );
  Add( Tel_List, "Australia", "+61", " " );
  Add( Tel_List, "Japan", "+81", " " );
end Main;
```

## 19.2 The class PIC

The class `Pic` (Persistent Indexed Collection) implements an indexed collection as a binary tree. An identity is given to an instance of the class `PIC` so that when the object's life-time ends, its state will be saved to disk. The file name used to save the state is the object's identity.

The specification for the class `PIC` is as follows:



```

with Ada.Strings.Unbounded, Ada.Finalization;
use   Ada.Strings.Unbounded, Ada.Finalization;
generic
  type Index is private;           --Index for record
  type Data  is private;           --Data for record
  with function ">"( F:in Index; S:in Index ) return Boolean;
package Class_Pic is
  Not_There, Mainists, Per_Error : exception; --Raised Exceptions
  type Pic is new Limited_Controlled with private;
  procedure Initialize( The:in out Pic );
  procedure Initialize( The:in out Pic; Id:in String );
  procedure Finalize( The:in out Pic );
  procedure Discard( The:in out Pic );
  procedure Set_Name( The:in out Pic; Id:in String );
  function  Get_Name( The:in Pic ) return String;

  procedure Add( The:in out Pic; I:in Index; D:in Data );
  procedure Extract( The:in out Pic; I:in Index; D:in out Data );
  procedure Update( The:in out Pic; I:in Index; D:in out Data );

private
  type Leaf;                       --Index + Data
  type Subtree is access Leaf;     --
  type Pic is new Limited_Controlled with record
    Tree : Subtree := null;        -- Storage
    Obj_Id : Unbounded_String;     -- Name of object
  end record;

  function Find( The:in Subtree; I:in Index) return Subtree;
  procedure Release_Storage( The:in out Subtree );

end Class_Pic;

```

*Note:* The declaration of the type `leaf` is a forward declaration so that the type `Subtree`, a pointer to a leaf can be declared. The type `leaf` is fully declared in the package body.

The two generic parameters, `Index` and `Data` represent the type of the index used to access the stored data. As this generic class will need to compare indices for ">" to establish the position of an index in the binary tree, a definition for ">" must be provided by the user of the class. Remember, the index may be of a type for which the operation ">" is not defined between two instances of `Index`. This forces the user of the package to provide implicitly or explicitly an implementation for the comparison function ">".

The implementation of the class `Pic` uses the data structure `Element` to hold the index and the data associated with the index. The data structure `Leaf` represents a leaf of the binary tree which is composed of a left and right pointer plus the data structure `Element`.

```

with Unchecked_Deallocation, Sequential_Io;
package body Class_Pic is

  type Element is record           --
    S_Index: Index;                --The Index
    S_Data : Data;                 --The Data
  end record;

  type Leaf is record              --
    Left  : Subtree;               --Possible left node
    Rec   : Element;               --Index + data
    Right : Subtree;               --Possible right node;
  end record;

```

For example, after the following data is added to the data structure:

Country	IDC
Canada	+1
USA	+1
Belgium	+32
Germany	+49

the resultant tree would be as illustrated in Figure 19.1.

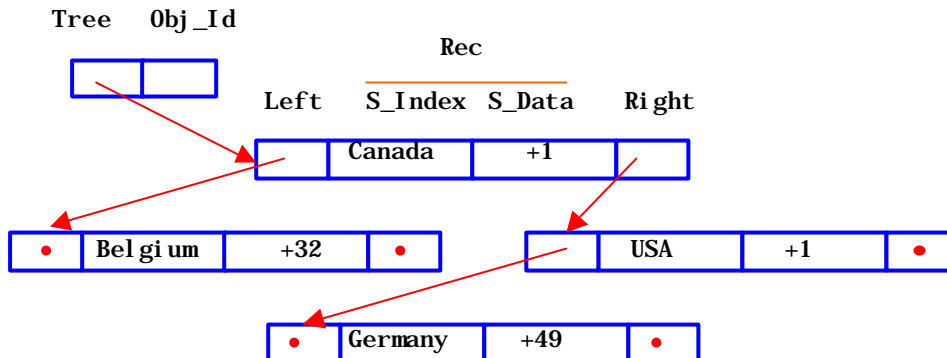


Figure 19.1 Binary tree holding four data items.

The rules for adding items to a binary tree are:

If the current pointer to a leaf is **null**:

- Insert the item at this point.

If the current pointer to a leaf is not **null**.

- If the index of the item to be inserted is less than the current index, then recursively call add on the left hand subtree.
- If the index of the item to be inserted is larger than the current index, then recursively call add on the right hand subtree.

For example, if the IDC of Norway were added, the resultant tree would be as illustrated in Figure 19.2.

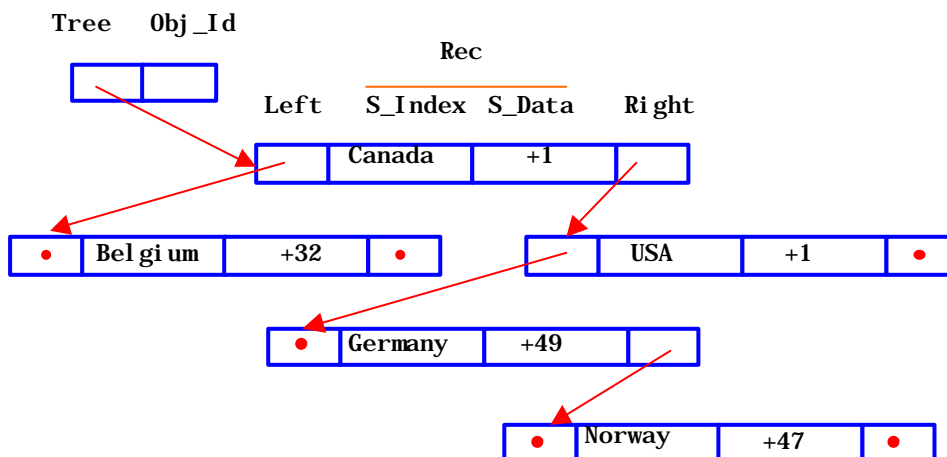


Figure 19.2 Binary tree after adding the country Norway.

The process to add the country Norway to the tree is:

Step	Current leaf contents	Action
1	Canada	Try inserting at RHS leaf.
2	USA	Try inserting at LHS leaf.
3	Germany	Try inserting at RHS leaf.
4	Empty	Insert new leaf (Norway).

The package `sequential_io` is used to hold the saved state of the binary tree. This is simply a file of records of type `Element`. An instantiation of this package `io` is created to allow input and output to take place on instances of `Element`.

```
package Io is new Sequential_Io( Element );
```

The procedure `initialize` sets the binary tree to a defined empty state.

```
procedure Initialize( The:in out Pic ) is
begin
  The.Tree := null;    --No storage
end Initialize;
```

*Note: This is not necessary as the elaboration of an instance of the class will set the tree to **null** as its initial value.*

The procedure `Initialize` is called to restore the state of the binary tree from a file. The second parameter to the procedure `Initialize` is the object's identity. The state of the object is held in a file with the same name as the object's name. This procedure reads in the stored index and data items and uses the procedure `Add` to rebuild the tree. The rebuilding of the tree re-creates an exact copy of the structure of the saved tree. This is due to the way the index and data items were stored. The process of saving the state of the binary tree is implemented in the procedure `Finalize`.

```
procedure Initialize( The:in out Pic; Id:in String ) is
  Per : Io.File_Type;    --File descriptor
  Cur : Element;         --Persistent data record element
begin
  Set_Name( The, Id );    --Name object
  Io.Open( Per, Io.In_File, Id ); --Open saved state
  while not Io.End_Of_File( Per ) loop --Restore saved state
    Io.Read( Per, Cur );
    Add( The, Cur.S_Index, Cur.S_Data );
  end loop;
  Io.Close( Per );
exception
  when others => raise Per_Error; --Return real exception
end Initialize;           -- as sub code
```

The procedure `Finalize` saves the state of an object which has an identity just before the object is destroyed. The data from the binary tree is saved in the order:

- Item held in the current node.
- The contents of the left-hand side.
- The contents of the right-hand side.

This unusual ordering saves the index and data item of the leftmost leaves nearest the root first. Thus, when the data is restored the structure of the tree will be the same. For example, if the data structure, illustrated in Figure 19.2 were saved, then the order of saving the data would be:

Canada, Belgium, USA, Germany, Norway.

When added to a binary tree, this would recreate the tree structure present in the original object.

```

procedure Finalize( The:in out Pic ) is
  Per : Io.File_Type;      --File descriptor
  procedure Rec_Finalize( The:in Subtree ) is --Save state
  begin
    if The /= null then                                --Subtree save as
      Io.Write( Per, The.Rec );                            -- Item
      Rec_Finalize( The.Left );                            -- LHS
      Rec_Finalize( The.Right );                          -- RHS
    end if;
  end Rec_Finalize;
begin
  if To_String(The.Obj_Id) /= "" then                    --If save state
    Io.Create( Per, Io.Out_File,
      To_String( The.Obj_Id ) );
    Rec_Finalize( The.Tree );
    Io.Close( Per );
  end if;
  Release_Storage( The.Tree );
exception                                --Return real exception
  when others => raise Per_Error;        -- as sub code
end Finalize;

```

The procedure Discard disassociates the object identity from the object and resets the state of the tree to empty. This procedure should be used when the object's state is not required to be saved to disk.

```

procedure Discard egin
  Set_Name( The, "" );                                --No name
  Release_Storage( The.Tree );                        --Release storage
end Discard;

procedure Set_Name( The:in out Pic; Id:in String ) is
begin
  The.Obj_Id := To_Unbounded_String(Id); --Set object name
end Set_Name;

function Get_Name( The:in Pic ) return String is
begin
  return To_String( The.Obj_Id );                    --Name of object
end Get_Name;

```

The procedure `Add` uses the classic recursive mechanism for adding data items to a binary tree. The process is to add the data item to an empty leaf of the tree. If this is not possible then the current leaf's data item is compared to the data item to be inserted. Depending on how the comparison collates the process is recursively called on either the left or the right-hand subtree.

```

procedure Add( The:in out Pic; I:in Index; D:in Data ) is
  procedure Add_S(The:in out Subtree; I:in Index; D:in Data) is
    begin
      if The = null then
        The := new Leaf'( null, Element'(I,D), null );
      else
        if I = The.Rec.S_Index then           --Index all ready exists
          raise Mainists;
        elsif I > The.Rec.S_Index then       --Try on RHS
          Add_S( The.Right, I, D );
        else                                --LHS
          Add_S( The.Left, I, D );
        end if;
      end if;
    end Add_S;
  begin
    Add_S( The.Tree, I, D );
  end Add;

```

The procedures `Extract` and `Update` respectively read a data value and update a data value using the supplied index. Both these procedures use the function `Find` to find the leaf in which the data item to be accessed is held.

```

procedure Extract(The:in out Pic; I:in Index; D:in out Data) is
  Node_Is : Subtree;
  begin
    Node_Is := Find( The.Tree, I );           --Find node with iey
    D := Node_Is.Rec.S_Data;                  --return data
  end Extract;

procedure Update(The:in out Pic; I:in Index; D:in out Data) is
  Node_Is : Subtree;
  begin
    Node_Is := Find( The.Tree, I );           --Find node with iey
    Node_Is.Rec.S_Data := D;                  --Update data
  end Update;

```

## 288 Persistence

The procedure `Find` uses a recursive descent of the tree to find the selected index. If the index is not found, then the exception `Not_There` is raised.

```
function Find( The:in Subtree; I:in Index) return Subtree is
begin
  if The = null then raise Not_There; end if;
  if I = The.Rec.S_Index then
    return The;                                --Found
  else
    if I > The.Rec.S_Index
      then return Find( The.Right, I );    --Try RHS
    else return Find( The.Left, I );      --Try LHS
    end if;
  end if;
end Find;
```

As the tree is built using dynamic storage, the storage must be released. The procedure `Release_Storage` carries out this task:

```
procedure Dispose is
  new Unchecked_Deallocation( Leaf, Subtree );

procedure Release_Storage( The:in out Subtree ) is
begin
  if The /= null then
    Release_Storage( The.Left ); --Free LHS
    Release_Storage( The.Right ); --Free RHS
    Dispose( The );             --Dispose of item
  end if;
  The := null;                  --Subtree root NULL
end Release_Storage;

end Class_Pic;
```

*Note: The implicit garbage collector could be used, but this would only be called when the instance of the class `Pic` went out of scope. If this object is serially re-used then storage used by the program could become excessive.*

## 20 Tasks

This chapter describes the Ada task mechanism that allows several threads of execution to take place in a program simultaneously. This facilitates the construction of real-time programs that can process messages generated from multiple sources in an orderly manner.

### 20.1 The task mechanism

A program may have sections of code that can be executed concurrently as they have no interaction or dependency. For example, the calculation of the factorial of an integer number and the determination of whether a number is prime, may be done concurrently as separate threads of execution. This can be implemented by means of a **task type** within a package. When elaborated, an instance of the task type will execute as a separate thread. Communication between the executing threads is performed using the **entry** construct which allows a rendezvous to be made between two concurrently executing threads. At the rendezvous, information may be interchanged between the tasks.

The specification for two packages is given below. The first package defines a task to calculate the factorial of a positive number and the second determines whether or not a positive number is a prime.

```
package Pack_Factorial is
  task type Task_Factorial is
    entry Start( F:in Positive );
    entry Finish( Result:out Positive );
  end Task_Factorial;
end Pack_Factorial;
```

--Specification  
--Rendezvous  
--Rendezvous

```
package Pack_Is_A_Prime is
  task type Task_Is_Prime is
    entry Start( P:in Positive );
    entry Finish( Result:out Boolean );
  end Task_Is_Prime;
end Pack_Is_A_Prime;
```

--Specification  
--Rendezvous  
--Rendezvous

*Note: The rendezvous Start is used to pass data to the task and the rendezvous Finish is used to pass the result back.*

A task is created using the normal Ada elaboration mechanism. To create an instance of the task `Task_Factorial` the following declaration is used:

```
Thread_1 : Task_Factorial;
```

## 290 Tasks

The task will start executing as an independent thread as soon as the block surrounded by the declaration is entered. A rendezvous with this executing task to pass it the number 5, is written as follows:

```
Thread_1.Start(5);           --Start factorial calculation
```

*Note:* This can be thought of as sending the message *Start* with a parameter of 5 to the task *Thread\_1*.

The tasks described above may be used as follows:

```
with Ada.Text_IO, Ada.Integer_Text_IO,
     Pack_Factorial, Pack_Is_A_Prime;
use  Ada.Text_IO, Ada.Integer_Text_IO,
     Pack_Factorial, Pack_Is_A_Prime;
procedure Main is
  Thread_1 : Task_Factorial;
  Thread_2 : Task_Factorial;
  Thread_3 : Task_Is_Prime;
  Factorial: Positive;
  Prime    : Boolean;

begin
  Thread_1.Start(5);           --Start factorial calculation
  Thread_2.Start(7);           --Start factorial calculation
  Thread_3.Start(97);          --Start is_prime calculation

  Put("Factorial 5 is ");
  Thread_1.Finish( Factorial ); --Obtain result
  Put( Factorial ); New_Line;

  Put("Factorial 7 is ");
  Thread_2.Finish( Factorial ); --Obtain result
  Put( Factorial ); New_Line;

  Put("97 is a prime is ");
  Thread_3.Finish( Prime );     --Obtain result
  if Prime then                --
    Put("True");               -- and print
  else
    Put("False");
  end if;
  New_Line;
end Main;
```

*Note:* The tasks start executing as soon as the **begin** of the block in which they are elaborated is entered. The rendezvous point *Start* is used to control this wayward behaviour.

This is in essence a client-server relationship between the main program, the client, which requests a service from the server tasks.

### 20.1.1 Putting it all together

When run, this would deliver the following results:

```
Factorial 5 is      120
Factorial 7 is     5040
97 is a prime is True
```



The execution of the above program can be visualized as Figure 20.1

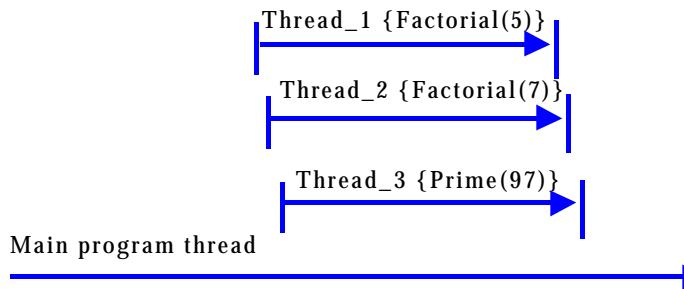


Figure 20.1 Illustration of active threads in the above program.

Once started, each of the threads will execute concurrently until the `Finish` rendezvous is encountered, which is used to deliver the result to the initiator of the tasks.

*Note: The actual implementation of the concurrency will depend on the underlying architecture, both software and hardware, of the platform on which the program is executed.*

### 20.1.2 Task rendezvous

The rendezvous mechanism is used for:

- synchronizing two separate threads so that information may be exchanged.
- synchronizing the execution of two threads.

A rendezvous is achieved by one task having an **entry** statement and the other task performing a call on this **entry**. For example, the code for a rendezvous to pass a `Positive` number to the task object `thread_1` the code would be:

Main program (client) which elaborates thread1	Body of task Thread_1 (server)
<code>Thread_1.Start(5);</code>	<pre> <b>accept</b> Start(F:<b>in</b> Positive) <b>do</b>     Factorial := F; <b>end</b> Start; </pre>

To achieve this effect, one of the threads of control will be suspended until the other thread catches up. Then at the rendezvous, data, in this case the number 5, is transferred between the tasks. The code between **do** and **end** is executed with the client task suspended. After the code between **do** and **end** has been executed both tasks resume their independent execution.

This rendezvous between the two tasks is illustrated in Figure 20.2 in which the main program task rendezvous with an instance of the task `Factorial`.

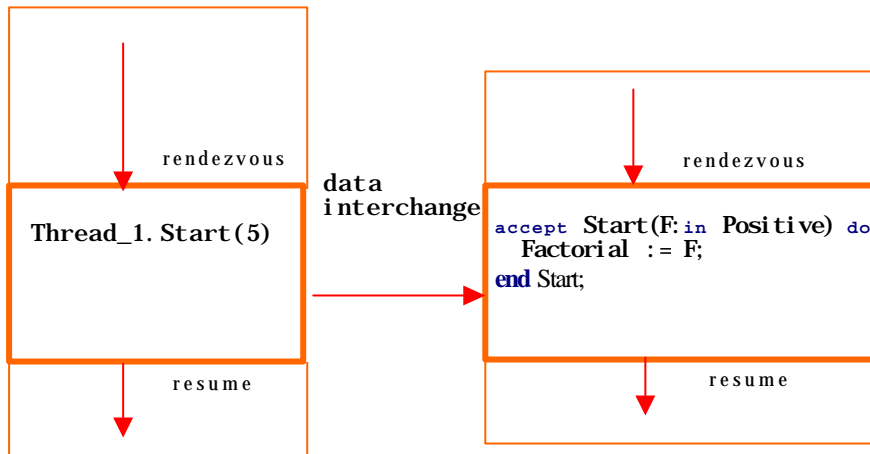


Figure 20.2 Illustration of a rendezvous.

Other variations on the rendezvous are:

Variation	Client	Server task
No information passed.	Thread_1.Start;	<b>accept</b> Start;
No information passed but Thread_1 executes statements during the rendezvous.	Thread_1.Start;	<b>accept</b> Start do Statements; <b>end</b> Start;

### 20.1.3 The task's implementation

In the body of the package Pack\_Factorial shown below, the task Task\_Factorial uses two rendezvous points:

- Start to obtain the data to work on.
- Finish to deliver the result.

When the task's thread of control reaches the end of the task body, the task terminates. Any attempted rendezvous with a terminated task will generate the exception Task\_Error.

```

package body Pack_Factorial is
  task body Task_Factorial is                                --Implementation
    Factorial : Positive;
    Answer    : Positive := 1;
  begin
    accept Start( F:in Positive ) do                          --Factorial
      Factorial := F;
    end Start;
    for I in 2 .. Factorial loop                               --Calculate
      Answer := Answer * I;
    end loop;
    accept Finish( Result:out Positive ) do                    --Return answer
      Result := Answer;
    end Finish;
  end Task_Factorial;
end Pack_Factorial;

```

Likewise, the task `Task_Is_Prime` in the package `Pack_Is_A_Prime` receives and delivers data to another thread of control.

```
package body Pack_Is_A_Prime is
  task body Task_Is_Prime is           --Implementation
    Prime : Positive;
    Answer: Boolean := True;
  begin
    accept Start( P:in Positive ) do   --Factorial
      Prime := P;
    end Start;
    for I in 2 .. Prime-1 loop         --Calculate
      if Prime rem I = 0 then
        Answer := False; exit;
      end if;
    end loop;
    accept Finish( Result:out Boolean ) do --Return answer
      Result := Answer;
    end Finish;
  end Task_Is_Prime;
end Pack_Is_A_Prime;
```

## 20.2 Parameters to a task type

In the previous example, the rendezvous `Start` is used to pass initial values to the task. This can be done explicitly, when the task is created by using a discriminated task type. However, the discriminant must be a discrete type or access type. For example, the specification of the task `Task_Factorial` can be defined as follows:

```
package Pack_Factorial is
  task type Task_Factorial(F:Positive) is --Specification
    entry Finish( Result:out Positive ); --Rendezvous
  end Task_Factorial;
end Pack_Factorial;
```

Then an instance of the task can be elaborated as follows:

```
Thread_1 : Task_Factorial(7);           --Task is
```

The body of the task type is now:

```
package body Pack_Factorial is
  task body Task_Factorial is           --Implementation
    Answer : Positive := 1;
  begin
    for I in 2 .. F loop                 --Calculate
      Answer := Answer * I;
    end loop;
    accept Finish( Result:out Positive ) do --Return answer
      Result := Answer;
    end Finish;
  end Task_Factorial;
end Pack_Factorial;
```

*Note: The discriminant to the task type is not specified in the body.*

### 20.2.1 Putting it all together

Using the new definition of the task type in the package `Pack_factorial` the following code can now be written:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Pack_Factorial;
use  Ada.Text_IO, Ada.Integer_Text_IO, Pack_Factorial;
procedure Main is
  Num      : Positive;
begin
  Num := 7;

  declare
    Factorial: Positive;           --Answer
    Thread_1 : Task_Factorial(Num); --Task is
  begin
    --Do some other work as well
    Put("Factorial "); Put(Num); Put(" is ");
    Thread_1.Finish( Factorial ); --Obtain result
    Put( Factorial ); New_Line;
  end;
end Main;
```

## 20.3 Mutual exclusion and critical sections

In many cases of real time working, sections of code must not be executed concurrently. The classic example is the adding or removing of data in a shared buffer. For example, to perform a copy operation between two separate devices a shared buffer can be used to even out the differences in response-time. This can be illustrated diagrammatically as shown in Figure 20.3.

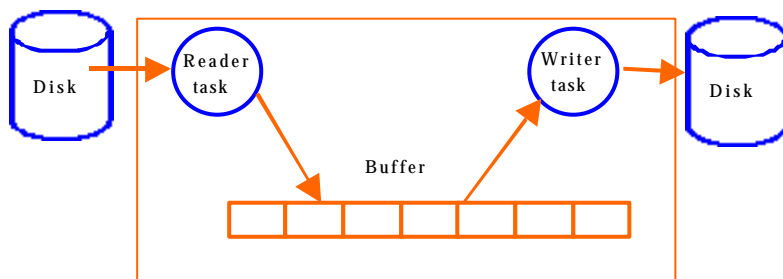


Figure 20.3 Illustration of copy with a buffer to even out the differences in read and write rates.

The problem is how to prevent both the read and write tasks accessing the buffer simultaneously, causing the consequential corruption of indices and data. The solution is to have the buffer as an instance of a protected type.

## 20.4 Protected type

In essence, an instance of a protected type is an object whose methods have strict concurrency access rules. A protected object, an instance of a protected type, is composed of data and the procedures and functions that access the data. The table below summarizes the concurrent access rules for procedures and functions in a protected object.

Unit	Commentary	Access
<b>procedure</b>	A procedure will only execute when no other units are being executed. If necessary the procedure will wait until the currently executing unit(s) have finished.	Read and write.
<b>function</b>	A function may execute simultaneously with other executing functions. However, a function cannot execute if a procedure is currently executing.	Read only.
<b>entry</b>	Like a procedure but may also have a barrier condition associated with the entry. If the barrier condition is false the entry is queued until the barrier becomes true.	Read and write

## 20.5 Implementation

The implementation of a program to perform an efficient copy using an in store buffer to even out differences in response rates can be implemented as two tasks and a protected object, as illustrated in Figure 20.4.

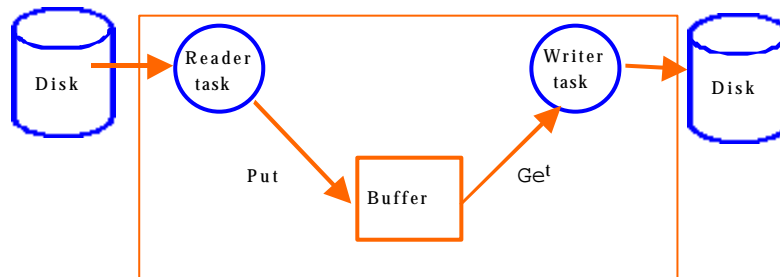


Figure 20.4 Copy implemented using two tasks and a protected object buffer.

The responsibilities of the components are as follows:

Name	Object is	Responsibilities
Task_Reader	Task	Read data from the file and then pass the data to the buffer. Note: The task will block if the buffer is full.
Task_Writer	Task	Take data from the buffer task and write the data to the file. Note: The task will block if there is no data in the buffer.
PT_Buffer	Protected type	Serialize the storing and retrieving of data to and from a buffer.

*Note: The blocking is achieved with a guard to the accept statement. This is described in the section on guarded accepts.*

A package Pack\_types is defined to allow commonly-used types to be conveniently kept together.

```
with Ada.Text_IO;
use  Ada.Text_IO;
package Pack_Types is
  type P_File_Type is access all Ada.Text_IO.File_Type;
  Eot  : constant Character := Character'Val(0);
  Cr   : constant Character := Character'Val(15);
  Queue_Size : constant := 3;

  type Queue_No is new Integer range 0 .. Queue_Size;
  type Queue_Index is mod Queue_Size;
  subtype Queue_Range is Queue_Index;
  type Queue_Array is array ( Queue_Range ) of Character;
end Pack_Types;
```

*Note:* The above package is used to define the type P\_File\_Type which is used by several other program units.

The specification for the buffer protected type is as follows:

```
with Pack_Types;
use  Pack_Types;
package Pack_Threads is
  protected type PT_Buffer is          --Task type specification
    entry Put( Ch:in Character; No_More:in Boolean );
    entry Get( Ch:in out Character; No_More:out Boolean);
  private
    Elements      : Queue_Array;          --Array of elements
    Head          : Queue_Index := 0;      --Index
    Tail          : Queue_Index := 0;      --Index
    No_In_Queue   : Queue_No := 0;         --Number in queue
    Fin           : Boolean := False;       --Finish;
  end PT_Buffer ;

  type P_PT_Buffer is access all PT_Buffer ;
```

The Ada specification for the reader and writer tasks are as follows:

```
task type Task_Read( P_Buffer:P_PT_Buffer ;
                    Fd_In:P_File_Type) is
  entry Finish;
end Task_Read;

task type Task_Write( P_Buffer:P_PT_Buffer ;
                    Fd_Out:P_File_Type) is
  entry Finish;
end Task_Write;
end Pack_Threads;
```

*Note:* To allow the reader and writer tasks to communicate with the buffer, a reference to the buffer protected object is passed to these tasks. A reference to the buffer protected object has to be passed as a protected object is of limited type.  
The same strategy is used to pass an instance of File\_Type.

The implementation of the above program is split into two procedures. The procedure `Do_Copy` does the actual work of copying between the two files.

```
with Ada.Text_Io, Pack_Threads, Pack_Types;
use Ada.Text_Io, Pack_Threads, Pack_Types;
procedure Do_Copy(From:in String; To:in String) is
  type State is ( Open_File, Create_File );
  Fd_In   : P_File_Type := new Ada.Text_Io.File_Type;
  Fd_Out  : P_File_Type := new Ada.Text_Io.File_Type;
  Mode    : State := Open_File;
begin
  Open( File=>Fd_In.all, Mode=>In_File, Name=>From);
  Mode := Create_File;
  Create(File=>Fd_Out.all, Mode=>Out_File, Name=>To);
  declare
    Buffers : P_PT_Buffer := new PT_Buffer ;
    Reader  : Task_Read( Buffers, Fd_In );
    Writer  : Task_Write( Buffers, Fd_Out );
  begin
    Reader.Finish; Close( Fd_In.all );  --Finish reader task
    Writer.Finish; Close( Fd_Out.all );  --Finish writer task
  end;
exception
  when Name_Error =>
    case Mode is
      when Open_File =>
        Put("Problem opening file " & From ); New_Line;
      when Create_File =>
        Put("Problem creating file " & To ); New_Line;
    end case;
  when Tasking_Error =>
    Put("Task error in main program"); New_Line;
end Do_Copy;
```

*Note:* Explicit de-referencing of instances of a `File_Type` is achieved using `.all`.

The procedure `copy` extracts the arguments for the copy operation.

```
with Ada.Text_Io, Ada.Command_Line, Do_Copy;
use Ada.Text_Io, Ada.Command_Line;
procedure Copy is
begin
  if Argument_Count = 2 then
    Do_Copy ( Argument(1), Argument(2) );
  else
    Put("Usage: copy from to"); New_Line;
  end if;
end Copy;
```

When a pointer to a **protected type** (for example, `PT_Buffer`) is elaborated, no object is created. The creation of an instance of the protected type `PT_Buffer` is performed using `new` as follows:

```
Buffers : P_PT_Buffer := new PT_Buffer ;
```

*Note:* As a protected object is limited, using an access value is one way of making the protected object visible to several program units.

The implementation of the reader task is then:

```
with Ada.Text_Io;
use   Ada.Text_Io;
package body Pack_Threads is

  task body Task_Read is                                --Task implementation
    Ch      : Character;
  begin
    while not End_Of_File( Fd_In.all ) loop
      while not End_Of_Line( Fd_In.all ) loop
        Get( Fd_In.all, Ch);                            --Get character
        P_Buffer.Put( Ch, False );                      --Add to buffer
      end loop;
      Skip_Line( Fd_In.all );                            --Next line
      P_Buffer.Put( Cr, False );                         --New line
    end loop;
    P_Buffer.Put( Eot, True );                          --End of characters

    accept Finish;
  exception
    when Tasking_Error =>
      Put( "Exception in Task read" ); New_Line;
  end Task_Read;
```

The rendezvous Finish is used by the reader to indicate that there is no more data.

*Note:* As tasking errors are not propagated beyond the task, a specific exception handler is used to detect this eventuality.

The character constant Cr is used to indicate the newline character.

Similarly, the writer task is implemented as follows:

```
task body Task_Write is                                --Task implementation
  Last      : Boolean := False;                        --No more data
  Ch        : Character;                               --Character read
begin
  loop
    P_Buffer.Get( Ch, Last );                          --From buffer
    exit when Last;                                    --No more characters
    if Ch = Cr then
      New_Line( Fd_Out.all );                          --New line
    else
      Put( Fd_Out.all, Ch );                          --Character
    end if;
  end loop;
  accept Finish;                                       --Finished
exception
  when Tasking_Error =>
    Put( "Exception in Task write" ); New_Line;
end Task_Write;
```

### 20.5.1 Barrier condition entry

The protected type uses one additional facility, that of a barrier entry. If the buffer becomes full, a mechanism is needed to prevent further data being added. The barrier:

```
entry Put( Ch:in Character; No_More:in Boolean )
  when No_In_Queue < Queue_Size is
```



to the **entry** prevents the **entry** being processed until there is room in the buffer. If the buffer is full then the reader task is suspended (blocked) until a successful get **entry** is made. The guards for an **entry** statement are re-evaluated after a successful call on the protected object. The full implementation of the protected type `PT_Buffer` is as follows:

```
protected body PT_Buffer is
```

The queue is implemented in sequential store with the two indices `head` and `tail` keeping track of the current extraction and insertion points respectively. A count of the active cells used in the buffer is held in `no_in_queue`. Figure 20.5 illustrates the queue after adding the characters 't', 'e', 'x', 't'.

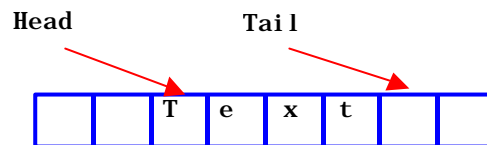


Figure 20.5 Queue holding the characters 'text'.

The procedure `Put` in the body of the protected object adds new data to the queue. Data can only be added to the queue when there is room. The index `Tail` marks the position of the next data item to be added. When no more data is available to add to the queue the variable `Fin` is set to true.

```
entry Put( Ch:in Character; No_More:in Boolean )
  when No_In_Queue < Queue_Size is
begin
  if No_More then
    Fin := True;
  else
    Elements( Tail ) := Ch;
    Tail := Tail+1;
    No_In_Queue := No_In_Queue + 1;
  end if;
end;
```

The procedure `Get` extracts data from the queue. The `head` indexes the data item at the front of the queue. The parameter `eof` is set to true when no more data is available. This is different from a temporary unavailability of data due to the reader task blocking.

```
entry Get(Ch:in out Character; No_More:out Boolean)
  when No_In_Queue > 0 or else Fin is
begin
  if No_In_Queue > 0 then
    Ch := Elements( Head );
    Head := Head+1;
    No_In_Queue := No_In_Queue - 1;
    No_More := False;
  else
    No_More := True;
  end if;
end;

end PT_Buffer ;

end Pack_Threads;
```

*Note:* When all the data has been exhausted from the buffer, the procedure `get` will return false in its second parameter.

## 300 Tasks

### 20.5.2 Putting it all together

When the above program is compiled and run it will perform a copy operation using the internal buffer to even out differences between the speed of the input and output streams. For example, to copy the contents of `from` to the file `to` a user can type:

```
copy from to
```

## 20.6 Delay

Execution of a program can be delayed for a specific number of seconds or until a specific time is reached. For example, to implement a delay of 2.5 seconds in a program the following **delay** statement is used.

```
delay 2.5;
```

*Note: The delay time is of type `Duration`, which has a range of 0.0 .. 86\_400.0 and is defined in the package `Ada.Calendar`. The specification of the package `Ada.Calendar` is contained in Section C.15, Appendix C.*

To delay until a specific time the **until** form of the **delay** statement is used. To delay part of a program until the 1st January 2000 the following statement is used:

```
delay until Time_Of(2010,1,1,0.0);    -- Until 1 Jan 2010
```

*Note: The package `Ada.Calendar` contains the definition for `Time_Of` which returns the date as an instance of `Time`.*

## 20.7 Choice of accepts

The **select** construct is used to select between several different possible rendezvous. The form of the select construct is as follows:

```
select                                -- Choice of accepts
  accept option1 do
    ...
  end;
or
  accept option2 do
    ...
  end;
end select;
```

This can be used when a task can have more than one rendezvous made with it from several different sources. For example, a task controlling output to a terminal may be accessed by either a text interface for information messages, or a block image interface for pictorial data. The **select** construct causes a wait until one of the specified rendezvous is made.

*Note:* A protected type may be simulated by using a task which consists of a loop in which a select statement is embedded. Each rendezvous within the **select** statement will then have its execution serialized. For example:

```
loop
  select
    accept option1 do end;
  or
    accept option1 do end;
  end select;
end loop;
```

### 20.7.1 Accept alternative

An **else** part may be added to a **select** statement. The statements after the **else** will be obeyed if a rendezvous cannot be immediately made with any of the **accept** statements in the **select** construct.

```
select
  accept option1 do
    ...
  end;
else
  Statements;
end select
```

### 20.7.2 Accept time-out

The **select** construct may also include a time-out delay after which, if there is no **accept** called following the statements, the **delay** will be executed. The format of this variation of the select construct is:

```
select
  accept option1 do
    ...
  end;
or
  delay TIME;
  Statements;
end select;
```

*Note:* There may be only one **delay** alternative and no **else** part.

This construct can be used to implement a watchdog task that will report an error if it has not been polled for a certain time. This watchdog task can act as a safety measure to report that the software is not performing as expected. An implementation of a simple watchdog timer is as follows:

```
package Pack_Watchdog is
  task type Task_Watchdog is
    entry Poll;
    entry Finish;
  end Task_Watchdog;
end Pack_Watchdog;
```

## 302 Tasks

The entry `Poll` is called at regular intervals to prevent the watchdog task from reporting an error. The task is terminated by a call to `Finish`. The implementation of the task is as follows:

```
with Ada.Text_Io;
use   Ada.Text_Io;
package body Pack_Watchdog is
  task body Task_Watchdog is
    begin
      loop
        select
          accept Poll;
        or
          accept Finish;
          exit;
        or
          delay 0.2;
          Put( "WARNING Watchdog failure");
          New_Line;
          exit;
        end select;
        delay 0.0001;
      end loop;
    end Task_Watchdog;
end Pack_Watchdog;
```

*--Implementation*  
*--Successful poll*  
*--Terminate*  
*--Time out*  
*--Cause task switch*

If a poll is not received every 0.1 seconds then the task will report a warning to the user.

## 20.8 Alternatives to a task type

Tasks do not have to be defined as a task type. They can be defined as a package or even as part of a program unit.

### 20.8.1 As part of a package

A task can be specified as a package. In this format, there is less flexibility as now there can only be one instance of the task. For example, the task to calculate a factorial could have been specified as follows:

```
package Pack_Factorial is
  task Task_Factorial is
    entry Start( F:in Positive);
    entry Finish( Result:out Positive);
  end Task_Factorial;
end Pack_Factorial;
```

*--Specification*  
*--Rendezvous*  
*--Rendezvous*

the implementation of which is:

```
package body Pack_Factorial is
  task body Task_Factorial is           --Implementation
    Factorial : Positive;
    Answer    : Positive := 1;
  begin
    Put("Pack_factorial"); New_Line;
    accept Start( F:in Positive) do     --Factorial
      Factorial := F;
    end Start;
    for I in 2 .. Factorial loop        --Calculate
      Answer := Answer * I;
    end loop;
    accept Finish( Result:out Positive ) do --Return answer
      Result := Answer;
    end Finish;
  end Task_Factorial;
end Pack_Factorial;
```

The code to interact with this task in a package would be as follows:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Pack_Factorial;
use   Ada.Text_IO, Ada.Integer_Text_IO, Pack_Factorial;
procedure Main is
  Factorial: Positive;
begin
  Task_Factorial.Start(5);              --Start factorial calculation
                                       --Task running

  Put("Factorial 5 is ");
  Task_Factorial.Finish( Factorial );  --Obtain result
  Put( Factorial ); New_Line;
end Main;
```

*Note: If this form is used, then the task will come into immediate existence as soon as the program is executed.*

*Only one instance of the factorial task can be created.*

## 20.8.2 As part of a program unit

```

package Pack_Factorial is
  task type Task_Factorial(F:Positive) is      --Specification
    entry Finish( Result:out Positive );      --Rendezvous
  end Task_Factorial;
end Pack_Factorial;

--[pack_factorial.adb] Implementation
package body Pack_Factorial is
  task body Task_Factorial is                  --Implementation
    Answer    : Positive := 1;
  begin
    for I in 2 .. F loop                        --Calculate
      Answer := Answer * I;
    end loop;
    accept Finish( Result:out Positive ) do --Return answer
      Result := Answer;
    end Finish;
  end Task_Factorial;
end Pack_Factorial;

with Ada.Text_Io, Ada.Integer_Text_Io, Pack_Factorial;
use  Ada.Text_Io, Ada.Integer_Text_Io, Pack_Factorial;
procedure Main is
  Num      : Positive;
begin
  Num := 7;

  declare
    Factorial: Positive;                --Answer
    Thread_1 : Task_Factorial(Num);     --Task is
  begin
    --Do some other work as well
    Put("Factorial "); Put(Num); Put(" is ");
    Thread_1.Finish( Factorial );      --Obtain result
    Put( Factorial ); New_Line;
  end;
end Main;

```

*Note:*     The task will come into existence as soon as the program unit is executed.

## 20.9 Self-assessment

- What is a thread or task in a programming language?
- What Ada construct can be used to implement a thread or task?
- How is information passed between two threads? Explain why a special construct is required for this activity.
- What is the difference between a task type and a normal type?
- What is the difference between execution of a procedure and execution of a function in a protected type?
- What happens when a task type is elaborated?

- How can a task select the current rendezvous that is made with it, from a number of possible rendezvous?
- Why might a program need to employ a watchdog timer?
- How might a constantly running Ada program execute some code at a particular time of the day?

## 20.10 Exercises

Construct the following:

- *Fibonacci task*  
A thread or task which will calculate the n'th term of the Fibonacci series. The rendezvous with this task are:
  - `Calculate( n );` -- What term to find;
  - `Deliver( res );` -- The result.
- *Factorial*  
A thread or task which will calculate the factorial of a supplied value. The task should allow multiple serial calculations to be requested. The rendezvous with this task are:
  - `Calculate( n );` -- What term to find;
  - `Deliver( res );` -- The result;
  - `Finish;` -- Terminate the task.
- *Fast copy*  
A program to perform an optimal block copy using an intermediate buffer of disk blocks to even out any differences in speed between the input and output streams.
- *Communication link*  
A program to allow the sending of data between two computer systems using a serial port. The program should be able to inform the user if the other machine has not responded within the last two minutes.

## 21 System programming

This chapter shows how access can be made to the low-level facilities of the Ada language. This facilitates the construction of programs which interact with the system host system directly.

### 21.1 Representation clause

An enumeration may be given a specific value by a representation clause. For example, the following enumerations, defined for the type Country:

```
type Country is (USA, France, UK, Australia);
```

may each be given their international telephone dialling code with the following representation clause:

```
type Country is (USA, France, UK, Australia);  
for Country use (USA=> 1, France=> 33, UK=> 44, Australia=> 61);
```

Thus, internally the enumeration `France` would be represented by the number 33.

*Note: The values given to each enumeration must be in ascending order and unique.*

However, even though the enumerations may have non-consecutive representations, attributes of the enumeration will be as if there had been no representation clause. For example:

Expression	Delivers
Country'Succ( USA )	France
Country'Pred( Australia )	UK
Country'Pos( France )	1
Country'Val( 2 )	UK

To access the value of the enumeration requires the use of the generic package `Unchecked_Conversion` that will deliver an object as a different type without any intermediate conversions. The only restriction with the use of the generic function `Unchecked_Conversion` is that the source and destination objects must be of the same size.

In this case this can be ensured by informing the compiler of the size in bits required to use for the representation of an object of type `Country`. For example, to set the size for the enumeration `Country` to be the same size as the type `Integer` the following representation clause would be used:

```
type Country is (USA, France, UK, Australia);  
for Colour'Size use Integer'Size;  
for Country use (USA=> 1, France=> 33, UK=> 44, Australia=> 61);
```

*Note: The attribute 'Size delivers the size in bits of an instance of the type.*



### 21.1.1 Putting it all together

To print the international telephone code for France the following code can be used:

```
with System, System.Storage_Elements;
use System, System.Storage_Elements;
procedure Main is
  type Country is (USA, France, UK, Australia);
  for Colour'Size use Integer'Size;
  for Country use (USA=>1, France=>33, UK=>44, Australia=>61);

  function Idc is new Unchecked_Conversion( Country, Integer );
begin
  Put( "International dialling code for France is " );
  Put( Idc(France) );
  New_Line;
end Main;
```

which when run, would produce the following results:

```
International dialling code for France is 33
```

It would also be convenient to also include Canada in the Country enumeration for telephone codes. However, as Canada has the same country code as the USA, this cannot be done directly. The reason for this is that two enumerations may not have the same physical representation. The way round this is to define a renaming for Canada as follows:

```
function Canada return Country renames USA;
```

which defines Canada as a function that returns the enumeration USA as its result.

### 21.2 Binding an object to a specific address

In some limited situations it is necessary to read or write from absolute locations in memory. In the historic operating system MS DOS the time of day is stored in locations (in hexadecimal) 46E and 46C. The exact specification of what is stored is as follows:

Location (hexadecimal)	Contents
046E - 046F	The time of day in hours.
046C - 046D	The ticks past the current hour. Each tick is 5/91 seconds.

An object may be bound to an absolute location with the `for use` clause. For example, to bind the integer variable `Time_High` to the absolute location `16#46E#` the following declaration can be used:

```
Time_High_Address : constant Address := To_Address( 16#046C# );

type Time          is range 0 .. 65365;          --Unsigned
for Time'Size use 16;                             -- in 2 bytes

Time_High: Time;
  for Time_High'Address use Time_Low_Address;
```

*Note: Time is a type describing a 16 bit unsigned integer.  
The address 16#046E# must be of type Address that is defined in the package System. The child package System.Storage\_elements contains the function to\_address which converts an integer into an address.*

## 308 System programming

A program to print the current time of day in hours, minutes and seconds in a programming running under the DOS operating system is as follows:

```
with System, System.Storage_Elements,
     Ada.Text_IO, Ada.Integer_Text_IO;
use System, System.Storage_Elements,
     Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Time_High_Address : constant Address := To_Address( 16#046C# );
  Time_Low_Address  : constant Address := To_Address( 16#046E# );
  type Seconds_T is range 0 .. 1_000_000_000; --up to 65k * 5
  type Time is range 0 .. 65365; --Unsigned
  for Time'Size use 16; -- in 2 bytes
  Time_Low : Time;
  for Time_Low'Address use Time_High_Address;
  Time_High : Time;
  for Time_High'Address use Time_Low_Address;
  Seconds : Seconds_T;
begin
  Put("Time is ");
  Put( Time'Image(Time_High) ); Put(" :"); --Hour
  Seconds := (Seconds_T(Time_Low) * 5) / 91;
  Put(Seconds_T'Image(Seconds/60)); Put(" :"); --Mins
  Put(Seconds_T'Image(Seconds rem 60)); --Seconds
  New_Line;
end Main;
```

which when run on a DOS system would produce output of the form:

```
Time is 17 : 54 : 57
```

*Note: For this to work, the generated code must be able to access these low locations in DOS.*

### 21.2.1 Access to individual bits

On an MS DOS system memory address 16#0417# contains the status of various keyboard settings. Individual bits in this byte indicate the settings (set or not set) for the scroll lock, number lock, caps and insert keys. The layout of this byte is illustrated in Figure 21.1.

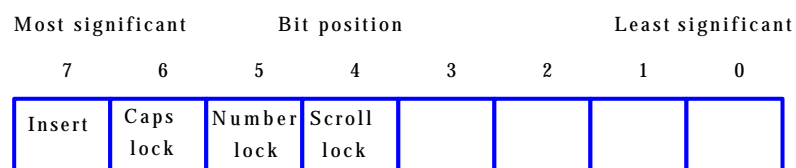


Figure 21.1 Keyboard status on an MSDOS system.

The following demonstration program prints out the status of the insert, caps lock, and number lock keys:

```
with System, System.Storage_Elements,
    Ada.Text_IO, Ada.Integer_Text_IO;
use System, System.Storage_Elements,
    Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Keyboard_Address : constant Address := To_Address( 16#417# );
  type Status      is ( Not_Active, Active );
  for Status        use ( Not_Active => 0, Active => 1 );
  for Status'Size   use 1;
```

The above declarations define the enumeration Status to occupy a single bit. The next set of declarations define Keyboard\_Status and access to the individual bits that make up the status byte. This is defined using a record structure with a representation clause for the specific layout of the bits.

```
type Keyboard_Status is
record
  Scroll_Lock : Status;           --Scroll lock status
  Num_Lock    : Status;           --Num lock status
  Caps_Lock   : Status;           --Caps lock status
  Insert      : Status;           --Insert status
end record;

for Keyboard_Status use
  record
    Scroll_Lock at 0 range 4..4; --Storage unit 0 Bit 4
    Num_Lock    at 0 range 5..5; --Storage unit 0 Bit 5
    Caps_Lock   at 0 range 6..6; --Storage unit 0 Bit 6
    Insert      at 0 range 7..7; --Storage unit 0 Bit 7
  end record;
Keyboardstatus_Byte : Keyboard_Status;
for Keyboardstatus_Byte'Address use Keyboard_Address;
```

The representation clause `Scroll_Lock at 0 range 4..4` requests that the object Scroll\_Lock be stored at an offset of 0 storage locations from the start of the record at bit position 4.

*Note: On a PC the storage unit size is one byte.*

*The bits selected may be outside the storage unit.*

The body of the program which interrogates these individual bits using the individual record components of Keyboard\_Status\_Byte is:

```
begin
  if Keyboardstatus_Byte.Insert = Active then
    Put("Insert mode set"); New_Line;
  else
    Put("Insert mode not set"); New_Line;
  end if;
  if Keyboardstatus_Byte.Caps_Lock = Active then
    Put("Caps lock set"); New_Line;
  else
    Put("Caps lock not set"); New_Line;
  end if;
  if Keyboardstatus_Byte.Num_Lock = Active then
    Put("Number lock set"); New_Line;
  else
    Put("Number lock not set"); New_Line;
  end if;
end Main;
```

## 310 System programming

which when run on an MSDOS system with none of these keys set would print:

```
Insert mode not set
Caps lock not set
Number lock not set
```

*Note: For this to work, the generated code must be able to access these low locations in DOS.*

### 21.3 Self-assessment

- Using a representation clause, the following enumeration for Country defines the IDC(International Dialling Code) for a small selection of countries.

```
type Country is (USA, France, UK, Australia);
for Country'Size use Integer'Size;
for Country use (USA=> 1, France=> 33,
                 UK=> 44, Australia=> 61);
```

What do the following deliver?

- (a) Country'Pos( USA )
- (b) Country'Val( 2 ).

- How can the IDC of France be extracted from the enumeration for France?
- As Canada has the same IDC as the USA, how can an enumeration for Canada be included in the list of countries above?
- How can the variable Cost be declared so that its address maps on to the byte at absolute location 040 in programs address space?

### 21.4 Exercises

Construct the following:

- *Memory dump*  
A program which prints in hexadecimal the contents of the bottom 100 locations of the current program.

## 22 A text user interface

This chapter defines an API (Application Program Interface) for use by an application program that reads and writes textual information to and from windows on a VDU screen. The TUI (Text User interface) uses the metaphor of non-overlapping windows. The application program is written using an event-driven regime. In this way, call-back functions are written to implement services requested by a user of the application.

The next chapter describes in detail the implementation of the TUI.

### 22.1 Specification

A TUI (Text User Interface) provides various types of non-overlapping windows on a VDU screen. The windows provided are:

- A text window into which characters from the Ada type `Character` can be written.
- A dialog window that provides a mechanism for a user to enter character data. Associated with a dialog window is a function that is called on completion of the user's entered data.
- A menu window from which a user can select a menu item from a list of available options.

For example, a demonstration application that converts miles into kilometres uses the TUI to input and display data as follows:

+-----+   Miles to kilometres   +-----+	
#-----+   Dialog  Miles	+-----+   Distance in miles = 50.00
-----+   50.0*	-----+   Distance in Kms = 80.47
+-----+	+-----+

*Note: A # in the top left-hand corner of a window signifies which window has the focus for input.*

The interface for this program consists of three distinct windows:

- A text window which displays the title of the program  
"Miles to kilometres"
- A dialog window that solicits input from the user. In this distance in miles to be converted into kilometres.
- A text window to display of the results of the conversion.

The TUI interface for writing to windows is modelled on the procedure `Put` in the package `Ada.Text_IO`. Associated with an instance of a Dialog window is a call-back function that implements the functionality of the user interaction. This is often referred to as an event-driven metaphor.

The dialog window's call-back function is executed when a user has finished inputting data into the program. For example, in the miles to kilometres program the dialog window's call-back function is executed when a user presses return completing the entry of the miles to be converted into kilometres. The call-back function calculates the conversion to kilometres and displays the answer in the result's window.

## 22.2 API for TUI

The API (Application Program Interface) for the TUI consists of a set of function and procedure calls. These are implemented as methods in the classes used to form the complete TUI interface. These window classes form an inheritance hierarchy illustrated in Figure 22.1.

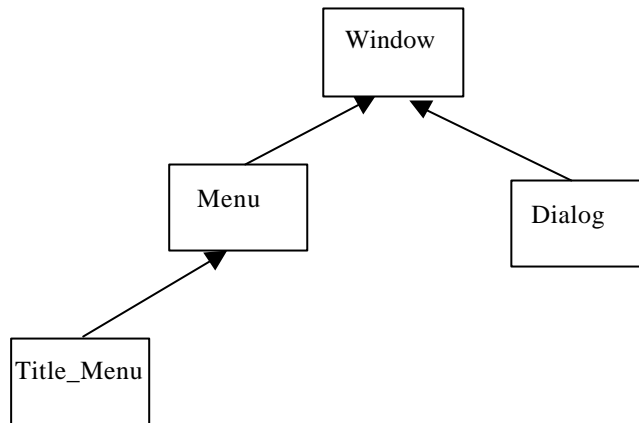


Figure 22.1 Window hierarchy.

The API calls for a Window are inherited to form the base API calls for a Menu and Dialog. Likewise, the base API calls for a Title\_Menu are inherited from a Menu.

### 22.2.1 To set up and close down the TUI

The following procedures are responsible for setting up and closing down the TUI system. These API calls are independent of any window and thus do not require as a parameter an actual instance of a window. These methods are class methods of the class Input\_manager.

function / procedure	Note
Window_Prolog;	Set up the environment for the TUI. This must be called outside the block in which windows are elaborated.
Window_Start;	After initializing any program-generated windows, start the application by allowing a user to interact with the program.
Window_Epilog;	Close down the window system. This must be called outside the block in which the windows are elaborated.

For example, the structure of a program using the TUI is:

```

procedure Main is
begin
  Window_Prologue;           -- Set-up window system
  declare
                                -- Declaration of windows used in
                                -- the program

  begin
                                -- Initialization of windows
                                -- used in program
                                -- Start the user interaction
    Window_Start;
  end;
  Window_Epilog;             -- Close window system
end Main;
  
```

*Note: The reason for this structure is to allow initialization code for any declared windows to be run after the window system has been initiated by the procedure Window\_Prolog and to allow any finalization code for the elaborated windows to be executed before the procedure Window\_Epilog is called. To avoid simultaneous access to a window, program initialization of a window must occur before the user is allowed to interact with the system.*

### 22.2.2 Window API calls

A text window is created with a declaration of the form:

```
Win : Window;
```

A text window can be created and written to using the following API calls:

Notes	Function / procedure
1	<b>procedure</b> Framework( The: <b>in out</b> Window; ABS_X_CRD, ABS_Y_CRD: POSITIVE; Max_X_Crd, Max_Y_Crd: Positive; Cb: <b>in</b> P_Cbf := null );
2	<b>procedure</b> Put( The: <b>in out</b> Window; Mes: <b>in</b> String );
2	<b>procedure</b> Put( The: <b>in out</b> Window; Ch: <b>in</b> Character );
2	<b>procedure</b> Put( The: <b>in out</b> Window; N: <b>in</b> Integer );
3	<b>Procedure</b> Position( The: <b>in out</b> Window; X, Y: <b>in</b> Positive );
4	<b>procedure</b> Clear( The: <b>in out</b> Window );
5	<b>procedure</b> New_Line( The: <b>in out</b> Window );
6	<b>procedure</b> Make_Window( The: <b>in out</b> Window; Mo: <b>in</b> Mode );

Notes:

- 1 Sets the absolute position and size of the window on the screen.  
The top left hand corner position is at: (abs\_x\_crd, abs\_y\_crd)  
The bottom right hand corner position is at:  
(abs\_x\_crd+max\_x\_crd-1, abs\_y\_crd+max\_y\_crd-1)
- 2 Displays information in a window. These functions are modelled after the procedures in Ada.Text\_Io.
- 3 Sets the current output position in the window.
- 4 Clears the window to all spaces.
- 5 Writes a newline to the window. This will cause the information in the window to scroll up if the current position is at the last line of the window.
- 6 Makes the displayed window visible or invisible.

### 22.2.3 Dialog API calls

A dialog window is created with a declaration of the form:

```
Diag : Dialog;
```

## 314 A Text user interface

A dialog window is inherited from a Window and as well as all the API calls of a Window has the following additional API call:

Note	Function / procedure
1	<code>procedure Framework ( The:in out Dialog; Abs_X, Abs_Y:in Positive; Max_X: in Positive; Name:in String; Cb:in P_Cbf );</code>

Note:

- 1 Sets the absolute position of the window on the screen. The size of the window is set with `max_x`. The call-back function `cb` will be called after the user has constructed a message in the dialog box. This is initiated by the user entering the Enter character (return key). When the Enter character is received the Dialog window calls the call-back function with a string parameter containing the user's entered text. The signature of the call-back function is:

`function Cb(Mes:in String) return String`

*where mes is the message typed by the user.*

### 22.2.4 User interaction with the TUI

A user of an application program that is built using the TUI API has the following switch characters defined:

Switch character	Description
TAB	Swaps the focus for user input to another window on the VDU screen. The active window is indicated by a # in the top left hand corner.
ESC	Activates the menu system. The menu system is described in detail in Section 22.4.
^E	Terminates the TUI session. All windows will be closed and the user returned to the environment which initiated the program.

A switch character is used to activate a specific window on the system or cause a global effect.

### 22.2.5 Classes used

The TUI API is contained in the following classes:

API for	Contained in the package	Notes
A window	Class_Window	-
A dialog box	Class_Dialog	Plus the API inherited from a Window.
A Menu bar	Class_Menu	Plus the API inherited from a Window.
A Menu Title	Class_Menu_Title	Plus the API inherited from a Menu
The TUI set up	Class_Input_Manager	Controls the input sent to the TUI.

## 22.3 An example program using the TUI

A short program to illustrate the use of many of the API calls is shown below. This example program converts a distance in miles entered by the user into kilometres. The package Pack\_Program contains the procedure Main that implements this program.



```

with Class_Window;
use Class_Window;
package Pack_Program is
  procedure Main;
private
  P_Result : P_Window;
end Pack_Program;

```

The call-back function `User_Input` is executed when a user has entered the distance in miles and then pressed Enter. This entered distance is converted to a floating point number using the procedure `get` in `Ada.Float_Text_IO` to convert a string into an instance of a `Float`. If the number is not valid or an error in the calculation occurs, then an appropriate message is displayed to the user.

```

with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;
with Class_Input_Manager, Class_Window, Class_Dialog;
use Class_Input_Manager, Class_Window, Class_Dialog;
package body Pack_Program is
  function User_Input( Cb_Mes:in String ) return String is
    Miles : Float;           --Miles input by user
    Last   : Positive;       --
    Str_Kms: String( 1 .. 10 ); --As a string in Kms
    Str_Mls: String( 1 .. 10 ); --As a string in Miles
  begin

```

```

    begin
      Get( Cb_Mes & ".", Miles, Last );
      Put( Str_Kms, Miles * 1.609_344, Aft=>2, Exp=>0 );
      Put( Str_Mls, Miles, Aft=>2, Exp=>0 );
      Put( P_Result.all, "Distance in Miles = " );
      Put( P_Result.all, Str_Mls ); New_Line( P_Result.all );
      Put( P_Result.all, "Distance in Kms = " );
      Put( P_Result.all, Str_Kms ); New_Line( P_Result.all );
    exception
      when Data_Error =>
        Put( P_Result.all, " Not a valid number" );
        New_Line( P_Result.all );
      when others =>
        Put( P_Result.all, " [Calculation error]" );
        New_Line( P_Result.all );
    end;
    return " ";
  end User_Input;

```

*Note:* The call-back function returns a string as its result. This provides a mechanism for returning information to the environment which called it. In this particular case, no information is returned. A decimal point ( .0 ) is appended to the user's input to allow a user to enter an integer value and still have the number processed correctly. The package `Ada.Text_IO` is required for the exception `Data_Error`.

In the procedure `Main` the three windows that will be displayed are declared. Then the call-back function `User_Input` is associated with dialog window and the program executes and waits for a user interaction.

```

procedure Main is
begin
  Window_Prologue;           --Setup window system
  declare
    Result : aliased Window;   --Result window
    Input  : Dialog;           --Input Window
    Title  : Window;           --title Window
  begin
    Framework( Title, 20, 1, 36, 5 ); --Title Window
    Framework( Result, 30, 10, 36, 5 ); --Result Window

    Position( Title, 8, 2 );
    Put( Title, "Miles to kilometres" );
    Framework( Input, 5, 10, 22,           --Input Window
               "Miles", User_Input'access );
    P_Result := Result'Unchecked_Access;

    Window_Start;             --Start the user interaction
  end;
  Window_Epilogue;           --Close window system
end Main;

end Pack_Program;

```

*Note:* The call to `window_prolog` initializes the TUI system.

### 22.3.1 How it all fits together

In the procedure `Main` the API framework is called to set the size and position of the various windows on the screen. This initialization is done before a user of the application is allowed to interact with the system.

```

begin
  Framework( Title, 20, 1, 36, 5 ); --Title Window
  Framework( Result, 30, 10, 36, 5 ); --Result Window

  Position( Title, 8, 2 );
  Put( Title, "Miles to kilometres" );
  Framework( Input, 5, 10, 22,           --Input Window
             "Miles", User_Input'access );

```

*Note:* The access value of the function `User_Input` is passed to the function `Framework` that sets up the call-back function.

The title window top left hand corner is at position (20,1) and the bottom right-hand corner at (20+36-1,1+5-1).

The access value of the Results window is assigned to `P_Result` so that it can be accessed by the call-back function `User_Input`. Remember, the call-back function must be at the library level.

```

P_Result := Result'Unchecked_Access;

```

*Note:* As the type used to declare the access value for a window is at the library level, `'Unchecked_Access` is required to be used. This is used to override the error that there is a potential inconsistency in using a library level variable (`P_Result`) to hold the access value of a local variable (`Result`).

The event-driven component part of the program is activated by a call to the procedure `Window_Start`. From this point onwards the program flow is driven by the user interacting with windows displayed on the terminal screen. Eventually the user will terminate the program, at which point the procedure `Window_Start` will return.

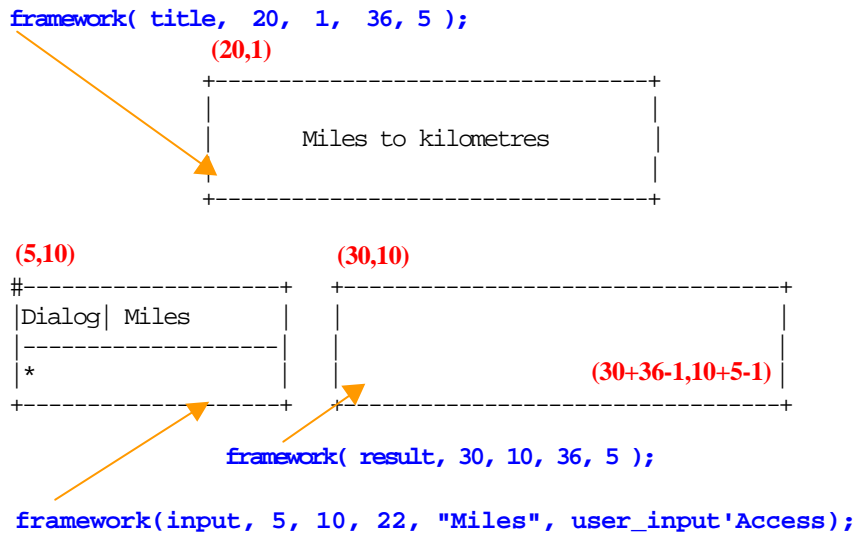
```
Window_Start;           --Start the user interaction
```

The procedure Window\_Epilog closes down the system. This must be called outside the block in which the instances of the windows were elaborated. This is to allow the finalization code to be called on the elaborated windows before Window\_Epilog is called.

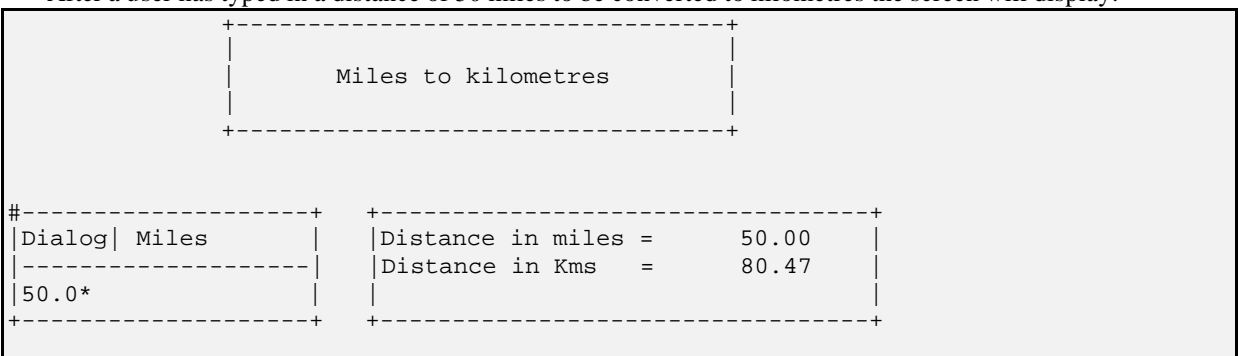
```
end;
Window_Epilogue;       --Close window system
end Main;
```

### 22.3.2 Putting it all together

When compiled with the TUI API library code, the screen display with added annotations to show which call of framework was used, together with position information is shown below:



After a user has typed in a distance of 50 miles to be converted to kilometres the screen will display:



### 22.4 The menu system

The menu system is based on the metaphor of a menu title bar at the top of the screen which changes as new menu options are selected. When a user selects away from the menu bar, the menu bar is returned to the top level of the menu hierarchy. The menu system is activated by typing the switch character ESC.

## 318 A Text user interface

For example, a menu title bar of two items:

Menu Component	Effect
About	Prints information about the program
Reset	Resets the program to an initial state

would be displayed as:

```
+-----+
|*About  | Reset  |
+-----+
```

The character \* indicates the current menu item. To select this menu item the user presses the selection key Enter. To change the current menu item the Arrow keys are used to move between the various options. Left arrow moves left and right arrow moves right the selected component. The effect of going right at the right-most menu item or left at the left-most menu item is to wrap around to the next item.

In addition to the API of a Window, the Menu and Menu\_Title API have the additional method of:

Note	Function / procedure
1	<pre>procedure Framework( The:in out Menu'Class;   M1:in String:=" "; W1:in P_Menu:=null; Cb1:in P_Cbf:=null;   M2:in String:=" "; W2:in P_Menu:=null; Cb2:in P_Cbf:=null;   M3:in String:=" "; W3:in P_Menu:=null; Cb3:in P_Cbf:=null;   M4:in String:=" "; W4:in P_Menu:=null; Cb4:in P_Cbf:=null;   M5:in String:=" "; W5:in P_Menu:=null; Cb5:in P_Cbf:=null;   M6:in String:=" "; W6:in P_Menu:=null; Cb6:in P_Cbf:=null );</pre>

Note:

1. This sets up a menu title bar or a menu title. The first parameter can be an instance of either a Menu or a Menu\_Title. Each menu item in the menu bar has three parameters:

- The displayed name of the menu item.
- A possible pointer to another menu bar.
- A possible pointer to a call-back function which is to be called when the menu is selected.

The second and third parameter are mutually exclusive. Thus, you can have either another menu bar or a call-back function.

As the menu bar is always at the top of the screen its position is not selected. It would of course be an error to have a window overlapping the menu bar.

The type P\_Cbf is defined as:

```
function Cb(Mes:in String) return String
String;
```

The following frameworks are used to set up a menu system with a main menu title bar and a selectable secondary menu bar tied to the main menu item Print.

```
with Class_Input_Manager, Class_Menu, Class_Menu_Title,
     Laser, Ink_Jet, About;
use   Class_Input_Manager, Class_Menu, Class_Menu_Title;
procedure Main is
begin
  Window_Prologue;
  declare
    Menu_Bar      : Menu_Title;
    Printer_Type  : aliased Menu;
  begin
    Framework( Printer_Type,
               "Laser",    null, Laser'access,
               "Ink jet",  null, Ink_Jet'access );
    Framework( Menu_Bar,
               "About",    null, About'access,
               "Print",    Printer_Type'Unchecked_Access, null );
    Window_Start;
  end;
  Window_Epilogue;
end Main;
```

*Note:*    The call-back functions *Laser*, *Ink\_Jet*, and *About* process a user request when the appropriate menu option is selected.

*The use of 'Unchecked\_Access to deliver the access value of printer\_type.*

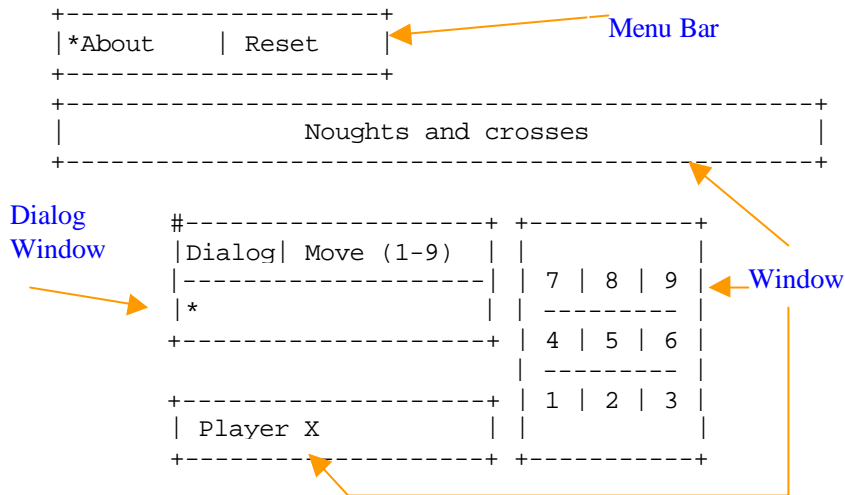
When the above code is incorporated into a program the menu system would display as follows:

Main menu bar	Secondary menu bar
<div style="border: 1px solid black; padding: 5px; margin: 5px;"> <pre>+-----+   About    *Print   +-----+</pre> </div>	<div style="border: 1px solid black; padding: 5px; margin: 5px;"> <pre>+-----+   *Laser   Ink jet   +-----+</pre> </div>

*Note:*    The secondary menu bar will overwrite the main menu bar regardless of the number of items in the main menu.

## 22.5 Noughts and crosses program

Using the GUI system the following layout can be created to graphically represent the game of noughts and crosses.



The layout uses most of the components in the GUI system. These are:

- A menu bar.  
Used to help control the game allows the user to select who starts and details about the game.
- A Dialog window.  
Used by the player of the game to enter moves.
- Several Windows that display information about the current state of the game.

### 22.5.1 The class Board

A program to play the game of noughts and crosses using the TUI API interface is composed of a class Board that has the following methods:

Method	Responsibility
Add	Add a piece to the board.
Reset	Reset the board to empty.
State	Return the state of the board.
Update	Update onto a window the state of the board.
Valid	Check if the move is valid.

The Ada specification for the class Board is:

```

package Class_Board is

    type Board          is private;
    type Game_State is ( Win, Playable, Draw );

    procedure Add( The:in out Board; Pos:in Integer;
                  Piece:in Character );
    function Valid( The:in Board; Pos:in Integer ) return Boolean;
    function State( The:in Board ) return Game_State;
    function Cell( The:in Board; Pos:in Integer ) return Character;
    procedure Reset( The:in out Board );
private
    subtype Board_Index is Integer range 1 .. 9;
    type Board_Array is array( Board_Index ) of Character;
    type Board is record
        Sqrns : Board_Array := ( others => ' ');  --Initialize
        Moves : Natural      := 0;
    end record;
end Class_Board;

```

*Note: It would have been elegant to re-use the previous noughts and crosses class for Board but the functionality is too dissimilar to make this a practical proposition. Code re-use is not always possible!*

In the implementation of the class the function Valid returns True if the suggested noughts and crosses move is valid.

```

package body Class_Board is

    function Valid(The:in Board; Pos:in Integer) return Boolean is
    begin
        return Pos in Board_Array'Range and then
            The.Sqrns( Pos ) = ' ';
    end Valid;

```

The procedure Add is responsible for adding a new piece onto the board..

```

procedure Add( The:in out Board; Pos:in Integer;
              Piece:in Character ) is
begin
    The.Sqrns( Pos ) := Piece;
end Add;

```

The function State returns the enumeration Win, Draw or Playable depending on the current state of the game. The constant array cells holds all possible win lines and the main body of the code uses the values held in this array to determine the current state of the game.

```

function State( The:in Board ) return Game_State is
  subtype Position is Integer range 1 .. 9;
  type Win_Line is array( 1 .. 3 ) of Position;
  type All_Win_Lines is range 1 .. 8;
  Cells: constant array ( All_Win_Lines ) of Win_Line :=
    ( (1,2,3), (4,5,6), (7,8,9), (1,4,7),
      (2,5,8), (3,6,9), (1,5,9), (3,5,7) ); --All win lines
  First : Character;
begin
  for Pwl in All_Win_Lines loop
    --All Pos Win Lines
    First := The.Sqrs( Cells(Pwl)(1) ); --First cell in line
    if First /= ' ' then
      -- Looks promising
      if First = The.Sqrs(Cells(Pwl)(2)) and then
        First = The.Sqrs(Cells(Pwl)(3)) then return Win;
      end if;
    end if;
  end loop;
  if The.Moves >= 9 then
    --Check for draw
    return Draw;
    -- Board full
  else
    return Playable;
    -- Still playable
  end if;
end State;

```

The procedure Cell returns the contents of a cell on the playing board.

```

function Cell( The:in Board; Pos:in Integer ) return Character is
begin
  return The.Sqrs( Pos );
end Cell;

```

The procedure reset resets the board to its initial empty state.

```

procedure Reset( The:in out Board ) is
begin
  The.sqrs := ( others => ' '); --All spaces
  The.moves := 0; --No of moves
end reset;

end Class_Board;

```

### 22.5.2 Package Pack\_Program

The package Pack\_Program contains the publicly visible procedure Play that will play the GUI based game against two human opponents. In the private part of the package, the procedure Play is broken down into further procedures and functions. The variables in the private part of the specification will be visible to all these procedure and functions.

These variables define windows that are used in the playing of the game of noughts and crosses.



```

with Class_Board, Class_Window;
use   Class_Board, Class_Window;
package Pack_Program is
  procedure Play;
private
  Game      : Board;      --The board
  P_Win_Brd : P_Window;   --Window to display OXO board in
  P_Win_Bnr : P_Window;   --Window to display Banner in
  P_Win_R   : P_Window;   --Window to display commentary in
  Player    : Character;  --Either 'X' or 'O'
end Pack_Program;

```

The private part of the package is as follows:

```

with Ada.Integer_Text_Io,
      Class_Dialog, Class_Menu, Class_Input_Manager, Class_Menu_Title;
use   Ada.Integer_Text_Io,
      Class_Dialog, Class_Menu, Class_Input_Manager, Class_Menu_Title;
package body Pack_Program is

```

The procedure Play sets up the various windows used to display the game.

```

procedure Play is
begin
  Window_Prologue;      --Setup window system
  declare
    Win_Brd  : aliased Window; --Board Window
    Win_R    : aliased Window; --Result Window
    Win_Bnr  : aliased Window; --title Window
    Win_Usr  : aliased Dialog; --Input Window
    Ttt_Reset: aliased Menu;   --Reset menu
    Ttt_Menu : Menu_Title;     --Title menu

```

The various windows on the screen are then initialized to their fixed co-ordinate positions.

```

begin
  Framework( Win_Bnr, 1, 4, 52, 3 ); --Banner
  Framework( Win_Brd, 32, 8, 13, 9 ); --OXO board
  Framework( Win_R, 9, 14, 22, 3 ); --Results

```

The menu bar sequence is then defined with the following frameworks:

```

Framework( Ttt_Reset,
  "X start", null, Reset_X'access,
  "O start", null, Reset_O'access );

Framework( Ttt_Menu,
  "About", null, About'access,
  "Reset", Ttt_Reset'Unchecked_Access, null );

```

Following the initialization of global variables the writing of various introductory messages is performed:

## 324 A Text user interface

```
Position( Win_Bnr, 17, 1 );
Put( Win_Bnr, "Noughts and crosses" );

Framework( Win_Usr, 9, 8, 22,
           "Move (1-9)", User_Input'access );

Player := 'X';                                --Set player
P_Win_Brd := Win_Brd'Unchecked_Access;      --OXO Board
P_Win_Bnr := Win_Bnr'Unchecked_Access;      --Banner
P_Win_R   := Win_R'Unchecked_Access;        --Commentary

Display_Board( P_Win_Brd );                  --Empty board
New_Line( Win_R );                          --Clear
Put( Win_R, " Player " & Player );          --Players turn is

Put( Win_Usr, " " );                        --Cursor
```

The user is only then allowed to start playing the game.

```
Window_Start;                                --Start the user interaction
end;
Window_Epilogue;                             --Close window system
end Play;
```

The procedure `Display_Board` writes the initial representation of the board into a GUI window on the screen.

```
procedure Display_Board( Win:in P_Window ) is
begin
  Position( Win.all, 1, 2 );
  Put(Win.all, " 7 | 8 | 9" ); New_Line( Win.all );
  Put(Win.all, " -----" ); New_Line( Win.all );
  Put(Win.all, " 4 | 5 | 6" ); New_Line( Win.all );
  Put(Win.all, " -----" ); New_Line( Win.all );
  Put(Win.all, " 1 | 2 | 3" ); New_Line( Win.all );
end Display_Board;
```

The procedure `Update` updates the representation of the board by adding the current move to it. Rather than re-display the board in its entirety only the square that has changed is re-written.

```
procedure Update( Move:in Integer; Win:in P_Window ) is
  type Co_Ordinate is ( X , Y );
  type Cell_Pos is array ( Co_Ordinate ) of Positive;
  type Board is array ( 1 .. 9 ) of Cell_Pos;
  Pos: constant Board := ( (2,6), (6,6), (10,6),
                           (2,4), (6,4), (10,4),
                           (2,2), (6,2), (10,2) );
begin
  Position( Win.all, Pos(Move)(X), Pos(Move)(Y) );
  Put( Win.all, Cell( Game, Move ) );      --Display counter;
end Update;
```

The function `User_Input` is a call-back function that is called when a player has entered their move into the Dialog window

```

function User_Input( Cb_Mes:in String ) return String is
    Move: Integer; Last: Positive;
begin
    Clear( P_Win_R.all );           --Clear
    Get( Cb_Mes, Move, Last );      --to int
    if Valid( Game, Move ) then    --Valid
        Add( Game, Move, Player ); --to board
        Update( Move, P_Win_Brd );
        case State( Game ) is      --Game is
            when Win =>
                Put( P_Win_R.all, " " & Player & " wins" );
            when Playable =>
                case Player is      --Next player
                    when 'X' => Player := 'O'; -- 'X' => 'O'
                    when 'O' => Player := 'X'; -- 'O' => 'X'
                    when others => null;      --
                end case;
                Put( P_Win_R.all, " Player " & Player );
            when Draw =>
                Put( P_Win_R.all, " It's a draw " );
            end case;
        else
            Put( P_Win_R.all, " " & Player & " Square invalid" );
        end if;
        return "";
    exception
        when others =>
            Put( P_Win_R.all, " " & Player & " re-enter move" );
            return "";
    end User_Input;

```

*Note:* The exception is used to handle invalid input from a user.

The menu system has three call-back functions, the first and second (Reset\_X and Reset\_O) reset the board to empty and start the game for either X or O.

```

procedure Re_Start( First_Player:in Character ) is
begin
    Player := First_Player;          --Start with
    Reset( Game );                   --Reset Board
    Display_Board( P_Win_Brd );      --Display
    Clear( P_Win_R.all );            --Status info
    Put( P_Win_R.all, " Player " & Player ); --Player name
end Re_Start;

function Reset_X( Cb_Mes:in String ) return String is
begin
    Re_Start( 'X' ); return "";
end Reset_X;

function Reset_O( Cb_Mes:in String ) return String is
begin
    Re_Start( 'O' ); return "";
end Reset_O;

```

*Note:* The common code is factored out in the procedure re\_start.

The third call-back function displays information about the program in the window represented by the handle P\_Win\_Bnr.

```

function About( Cb_Mes:in String ) return String is
begin
  Clear( P_Win_Bnr.all ); Position( P_Win_Bnr.all, 17, 1 );
  Put( P_Win_Bnr.all, "Written in Ada 95");
  return " ";
end About;

```

### 22.5.3 Putting it all together

When compiled and linked with the TUI API code the opening screen layout is as follows:

```

+-----+
|*About   | Reset   |
+-----+
+-----+-----+
|                                     |
|                               Noughts and crosses                               |
|                                     |
+-----+-----+

#-----+-----+
|Dialog| Move (1-9) | | 7 | 8 | 9 |
|-----+-----+ | |-----+
|*      |          | | 4 | 5 | 6 |
+-----+-----+ | |-----+
|                                     |
| Player X          | | 1 | 2 | 3 |
|-----+-----+ | |-----+
+-----+-----+

```

After the following moves have been made:

X's move	Commentary	O's move	Commentary
5	Claim the centre square	2	Not the correct move
9	Setting up a win	3	Block the X's
1	Two win lines	9	Block one of them
4	Win with three X's		

the screen layout will be:

```

+-----+
|*About   | Reset   |
+-----+
+-----+-----+
|                                     |
|                               Noughts and crosses                               |
|                                     |
+-----+-----+

#-----+-----+
|Dialog| Move (1-9) | | X | 8 | O |
|-----+-----+ | |-----+
|*      |          | | X | X | 6 |
+-----+-----+ | |-----+
|                                     |
| X Wins          | | X | O | O |
|-----+-----+ | |-----+
+-----+-----+

```

## 22.6 Self-assessment

- What is a call-back function and how is it used?
- How might the reversi program shown in Chapter 8 be modified so that it can be used with the TUI interface? To what extent is code re-use possible from the original version?

## 22.7 Exercises

Construct the following programs:

- *Currency converter*  
A program to allow a user to convert between two currencies. The program should allow the input of the current currency rate.
- *Reversi program*  
Re-implement the reversi program to use the TUI interface or any graphical interface that is available.
- *Draughts program*  
Implement the game of draughts to use the TUI interface or any graphical interface that is available.

## 23 TUI: the implementation

This chapter looks at the implementation of the TUI (Text User Interface) described in Chapter 21. An ANSI terminal or environment that supports ANSI terminal emulation is used as the output screen on which the TUI is implemented.

### 23.1 Overview of the TUI

The TUI is composed of the following window types: Window, Dialog, Menu and Title\_menu. These windows have the following properties:

Type of window	Explanation
Window	A plain scrolling window into which text can be written.
Dialog	A dialog window into which the user can enter text. The text is passed to a call-back function.
Menu	A menu pane, from which a user can select a menu option. The menu option selected either calls a call-back function or selects a new menu pane.
Title_menu	The root of a series of menu panes which overlay each other.

*Note:* A call-back function is an Ada function called in response to user input.

The relationship between the different types of windows is shown in the inheritance hierarchy illustrated in Figure 23.1.

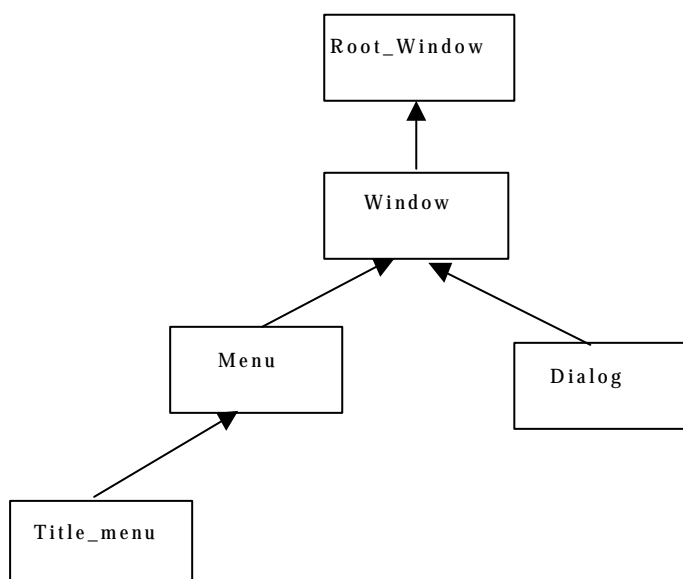


Figure 23.1 Inheritance diagram of the types of window in the TUI.

### 23.1.1 Structure of the TUI

In essence the TUI is composed of the following components:

Component	Description
Windows	A collection of heterogeneous windows created in the application program.
Event loop	The main processing loop for the program. The event loop obtains the next character from the user and determines which window this should be passed on to.
Display	The displayable representation of the windows in the system. This is an ANSI terminal compatible display area.

The relationship between these components is illustrated in Figure 23.2.

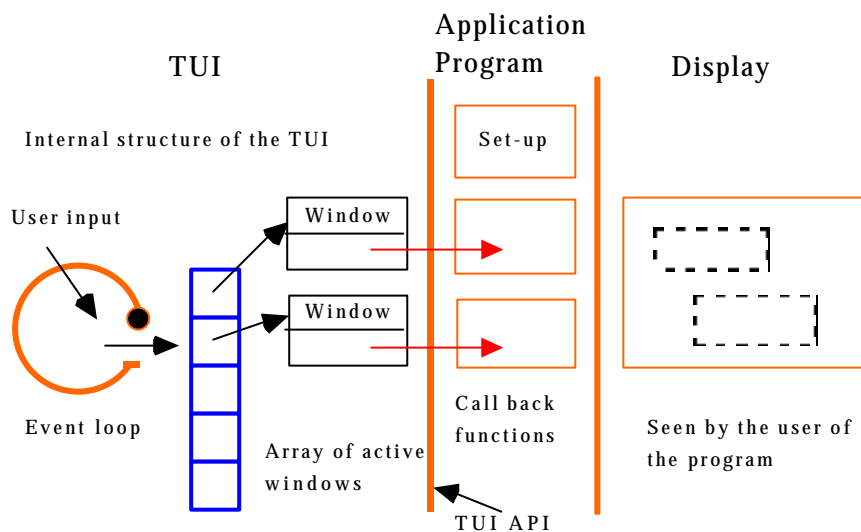


Figure 23.2 Structure of a program using the TUI.

A user of a program which employs the TUI classes interacts with a program by looking at the display, selecting a window capable of receiving input and then sending one or more character(s) to this window. The window selected by the user processes each character independently. For example, a dialog window will accumulate the characters sent to it, displaying the characters as they are received. Then, when the end of message character is received, the whole text string is passed to the call-back function associated with the dialog window. The call-back function implements a service requested by the user.

An application programmer using the TUI's API to implement a program, first constructs the windows used in the program. The application programmer then passes control to the input manager. The input manager accepts input from the user of the application and sends the input onto the window that has the input focus. Text sent to a window may cause the execute of a call-back function provided by the application programmer. The call-back functions implement the functionality of the program.

## 23.2 Implementation of the TUI

At the heart of the TUI is an event loop that receives characters from the user and dispatches the received characters to the appropriate window. This process is managed by an instance of the class `Input_manager`.

The input manager accesses the individual windows using an instance of the class `Window_control`. An instance of class `Window_control` stores windows in a linear list, the top window in the list representing the window that has the input focus.

Associated with each window is its switch character. A switch character typed by the user selects the window associated with this switch character as the window for input focus. As several windows may have the same switch character, the search mechanism will cycle through windows that have the same switch character.

If the typed character is not a switch character, the character is sent to the window that is the focus for the input.

## 330 TUI the implementation

### 23.2.1 Constants used in the TUI

The TUI uses several constants to define the state of the TUI and its environment. The size of the screen and the maximum size of windows created on the screen are defined by:

```
package Pack_Constants is
  Vdt_Max_X      : constant := 79;      --Columns on VDT
  Vdt_Max_Y      : constant := 25;      --Lines on VDT
  Window_Max_X   : constant := 79;      --MAX columns window
  Window_Max_Y   : constant := 25;      --MAX lines window
```

Various special characters that can be sent to or are used by the TUI, have the following values:

```
C_Cursor      : constant Character := '*';
C_Blank        : constant Character := ' ';
C_Win_A        : constant Character := '#';
C_Win_Pas      : constant Character := '+';
C_Exit         : constant Character := Character'Val(05); --^E
C_Where        : constant Character := Character'Val(255);
C_Action       : constant Character := Character'Val(13); --cr
C_Switch       : constant Character := Character'Val(09); --ht
C_Menu         : constant Character := Character'Val(27); --esc
C_Del          : constant Character := Character'Val(08); --^B
```

Various internal states and representations of actions are defined in the list below. In this list, the arrow keys that a user presses to interact with the TUI are internally defined as a single character. This is to simplify the internal code that processes the key's representations.

```
C_No_Char      : constant Character := Character'Val(00);

C_Left         : constant Character := Character'Val(12); --^L
C_Right        : constant Character := Character'Val(18); --^R
C_Up           : constant Character := Character'Val(21); --^U
C_Down         : constant Character := Character'Val(04); --^D
end Pack_Constants;
```

### 23.2.2 Raw input and output

The TUI works on the assumption that a character is sent immediately to the screen without any internal processing or buffering. Likewise, the TUI receives each character as it is typed without any internal buffering or processing.

An Ada package specification and possible implementation are shown below. In this implementation, a library procedure C\_No\_Echo written in the language C is used to turn off the echoing of the character that is read immediately from the keyboard.



```

package Raw_Io is
  procedure Get_Immediate( Ch:out Character );
  procedure Put( Ch:in Character );
  procedure Put( Str:in String );
end Raw_Io;

with Interfaces.C, Ada.Text_IO;
use Interfaces.C, Ada.Text_IO;
package body Raw_Io is

  First_Time : Boolean := True;

  procedure Get_Immediate( Ch:out Character) is
    procedure C_No_Echo;
    pragma Import (C, C_No_Echo, "c_no_echo");    --Turn off echo
  begin
    if First_Time then
      C_No_Echo; First_Time := False;
    end if;
    Ada.Text_IO.Get_Immediate(Ch);
    if Character'Pos(Ch) = 10 then                --Real Return ch
      Ch := Character'Val(13);
    end if;
  end Get_Immediate;

```

```

  procedure Put( Ch:in Character ) is            --Raw write
  begin
    Ada.Text_IO.Put( Ch ); Ada.Text_IO.Flush;
  end Put;

  procedure Put( Str:in String ) is             --Raw write
  begin
    Ada.Text_IO.Put( Str ); Ada.Text_IO.Flush;
  end Put;

end Raw_Io;

```

Note: The Ada reference manual does not define whether `get_immediate` echoes the read character. Chapter 25 describes how this package `raw_io` may be written in another language.

### 23.2.3 Machine-dependent I/O

In association with the package `raw_io`, the package `Pack_md_io` provides higher level machine specific input procedures. For output, these allow the use of the overloaded procedures `put` on a character and a string. For input, the responsibility is slightly more complex as the arrow keys are mapped onto an internal representation. The responsibilities for the procedures in the package `Pack_md_io` are:

Procedure	Responsibility
<code>Put( Ch :in Character );</code>	Write Ch immediately to the output screen.
<code>Put( Str:in String );</code>	Write Str immediately to the output screen.
<code>Get_Immediate ( ch:out Character );</code>	Read a character immediately from the keyboard. Do not echo this character onto the screen.

The specification for this package is:

```

package Pack_Md_Io is
  procedure Put( Ch :in Character );            --Put char
  procedure Put( Str:in String );              --Put string
  procedure Get_Immediate( Ch:out Character ); --no echo
end Pack_Md_Io;

```

## 332 TUI the implementation

The implementation code for `Get_Immediate` is shown mapping the three-character ANSI sequence for the arrow keys into an internal single character to representation. The actual character (s) generated will depend on the operating environment. The arrow key presses are mapped upon input into the character constants `C_LEFT`, `C_RIGHT`, `C_UP` and `C_DOWN` so that they can be processed in the program in a machine-independent way.

```
with Raw_Io, Pack_Constants;
use Raw_Io, Pack_Constants;
package body Pack_Md_Io is
  procedure Put( Ch:in Character ) is
  begin
    Raw_Io.Put( Ch );
  end Put;

  procedure Put( Str:in String ) is
  begin
    Raw_Io.Put( Str );
  end Put;
```

```
procedure Get_Immediate( Ch:out Character) is
  Esc: constant Character := Character'Val(27);
begin
  Raw_Io.Get_Immediate( Ch );
  if Ch = Esc then
    Raw_Io.Get_Immediate( Ch );
    if Ch = '[' then
      Raw_Io.Get_Immediate( Ch );
      case Ch is
        when 'A' => Ch := C_Up;
        when 'B' => Ch := C_Down;
        when 'C' => Ch := C_Right;
        when 'D' => Ch := C_Left;
        when others => Ch := '?';
      end case;
    end if;
  end if;
end Get_Immediate;

end Pack_Md_Io;
```

Note: In the implementation of `get_immediate` the arrow keys are converted into an internal single character. The implementation for `Raw_Io` that I used returns three characters when an arrow key is pressed.

One way of simplifying this procedure is to make the user of the TUI use the following control keys for arrow movements.

Character	Meaning
<code>^L</code>	Same as left arrow key
<code>^R</code>	Same as right arrow key
<code>^U^D</code>	Same as up arrow / down arrow key

These definitions can, of course, be changed by modifying the definitions of `C_Up` etc. in the package `Pack_Constants`.

### 23.2.4 The class Screen

The package `Class_screen` implements cursor positioning and output to an ANSI compatible display screen. The responsibilities of the class are:

Method	Responsibility
<code>Clear_Screen</code>	Clears all the screen.
<code>Position_Cursor</code>	Position the cursor at x, y on the screen.
<code>Put</code>	Write information to the current position on the screen.

Note: The co-ordinate system for the screen is shown in Figure 23.3.

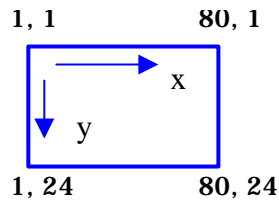


Figure 23.3 Co-ordinate system for the screen.

The class specification for `Class_screen` is:

```
package Class_Screen is
  procedure Put( Ch :in Character );      --Put char
  procedure Put( Str:in String );        --Put string
  procedure Clear_Screen;                --Clear screen
  procedure Position_Cursor(Col:in Positive; Row:in Positive);
private
end Class_Screen;
```

Note: As there is only one instance of a screen the class `Screen` contains all class methods.

The implementation of the class `Screen` uses the standard ANSI escape sequence to position the cursor onto a text terminal. The overloaded procedures `put` call the relevant `put` procedure in `Pack_md_io`.

```
with Pack_Md_Io; use Pack_Md_Io;
package body Class_Screen is
  Prefix: constant String := Character'Val(27) & "[";
  procedure Put( N:in Positive );      --Write decimal number

  procedure Put( Ch :in Character ) is
  begin
    Pack_Md_Io.Put( Ch );
  end Put;

  procedure Put( Str:in String ) is
  begin
    Pack_Md_Io.Put( Str );
  end Put;
```

If an ANSI terminal is not available then the bodies of the procedures `Clear_Screen` and `Position_Cursor` will need to be amended to reflect the characteristics of the user's terminal or output environment.

## 334 TUI the implementation

```
procedure Clear_Screen is          --Clear screen
begin
    Put( Prefix & "2J");
end Clear_Screen;

procedure Position_Cursor(Col:in Positive; Row:in Positive) is
begin
    Put( Prefix ); Put(Row); Put(";"); Put(Col); Put("H");
end Position_Cursor;
```

The procedure Put, when used to write a positive number without any leading or trailing spaces, is implemented as a recursive procedure. This procedure is used in the package by the public procedure Position\_Cursor.

```
procedure Put( N:in Positive ) is  --Write decimal number
begin
    if N >= 10 then Put( N / 10 ); end if;
    Put( Character'Val(N rem 10 + Character'Pos('0') ) );
end Put;

end Class_Screen;
```

### 23.3 The class Root\_window

A root window is the class from which all other windows are eventually derived. Its purpose is to define the minimum responsibilities that any type of window must implement. These minimum responsibilities are:

Method	Responsibility
Send_To	Send a character to the window.
Switch_To	Inform the window that it is to become the focus for input.
Switch_Away	Inform the window that it is to lose the focus of input.
Refresh	If appropriate, re-display the window.
About	Return information about a window.

*Note: There is no requirement that a window has a physical form. Thus, a root window does not need to provide a mechanism for writing into the displayable window.*

As no concrete instance of a root window is required, the specification for the class Root\_window is abstract. This abstract class will be specialized into the various forms of displayable windows on the screen.

```

with Ada.Finalization;
use  Ada.Finalization;
package Class_Root_Window is
  type Root_Window is abstract tagged limited private;
  type P_Root_Window is access all Root_Window'Class;
  type Attribute is ( Top, Bottom, Left, Right, Abs_X, Abs_Y );

  procedure Send_To( The:in out Root_Window;
    Ch:in Character) is abstract;
  procedure Switch_To( The:in out Root_Window ) is abstract;
  procedure Switch_Away( The:in out Root_Window ) is abstract;
  function About( The:in Root_Window;
    B:in Attribute) return Natural is abstract;
private
  type Root_Window is
    abstract new Limited_Controlled with null record;
end Class_Root_Window;

```

## 23.4 The classes Input\_manager and Window\_control

### 23.4.1 Specification of the class Input\_manager

The input manager controls all interactions between a user and the displayed windows. Currently the only input device is the keyboard. The responsibilities of the input manager are:

Method	Responsibility
Window_Prolog	Set up the initial window environment.
Window_Start	Start the windowing system by accepting input from a user of the TUI.
Window_Epilog	Close down the windowing system.

The Ada specification for the class Input\_manager is:

```

with Ada.Finalization;
use  Ada.Finalization;
package Class_Input_Manager is
  type Input_Manager is abstract tagged limited private;
  procedure Window_Prologue;           --Initialize window system
  procedure Window_Start;              --Start taking user input
  procedure Window_Epilogue;           --Clean up
private
  type Input_Manager is
    abstract new Limited_Controlled with null record;
end Class_Input_Manager;

```

*Note: As there is only one screen the class Input\_Manager has all class methods.*

### 23.4.2 Specification of the class Window\_control

The class Window\_Control has overall control of the windows displayed on the TUI screen. The responsibilities of this class are:

Method	Responsibility
Add_To_List	Add a new window to the list of managed windows.
Remove_From_List	Remove a window from the list of managed windows.

Top	Make the supplied window the top window. The top window in the list is the focus for input.
Find	Search the controlled windows for a window which has the supplied character as its switch character.
Send_To_Top	Send a character to the topmost window
Switch_To_Top	Prepare the top window as the focus of input.
Switch_Away_From_Top	Prepare a window to have the focus of input removed from it.
Write_To	Write to supplied window. Information has already been clipped to fit into the window.
Hide_Win	Remove the window from the screen.
Window_Fatal	Report a serious error in the TUI system.

The Ada specification for this class is:

```
with Ada.Finalization, Class_Root_Window;
use  Ada.Finalization, Class_Root_Window;
package Class_Window_Control is

  type Window_Control is abstract tagged limited private;
  procedure Add_To_List(P_W:in P_Root_Window; Ch:in Character);
  procedure Remove_From_List( P_W:in P_Root_Window );
  procedure Top( P_W:in P_Root_Window );
  procedure Find( P_W:out P_Root_Window; Ch:in Character );

  procedure Send_To_Top( Ch:in Character );
  procedure Switch_To_Top;
  procedure Switch_Away_From_Top;

  procedure Write_To( P_W:in P_Root_Window;
    X,Y:in Positive; Mes:in String );
  procedure Hide_Win( P_W:in P_Root_Window );
  procedure Window_Fatal( Mes:in String );
```

```
private
  type Window_Control is
    abstract new Limited_Controlled with null record;
  Max_Items : constant := 10;
  type Active_Window is record
    P_W : P_Root_Window;
    A_Ch: Character;
  end record;

  subtype Window_Index is Natural      range 0 .. Max_Items;
  subtype Window_Range is Window_Index range 1 .. Max_Items;
  type Window_Array is array (Window_Range) of Active_Window;

  The_Last_Win: Window_Index := 0;
  The_Windows : Window_Array;
end Class_Window_Control;
```

*Note:* As there is only one screen, the class `Window_Control` has all class methods.

Associated with each window is its switch character. When typed by a user this switch character activates the window as the focus for input.

### 23.4.3 Implementation of the class `Input_manager`

The Input\_manager is started by window\_prolog which clears the screen ready for the construction of the individual windows.

```
with Pack_Constants, Pack_Md_Io, Class_Screen,
     Class_Window_Control, Class_Root_Window;
use   Pack_Constants, Pack_Md_Io, Class_Screen,
     Class_Window_Control, Class_Root_Window;
package body Class_Input_Manager is

  procedure Window_Prologue is
  begin
    Clear_Screen;
  end Window_Prologue;
```

The procedure Window\_Start starts the window system by accepting input from the user and sending this input a character at a time to the active window. The input character is first tested (using the procedure find) to see if it is a window switch character. If it is, then the selected window is made the new focus for input.

```
procedure Window_Start is
  P_W : P_Root_Window;           --A window
  Ch  : Character;               --Current Char
begin
  loop
    Get_Immediate( Ch );         --From Keyboard
    exit when Ch = C_Exit;
    Find( P_W, Ch );             --Active window
    if P_W /= null then         --Window activation
      Switch_Away_From_Top;     -- No longer active
      Top( P_W );               -- Make p_w top
      Switch_To_Top;            -- & make active
      Send_To_Top( C_Where );   --In selected window
    else                         --
      Send_To_Top( Ch );        --Give to top window
    end if;
  end loop;
  Pack_Md_Io.Put( Character'Val(0) ); --Capture output
end Window_Start;
```

The window epilog is currently a null procedure as no specific shutdown action is required.

```
procedure Window_Epilogue is
begin
  null;
end Window_Epilogue;
end Class_Input_Manager;
```

#### 23.4.4 Implementation of the class Window\_control

In the implementation of the class Window\_Control the managed windows are held in an array. If the user of the TUI creates too many windows then the procedure Window\_Fatal will be called. The procedure Add\_To\_List adds a new window to the list of controlled windows.

```

with Class_Screen;
use Class_Screen;
package body Class_Window_Control is

  procedure Add_To_List(P_W:in P_Root_Window; Ch:in Character) is
  begin
    if The_Last_Win < Max_Items then
      The_Last_Win := The_Last_Win + 1;
      The_Windows( The_Last_Win ) := ( P_W, Ch );
    else
      Window_Fatal("Cannot register window");
    end if;
  end Add_To_List;

```

A window is removed from the list of controlled windows by the following procedure:

```

procedure Remove_From_List( P_W:in P_Root_Window ) is
begin
  for I in 1 .. The_Last_Win loop
    if The_Windows( I ).P_W = P_W then
      for J in I .. The_Last_Win-1 loop
        The_Windows( J ) := The_Windows( J+1 );
      end loop;
      The_Last_Win := The_Last_Win - 1; exit;
    end if;
  end loop;
end Remove_From_List;

```

The procedure Top makes the supplied window P\_W the top window and hence the focus for input from a user.

```

procedure Top( P_W:in P_Root_Window ) is
begin
  for I in 1 .. The_Last_Win loop
    if The_Windows( I ).P_W = P_W then
      declare
        Tmp : Active_Window := The_Windows( I );
      begin
        for J in I .. The_Last_Win-1 loop
          The_Windows( J ) := The_Windows( J+1 );
        end loop;
        The_Windows( The_Last_Win ) := Tmp;
      end;
      exit;
    end if;
  end loop;
end Top;

```

The procedure Find searches the controlled windows for a window with the supplied Ch as its switch character. If Ch is a windows switch character then a pointer to the window is returned.



```

procedure Find( P_W:out P_Root_Window; Ch:in Character ) is
begin
    P_W := null;
    for I in 1 .. The_Last_Win loop
        if The_Windows( I ).A_Ch = Ch then
            P_W := The_Windows( I ).P_W;
            exit;
        end if;
    end loop;
end Find;

```

When a character is received from a user of the TUI, and it is not a window switch character, it is sent to the top window.

```

procedure Send_To_Top( Ch:in Character ) is
begin
    if The_Last_Win >= 1 then
        Send_To( The_Windows(The_Last_Win).P_W.all, Ch );
    end if;
end Send_To_Top;

```

When the focus of input is changed, the newly selected window is forewarned that it will become the focus for input by sending it the message `Switch_To_Top`. This allows the window to change its appearance to indicate that it is the current focus for input.

```

procedure Switch_To_Top is
begin
    if The_Last_Win >= 1 then
        Switch_To( The_Windows(The_Last_Win).P_W.all );
    end if;
end Switch_To_Top;

```

Likewise when the focus of input is taken away from a window it is forewarned by the message `Switch_Away_From_Top`.

```

procedure Switch_Away_From_Top is
begin
    if The_Last_Win >= 1 then
        Switch_Away( The_Windows(The_Last_Win).P_W.all );
    end if;
end Switch_Away_From_Top;

```

The procedure `Write_To` writes text to the physical screen. The window is interrogated for its absolute position on the screen so that the physical position to write the text to can be calculated. As no window currently overlaps, no extra clipping needs to be performed.

```

procedure Write_To( P_W:in P_Root_Window;
    X,Y:in Positive; Mes:in String ) is
    Abs_X_Crd : Positive := About( P_W.all, Abs_X );
    Abs_Y_Crd : Positive := About( P_W.all, Abs_Y );
begin
    Position_Cursor( Abs_X_Crd+X-1, Abs_Y_Crd+Y-1 );
    Class_Screen.Put( Mes );
end Write_To;

```

## 340 TUI the implementation

A window is removed from the screen with the procedure `Hide_Win`. As no windows overlap in this implementation, the area that the window occupies is overwritten with spaces.

```
procedure Hide_Win( P_W:in P_Root_Window ) is
  Abs_X_Crd : Positive := About( P_W.all, Abs_X );
  Abs_Y_Crd : Positive := About( P_W.all, Abs_Y );
  Width     : Positive := About( P_W.all, Top );
  Height    : Positive := About( P_W.all, Left );
  Spaces    : String( 1 .. Width ) := ( others => ' ' );
begin
  for H in 1 .. Height loop
    Position_Cursor( Abs_X_Crd, Abs_Y_Crd+H-1 );
    Class_Screen.Put( Spaces );
  end loop;
end Hide_Win;
```

The next procedure is concerned with processing a fatal error. The implementation simply writes the error message directly onto the TUI screen.

```
procedure Window_Fatal( Mes:in String ) is
begin
  Position_Cursor( 1, 1 );
  Put( "Window fatal error: "& Mes );
end Window_Fatal;

end Class_Window_Control;
```

## 23.5 Overlapping windows

Though not implemented here, only minor code changes are required to allow overlapping windows on the output screen. The order in which windows are held in the class attribute `the_windows` can be used to indicate the overlapping order as viewed on the screen. For example, the bottom window in the list `The_Windows` is overlapped by all other windows. The top window in the list overlaps all the other windows displayed on the screen. An implementation of overlapping windows requires extra code to be added to `Switch_To_Top`, `Write_To`, and `Hide_Win` in the class `Window_Control` to perform any necessary clipping.

## 23.6 The class Window

A window object has two main responsibilities:

- To provide an application program interface API for a user program.
- To provide a system API for the manipulation of a window.

### 23.6.1 Application API

The application API is available to an application program using the TUI to display and process information to and from a window.

Method	Responsibility
Clear	Clear the window to spaces.
Framework	Create the framework for a window.
Make_window	Make the window visible or invisible.

New_Line	Move to the next line in the window. This may involve a rack up of the text in the window.
Position	Move to a new position for subsequent output to the window.
Put	Write information into a window.

### 23.6.2 Window system API

The system API should not normally be required by an application program. This API is used internally by the TUI system to manage the windows on the screen.

Method	Responsibility
About	Return information about the window.
Call_Call_Back	Call the call-back function for this window.
Create	Create a raw window.
De_Register	De-register the window with the Input_manager.
Finalize	Destruction of a window.
Initialize	Controlled initialization of a window.
Mark_Border	Set border to indicate state of window active, or inactive.
Refresh	Re-display the window.
Register	Register window on screen.
Send_To	Send a character to the window for processing.
Set_Call_Back	Set a call-back function for this window.
Switch_Away	Make window non active.
Switch_To	Make window active.

### 23.6.3 The specification for the classWindow

The specification is as follows:

```

with Pack_Constants, Class_Root_Window, Class_Window_Control;
use Pack_Constants, Class_Root_Window, Class_Window_Control;
package Class_Window is
  type Window    is new Root_Window with private;
  type P_Window  is access all Window;

  type Mode      is ( Visible, Invisible );
  type P_Cbf     is access function(Str:in String) return String;

```

Construction of a window is performed by:

```

procedure Initialize( The:in out Window );
procedure Finalize( The:in out Window );

procedure Framework( The:in out Window;
                     Abs_X_Crd, Abs_Y_Crd: Positive;
                     Max_X_Crd, Max_Y_Crd: Positive;
                     Cb:in P_Cbf := null );

procedure Create    ( The:in out Window;
                     Abs_X_Crd, Abs_Y_Crd: Positive;
                     Max_X_Crd, Max_Y_Crd: Positive );

```

A call-back function is set and executed with:

## 342 TUI the implementation

```
procedure Set_Call_Back( The:in out Window; Cb:in P_Cbf );
function Call_Call_Back( The:in Window;
                        Str:in String ) return String;
```

User output to a window is written using:

```
procedure Put( The:in out Window; Mes:in String );
procedure Put( The:in out Window; Ch:in Character );
procedure Put( The:in out Window; N:in Integer );

procedure Position( The:in out Window; X,Y:in Positive );
procedure Clear( The:in out Window );
procedure New_Line( The:in out Window );
procedure Refresh( The:in out Window );

procedure Make_Window( The:in out Window; Mo:in Mode );
procedure Mark_Border( The:in out Window;
                      A_Border:in Attribute;
                      Pos:in Positive; Ch:in Character );
```

Details about a window are obtained using:

```
function About(The:in Window; B:in Attribute) return Natural;
```

The window is controlled by:

```
procedure Switch_Away( The:in out Window );
procedure Switch_To( The:in out Window );
procedure Send_To( The:in out Window; Ch:in Character );

procedure Register( P_W:in P_Root_Window; Ch:in Character );
procedure De_Register( P_W:in P_Root_Window );
```

The instance attributes of the class are:

```

private
  subtype Y_Cord is Positive range 1 .. Vdt_Max_Y;
  subtype X_Cord is Positive range 1 .. Vdt_Max_X;

  subtype Line_Index is X_Cord range 1 .. Window_Max_X;
  subtype Line_Range is Line_Index;
  subtype Line is String( Line_Range );

  subtype Pane_Index is Y_Cord range 1 .. Window_Max_Y;
  subtype Pane_Range is Pane_Index;
  type Pane_Array is array ( Pane_Range ) of Line;

  type Window is new Root_Window with record
    Abs_X : X_Cord := 1;    --The position on the vdt
    Abs_Y : Y_Cord := 1;    --The position on the vdt
    C_X : X_Cord := 1;      --Current position in window
    C_Y : Y_Cord := 1;      --Current position in window
    Max_X : X_Cord := 5;    --X size of window (+Border)
    Max_Y : Y_Cord := 5;    --Y size of window (+Border)
    Pane : Pane_Array;      --Copy of window in memory
    Mode_Of : Mode := Invisible; --Invisible window by default
    Call_Back: P_Cbf := null; --Call back function
  end record;
end Class_Window;

```

### 23.6.4 Implementation of the class Window

In the implementation of the class Window the procedure put is used internally to write to a specific area on the screen.

```

package body Class_Window is

  procedure Put( The:in out Window;
                X,Y:in Positive; Mes:in String );

```

The controlled procedure finalize removes a window from the screen and de-registers the window from the input manager.

```

  procedure Initialize( The:in out Window ) is
  begin
    null;
  end Initialize;

  procedure Finalize( The:in out Window ) is
  begin
    Make_Window( The, Invisible );
    De_Register( The'Unchecked_Access );
  end Finalize;

```

The procedure create sets up the window to be at a defined position on the screen. Some simple validation is performed. If this fails, the procedure windows\_fatal is called.

```

procedure Create( The:in out Window;
  Abs_X_Crd, Abs_Y_Crd: Positive;
  Max_X_Crd, Max_Y_Crd: Positive ) is
begin
  if Max_X_Crd < 3 or else Max_X_Crd > Window_Max_X or else
    Max_Y_Crd < 3 or else Max_Y_Crd > Window_Max_Y or else
    Abs_X_Crd + Max_X_Crd - 1 > Vdt_Max_X or else
    Abs_Y_Crd + Max_Y_Crd - 1 > Vdt_Max_Y then
    Window_Fatal( "Creation window parameter error" );
  end if;
  declare
    Top_Bottom: String(1..Max_X_Crd)      := (others => '-');
    Spaces      : String(2 .. Max_X_Crd-1) := (others => ' ');
  begin
    Top_Bottom(1) := '+'; Top_Bottom(Max_X_Crd) := '+';
    The.Max_X := Max_X_Crd - 2;                --For border
    The.Max_Y := Max_Y_Crd - 2;                --For border
    The.Abs_Y := Abs_Y_Crd;                    --Abs position screen
    The.Abs_X := Abs_X_Crd;                    --
    The.Pane(1)(1..Max_X_Crd) := Top_Bottom;  --Clear / set up
    for Y in 2 .. Max_Y_Crd-1 loop
      The.Pane(Y)(1..Max_X_Crd) := '|' & Spaces & '|';
    end loop;
    The.Pane(Max_Y_Crd)(1..Max_X_Crd) := Top_Bottom;
    Position( The, 1, 1 );                    --Top left hand corner
  end;
end Create;

```

The user callable procedure framework defines the position of the window on the screen. This procedure uses create to do most of the initialization of the window.

```

procedure Framework( The:in out Window;
  Abs_X_Crd, Abs_Y_Crd: Positive;
  Max_X_Crd, Max_Y_Crd: Positive;
  Cb:in P_Cbf := null ) is
begin
  Create( The, Abs_X_Crd, Abs_Y_Crd, Max_X_Crd, Max_Y_Crd );
  Make_Window( The, Visible );
  if Cb /= null then
    Set_Call_Back( The, Cb );
    Register( The'Unchecked_Access, C_Switch );
  else
    Register( The'Unchecked_Access, C_No_Char );
  end if;
end Framework;

```

The call-back function is set by set\_call\_back and is called via call\_call\_back.

```

procedure Set_Call_Back( The:in out Window; Cb:in P_Cbf ) is
begin
  The.Call_Back := Cb;
end Set_Call_Back;

function Call_Call_Back( The:in Window;
  Str:in String ) return String is
begin
  if The.Call_Back /= null then
    return The.Call_Back(Str);
  end if;
  return "No call back function";
end;

```

*Note:* The value returned by the call-back function is a string.

The procedure `put` writes text into the selected window from the currently selected position. If the text will not fit in the window it is clipped to fit the window. The text is then added to the stored image of the window. Then, if the window is visible it is written to the screen.

```

procedure Put( The:in out Window; Mes:in String ) is
  Add : Natural;
begin
  Add := Mes'Length;           --Length
  if Add + The.C_X > The.Max_X then   --Actual characters
    Add := The.Max_X - The.C_X + 1;   -- to add
  end if;
  if Add >= 1 then           --There are some
    The.Pane(The.C_Y+1)(The.C_X+1 .. The.C_X+Add)
      := Mes( 1 .. Add );
    if The.Mode_Of = Visible then   --Add to screen
      Put(The, The.C_X+1, The.C_Y+1, Mes( 1 .. Add) );
    end if;
    The.C_X := The.C_X + Add;
  else
    Put(The, The.C_X+1, The.C_Y+1, " " );
  end if;
end Put;

```

The two following procedures use the above `put` procedure to write a character and a natural number into the window:

```

procedure Put( The:in out Window; Ch:in Character ) is
begin
  Put( The, " " & Ch );           --Convert to string
end Put;

```

```

procedure Put( The:in out Window; N:in Integer ) is
begin
  Put( The, Integer'Image(N) );   --Convert to string
end Put;

```

The procedure `Clear` clears a window to spaces. The border of the window is, however, left intact.

```

procedure Clear( The:in out Window ) is
  Empty : String( 1 .. The.Max_X ) := ( others => ' ' );
begin
  Position(The, 1, 1);           --Top right hand corner
  for Y in 1 .. The.Max_Y loop   --Clear text
    Put( The, Empty ); New_Line(The);
  end loop;
end Clear;

```

The procedure `New_Line` implements the writing of a new line in the selected window. This may result in the information in the window being scrolled. Scrolling is implemented by refreshing the whole window after changing the contents of the remembered window.

```

procedure New_Line( The:in out Window ) is
begin
    if The.C_Y >= The.Max_Y then                --Scroll text
        for Y in 2 .. The.Max_Y loop            -- Copy up
            The.Pane(Y) := The.Pane(Y+1);
        end loop;
        The.Pane(The.Max_Y+1)(2..The.Max_X+1) := (others=>' ');
        Refresh(The);                            -- refresh
    else
        The.C_Y := The.C_Y + 1;                  --Next line
    end if;
    The.C_X := 1;                                --At start
end New_Line;

```

The procedure Position allows a user to set the current writing position in the window.

```

procedure Position( The:in out Window; X,Y:in Positive ) is
begin
    if X <= The.Max_X and Y <= The.Max_Y then
        The.C_X := X; The.C_Y := Y;
    end if;
end Position;

```

The procedure Refresh re-draws the whole of the window on the screen.

```

procedure Refresh( The:in out Window ) is
begin
    if The.Mode_Of = Visible then                --Visible
        for Y in 1 .. The.Max_Y+2 loop            --Text
            Put( The, 1, Y,
                The.Pane(Y)(1 .. The.Max_X+2) ); --include border
        end loop;
        Put( The, " " );                          --Cursor
    end if;
end Refresh;

```

A window can be made visible or invisible by the procedure make\_window.

```

procedure Make_Window( The:in out Window; Mo:in Mode ) is
begin
    if The.Mode_Of /= Mo then                    --Change so
        The.Mode_Of := Mo;                       --Set new mode_of
        case Mo is
            when Invisible =>                    --Clear from screen
                Hide_Win( The'Unchecked_Access ); --Hide window
            when Visible =>                      --Redraw on screen
                Refresh( The );
        end case;
    end if;
end Make_Window;

```

The style of the border may be changed by mark\_border. A window may be customized to a style to suit the user by using this procedure.



```

procedure Mark_Border( The:in out Window;
    A_Border:in Attribute;
    Pos:in Positive; Ch:in Character ) is
    A_Y, A_X : Positive;
begin
    case A_Border is
        when Top      => A_X := Pos; A_Y := 1;
        when Bottom  => A_X := Pos; A_Y := The.Max_Y+2;
        when Left    => A_X := 1; A_Y := Pos;
        when Right   => A_X := The.Max_X+2; A_Y := Pos;
        when others  => null;
    end case;
    if A_X <= The.Max_X+2 and then A_Y <= The.Max_Y+2 then
        The.Pane(A_Y)(A_X) := Ch;           --Store
        if The.Mode_Of = Visible then       --Update on screen
            Put( The, A_X, A_Y, Ch & " " );
            Put( The, " " );
        end if;
    end if;
end Mark_Border;

```

The procedure about returns details about the various attributes of a window.

```

function About(The:in Window; B:in Attribute) return Natural is
begin
    case B is
        when Top      | Bottom => return The.Max_X+2;
        when Left    | Right  => return The.Max_Y+2;
        when Abs_X    => return The.Abs_X;
        when Abs_Y    => return The.Abs_Y;
        when others   => return 0;
    end case;
end;

```

Whilst publicly visible, the following procedures are not intended to be used by an application programmer. These procedures are used by the event loop to allow a window to:

- Clean up before the focus of user input is removed from the window.
- Prepare for the focus of user input to be directed at the window.

The effect of these procedures is to mark the window with a visual indicator of its state.

```

procedure Switch_Away( The:in out Window ) is
begin
    Mark_Border( The, Top, 1, C_Win_Pas );
end Switch_Away;

procedure Switch_To( The:in out Window ) is
begin
    Mark_Border( The, Top, 1, C_Win_A );
end Switch_To;

```

When a user types a character which is not recognized by the system as a switch character it is sent to the window which has the focus for input. The procedure Send\_To receives this character. The procedure is simply null, because an instance of a Window does not process user input.

```

procedure Send_To( The:in out Window; Ch:in Character ) is
begin
    null;
end Send_To;

```

The window is registered with the input manager by the procedure `register` and de-registered with `De_Register`.

```

procedure Register( P_W:in P_Root_Window;
    Ch:in Character ) is
begin
    Switch_Away_From_Top;           --Register window focus
    Add_To_List( P_W, Ch );         --Register window
    Switch_To_Top;                  --Make focus
end Register;

procedure De_Register( P_W:in P_Root_Window ) is
begin
    Top( P_W );                    --Make top
    Switch_Away_From_Top;           -- prepare for demise
    Remove_From_List( P_W );        --De register window
    Switch_To_Top;                  --Make focus
end De_Register;

```

The next procedure is used internally by the class to write directly to a position in a window on the screen. This procedure uses `Write_To` in the class `Window_Control` to perform the actual write.

```

procedure Put( The:in out Window;
    X,Y:in Positive; Mes:in String ) is
begin
    Write_To( The.Unchecked_Access, X, Y, Mes );
end Put;

end Class_Window;

```

## 23.7 The class Dialog

A normal window is specialized to a dialog window by overloading the following windows methods with new responsibilities:

Method	Responsibility
Framework	Set up the window as a dialog box.
Send_To	Process user input sent to the dialog window.

The Ada specification for the class `Dialog` is as follows:

```

with Pack_Constants, Class_Root_Window, Class_Window;
use Pack_Constants, Class_Root_Window, Class_Window;
package Class_Dialog is
  type Dialog is new Window with private;

  procedure Framework ( The:in out Dialog;
    Abs_X, Abs_Y:in Positive;
    Max_X: in Positive;
    Name:in String; Cb:in P_Cbf );

  procedure Send_To( The:in out Dialog; Ch:in Character );
private
  subtype Message is String( 1 .. Window_Max_X );
  type Dialog is new Window with record
    Dialog_Pos: Positive := 1; --Position in input message
    Dialog_Len: Positive := 1; --Length of dialogue message
    Dialog_Mes: Message := ( others => ' '); --Input message
  end record;
end Class_Dialog;

```

### 23.7.1 Implementation of the class Dialog

The implementation of the class dialog is:

```

package body Class_Dialog is

```

The procedure Framework constructs the style of the dialog window and registers the window with the input manager so that user input may be directed to the window:

```

procedure Framework( The:in out Dialog;
  Abs_X, Abs_Y:in Positive;
  Max_X:in Positive;
  Name:in String; Cb:in P_Cbf ) is
  Dashes : String( 1 .. Max_X ) := (others=>' ');
begin
  Create( The, Abs_X, Abs_Y, Max_X, 5 );
  The.Dialog_Len := Max_X-2;           --User input
  The.Dialog_Pos := 1;                 --In Dialog
  Set_Call_Back( The, Cb );            --Call back fun
  Put( The, "Dialog| " ); Put( The, Name ); --Dialog title
  Position( The, 1, 2 ); Put( The, Dashes ); --Line
  Position( The, 1, 3 ); Put( The, C_Cursor );--Cursor
  Make_Window( The, Visible );
  Register( The'Unchecked_Access, C_Switch ); --Activation Chr
end Framework;

```

For example, the fragment of code:

```

declare
  Input : Dialog;           --Input Window
begin
  Framework( Input, 5, 10, 22, --Input Window
    "Miles", User_Input'access );
end;

```

would produce the following dialog box whose top left-hand corner on the screen is at position (5,10):

```

#-----+
|Dialog| Miles|
|-----+
|*|
+-----+

```

User input sent to the dialog window is processed by the procedure `send_to`. This stores characters in the string `Dialog_Mes`. When the user enters `C_ACTION` this causes a call to an application programmer written call-back function with the string `Dialog_Mes` as its parameter. The character `C_ACTION` is the normal Enter character on the keyboard.

```

procedure Send_To( The:in out Dialog; Ch:in Character ) is
  Spaces : String(1 .. About(Window(The),Top)) := (others => ' ');
  Res     : String(1..0);
begin
  case Ch is
    when C_Where =>
      Put( The, " ");
    when C_Action =>
      Res := Call_Call_Back( The,
        The.Dialog_Mes(1..The.Dialog_Pos-1) )(1..0);
      The.Dialog_Pos := 1;
      The.Dialog_Mes := (others => ' ');
      Position( The, 1, 3 );           --Start
      Put( The, C_Cursor & Spaces );   --Clear
      Position( The, 2, 3 );           --Cursor
      Put( The, " ");                 --Cursor
    when C_Del =>
      if The.Dialog_Pos > 1 then           --Can delete
        The.Dialog_Pos := The.Dialog_Pos - 1; --Make avail.
        The.Dialog_Mes(The.Dialog_Pos):= ' '; --Remove
        Position( The, The.Dialog_Pos, 3 );
        Put( The, C_Cursor & " " );       --Overwrite
        Position( The, The.Dialog_Pos, 3 );
        Put( The, " " );                 --Cursor
      end if;

```

```

    when others =>
      if The.Dialog_Pos <= The.Dialog_Len then
        if Ch in ' ' .. '~' then           --Add to
          The.Dialog_Mes( The.Dialog_Pos ) := Ch; --Save ch
          Position( The, The.Dialog_Pos, 3 );
          Put( The, The.Dialog_Mes(The.Dialog_Pos) );
          Put( The, C_Cursor );
          The.Dialog_Pos := The.Dialog_Pos + 1;
        end if;
      end if;
    end case;
  end Send_To;
end Class_Dialog;

```

## 23.8 The class Menu

A normal window is specialized to a menu window by overloading the following procedures:

Method	Responsibility
Framework	Set up the window as a menu window.
Send_To	Process user input sent to the menu window.

and adding the procedures:

Method	Responsibility
Set_Up	Set up the window as a menu window.
Menu_Spot	Highlight the selected menu item.

The specification of the class Menu is as follows:

```
with Class_Root_Window, Class_Window;
use Class_Root_Window, Class_Window;
package Class_Menu is
  type Menu is new Window with private;
  type P_Menu is access all Menu;

  procedure Framework( The:in out Menu'Class;
    M1:in String:=""; W1:in P_Menu:=null; Cb1:in P_Cbf:=null;
    M2:in String:=""; W2:in P_Menu:=null; Cb2:in P_Cbf:=null;
    M3:in String:=""; W3:in P_Menu:=null; Cb3:in P_Cbf:=null;
    M4:in String:=""; W4:in P_Menu:=null; Cb4:in P_Cbf:=null;
    M5:in String:=""; W5:in P_Menu:=null; Cb5:in P_Cbf:=null;
    M6:in String:=""; W6:in P_Menu:=null; Cb6:in P_Cbf:=null );
```

```
  procedure Set_Up( The:in out Menu; Active:in Positive);
  procedure Menu_Spot( The:in out Menu; Ch:in Character );
  procedure Send_To( The:in out Menu; Ch:in Character );

  Max_Menu : constant Positive := 10;
  subtype Menu_Item is String( 1 .. Max_Menu );

  procedure Get_Menu_Name( The:in Menu; I:in Positive;
    N:out Menu_Item );
  procedure Get_Cur_Selected_Details( The:in P_Menu;
    W:out P_Menu; Cb:out P_Cbf );
```

The private part of the specification contains details about how a menu item is stored. A menu consists of the names of menu items and associated with each name is either a call-back function or a link to another menu item.

```
private
  type Direction is (D_Reverse, D_Forward);
  procedure Next( The:in out Menu; Dir:in Direction );

  type Menu_Desc is record
    Name: Menu_Item;           --Name of menu item
    P_M : P_Menu;              --Menu window
    Fun : P_Cbf;               --Call back function
  end record;

  Max_Menu_Items : constant := 6;  --Maximum menu items

  type Menus_Index is range 0 .. Max_Menu_Items;
  subtype Menus_Range is Menus_Index range 1 .. Max_Menu_Items;
  type Menus is array ( Menus_Range ) of Menu_Desc;

  type Menu is new Window with record
    Number : Menus_Index := 0;  --Number of menu items
    Cur_Men : Menus_Index := 1; --Currently selected item
    Menu_Set : Menus;           --Components of a menu
  end record;
end Class_Menu;
```

### 23.8.1 Implementation of the class Menu

The implementation of the class is:

```
with Pack_Constants;
use Pack_Constants;
package body Class_Menu is
```

The procedure Set\_Up populates the displayed menu window with the names of the menu items.

```
procedure Set_Up( The:in out Menu;
                  Active:in Positive ) is
  Me: Menu_Item;
begin
  Create( The, 1, 1, (1+Max_Menu)*Active+1, 3 );
  for I in 1 .. Active loop      --Display menu names
    Get_Menu_Name( The, I, Me );
    Put( The, Me ); Put( The, "|" );
    null;
  end loop;
  Menu_Spot( The, C_Cursor );    --Mark current
end Set_Up;
```

In the procedure Framework the class type is described as Menu'Class. This is so that a run-time dispatch will be performed on inherited procedures or functions in any class derived from this class.

```
procedure Framework( The:in out Menu'Class;
  M1:in String:=" "; W1:in P_Menu:=null; Cb1:in P_Cbf:=null;
  M2:in String:=" "; W2:in P_Menu:=null; Cb2:in P_Cbf:=null;
  M3:in String:=" "; W3:in P_Menu:=null; Cb3:in P_Cbf:=null;
  M4:in String:=" "; W4:in P_Menu:=null; Cb4:in P_Cbf:=null;
  M5:in String:=" "; W5:in P_Menu:=null; Cb5:in P_Cbf:=null;
  M6:in String:=" "; W6:in P_Menu:=null; Cb6:in P_Cbf:=null
) is

  Spaces : Menu_Item := ( others => ' ' );
  Active : Menus_Index := 1;
  procedure Set_Up( Mi:in String; Wi:in P_Menu;
                   Cb:in P_Cbf; N:in Menus_Index ) is
  begin
    if Mi /= " " then Active := N; end if;    --A menu item
    The.Menu_Set( N ) :=
      ( " "&Mi&Spaces(1 .. Max_Menu-1-Mi'Length), Wi, Cb);
  end Set_Up;
begin
  Set_Up( M1, W1, Cb1, 1 ); Set_Up( M2, W2, Cb2, 2 );
  Set_Up( M3, W3, Cb3, 3 ); Set_Up( M4, W4, Cb4, 4 );
  Set_Up( M5, W5, Cb5, 5 ); Set_Up( M6, W6, Cb6, 6 );
  The.Number := Active;
  Set_Up( The, Positive(Active) );
end Framework;
```

*Note: The procedure set\_up which is called from within framework constructs the internal representation for the window.*

The procedure menu\_spot highlights the menu item selected.

```

procedure Menu_Spot( The:in out Menu; Ch:in Character ) is
begin
    Position( The, (Max_Menu+1)*(Positive(The.Cur_Men)-1)+1, 1 );
    Put( The, Ch );
end Menu_Spot;

```

When user input is focused at the menu window, the arrow keys cause a new menu item to be selected.

```

procedure Send_To( The:in out Menu; Ch:in Character ) is
begin
    Menu_Spot( The, C_Blank );
    case Ch is
        when C_Right => Next( The, D_Forward );
        when C_Left  => Next( The, D_Reverse );
        when others  => null;
    end case;
    Menu_Spot( The, C_Cursor );
end Send_To;

```

The actual calculation of the menu item selected is performed by the procedure next.

```

procedure Next( The:in out Menu; Dir:in Direction ) is
begin
    case Dir is
        when D_Forward =>
            The.Cur_Men := The.Cur_Men rem The.Number + 1;
        when D_Reverse =>
            if The.Cur_Men = 1
                then The.Cur_Men := The.Number;
            else The.Cur_Men := The.Cur_Men-1;
            end if;
    end case;
end Next;

```

The procedure get\_menu\_item returns the name of the menu item selected:

```

procedure Get_Menu_Name( The:in Menu; I:in Positive;
                        N:out Menu_Item ) is
begin
    N := The.Menu_Set( Menus_Index(I) ).Name;
end Get_Menu_Name;

```

whilst get\_cur\_selected\_details returns a pointer to the selected potential window and call-back function.

```

procedure Get_Cur_Selected_Details( The:in P_Menu;
    W:out P_Menu; Cb:out P_Cbf ) is
begin
    W := The.Menu_Set( The.Cur_Men ).P_M;
    Cb := The.Menu_Set( The.Cur_Men ).Fun;
end Get_Cur_Selected_Details;

end Class_Menu;

```

## 23.9 The class Menu\_title

A Menu window is specialized to a Menu\_title window by overloading the following procedures:

Method	Responsibility
Set_Up	Set up the window as a menu title window.
Send_To	Process user input sent to the menu title window.
Switch_Away	Return to the base window.

The Ada specification of the class is:

```
with Class_Root_Window, Class_Window, Class_Menu;
use   Class_Root_Window, Class_Window, Class_Menu;
package Class_Menu_Title is
  type Menu_Title is new Menu with private;
  type P_Menu_Title is access all Menu_Title;

  procedure Set_Up( The:in out Menu_Title; Active:in Positive );
  procedure Send_To( The:in out Menu_Title; Ch:in Character );
  procedure Switch_Away( The:in out Menu_Title );
private
  Max_Act_Menu : constant := 6;    --Maximum depth of menus
  type Act_Index is range 0 .. Max_Act_Menu;
  subtype Act_Range is Act_Index range 1 .. Max_Act_Menu;
  type Act_Menus is array ( Act_Range ) of P_Menu;

  type Menu_Title is new Menu with record
    Act_Menu : Act_Menus;           --Stack of displayed menus
    Menu_Index: Act_Index := 0;    --Top of menu stack
  end record;
end Class_Menu_Title;
```

### 23.9.1 Implementation of the class Menu\_title

In the implementation of the class Menu\_title:

```
with Pack_Constants;
use   Pack_Constants;
package body Class_Menu_Title is
```

the procedure set\_up is called from the inherited procedure framework in the class Menu. This is because the call of the procedure set\_up is a dispatching call. Remember, the first parameter to the procedure framework is of type Menu'Class.

```
procedure Set_Up( The:in out Menu_Title; Active:in Positive ) is
  Me: Menu_Item;
begin
  Create( The, 1, 1, (1+Max_Menu)*Active+1, 3 ); --Fixed size
  Make_Window( The, Visible );
  The.Act_Menu( 1 ) := Menu(The)'Unchecked_Access;--Title menu
  The.Menu_Index := 1;
  for I in 1 .. Active loop
    Get_Menu_Name( The, I, Me );           --Get menu
    Put( The, Me ); Put( The, "|" );       -- name
                                          -- write
  end loop;
  Register( The'Unchecked_Access, C_Menu ); --Register
  Menu_Spot( The, C_Cursor );             --Cursor on
end Set_Up;
```



The procedure `send_to` implements the selection of either a new menu bar or the call of a call-back function.

```

procedure Send_To( The:in out Menu_Title; Ch:in Character ) is
    Current, Next : P_Menu;
    Proc          : P_Cbf;
    Res           : String( 1..0 );
begin
    Current := The.Act_Menu( The.Menu_Index ); --Active menu
    Get_Cur_Selected_Details( Current, Next, Proc );
    case Ch is
        when C_Where =>
            Put( Current.all, " " );
        when C_Action =>
            if Next /= null and The.Menu_Index < Max_Act_Menu then
                Make_Window( Current.all, Invisible ); --Hide cur.
                The.Menu_Index := The.Menu_Index + 1; --
                The.Act_Menu( The.Menu_Index ) := Next; --New menu
                Make_Window( Next.all, Visible ); --Reveal
            else
                if Proc /= null then --Call
                    Res := Proc("Action")(1 .. 0 );
                end if;
            end if;
        when others =>
            Send_To( Current.all , Ch ); --Treat as normal menu
    end case;
end Send_To;

```

The procedure `Switch_Away` replaces the current menu with the top level menu bar. Naturally this replacement is only performed if the displayed menu is not the top level menu.

```

procedure Switch_Away( The:in out Menu_Title ) is
begin
    Mark_Border( The, Top, 1, C_Win_Pas ); --Now inactive
    if The.Menu_Index > 1 then --Not top level menu
        Make_Window( The.Act_Menu(The.Menu_Index).all, Invisible );
        The.Menu_Index := 1;
        Make_Window( The.Act_Menu( 1 ).all, Visible ); --Top level
    end if;
end Switch_Away;

end Class_Menu_Title;

```

## 23.10 Self-assessment

- What changes in the implementation of the TUI would be required to allow for overlapping windows?
- The TUI execution is currently serial, in that messages sent to a window are performed before control is returned to the input event loop. What would be the effect of letting code associated with a window execute as a separate task?

## 23.11 Exercises

Extend the TUI by providing the following new types of window:

- A window to which *Integer* and *Float* numbers may be written

## 356 TUI the implementation

This will allow the output of formatted numeric data as well as textual data.

- *A radio button dialog window*

This will allow a user to create programs in which one of several options may be selected. For example, the conversion program shown in Section 22.3 could allow the distance to be input in feet, yards or miles.

- *A check box dialog window*

This will allow a user to create programs in which several different options may be selected.

- *Noughts and crosses*

In the previous chapter, in Section 22.5, an example program to play the game noughts and crosses was shown. A user entering a square has to press Enter to have the move accepted. Devise and implement a new version where the keystroke for a position is sufficient to activate a call-back function in the application code.

- *Overlapping windows*

Modify the TUI so that overlapping windows are allowed. A user of the program should also be able to move the windows on the screen.

Build an application framework for the TUI.

- An application framework allows a user to design the layout of screens used in the program without having to write any code. The application framework will have an interface similar to a drawing editor and allows an application programmer to position the different types of window onto a screen. Then when the user is satisfied with the layout, the application framework program produces an Ada program skeleton of the application.

Graphical Representation

- Modify the TUI so that the screen display is more graphical. You may wish to add procedures and functions that allow for bit mapped data to be written to a window.

## 24 Scope of declared objects

This chapter describes the access rules for objects and types declared in an Ada program. In Ada the scope or visibility of an item depends on the current lexical position in which the item is introduced.

### 24.1 Nested procedures

In Ada, nesting of procedures to an arbitrary level is allowed. Each nested procedure introduces a new lexical level. For example, the following program is made up of two procedures `Outer` and `Inner`. The lexical level for each line is shown as a comment.

```
procedure Outer is      --+1
  Outer_Int : Integer;  -- 1
  procedure Inner is    -- 1
    Inner_Int: Integer; -- +2
  begin                -- 2
    Outer_Int := 1;     -- 2
    Inner_Int := 2;     -- 2
  end Inner;           -- -2
begin                  -- 1
  Outer_Int := 1;       -- 1
end Outer;             ---1
```

*Note:*    + indicates that a new lexical level has been started.  
          - indicates that the current lexical level is about to end.

The procedure `Outer` is at lexical level 1 and the procedure `Inner` is at lexical level 2. Code that is at a specific lexical level can access items declared either at that lexical level or declared at a lower surrounding lexical level. For example, in the procedure `Inner` the integer objects `Inner_Int` and `Outer_Int` can both be accessed. However, in the procedure `Outer` only the integer object `Outer_Int` can be accessed. Access to procedures and functions follow the same rules.

It is important to realize that only items that are in a surrounding lower lexical level may be accessed. For example, the following nonsensical program illustrates the variables and procedures that may be accessed at any point in the program.

```

procedure Proc_1_1 is                                --+1
  Int_1_1 : Integer;                                    -- 1
  procedure Proc_2_1 is                                -- 1
    Int_2_1 : Integer;                                  -- +2
    procedure Proc_3_1 is                                -- 2
      Int_3_1 : Integer;                                -- +3
      begin                                              -- 3
        Int_1_1 := 11; Int_2_1 := 21; Int_3_1 := 31; -- 3
        Proc_1_1; Proc_2_1; Proc_3_1;                  -- 3
      end Proc_3_1;                                     -- -3
    begin                                              -- 2
      Int_1_1 := 11; Int_2_1 := 21;                    -- 2
      Proc_1_1; Proc_2_1; Proc_3_1;                    -- 2
    end Proc_2_1;                                       -- -2
    procedure Proc_2_2 is                                -- 1
      Int_2_2: Integer;                                  -- +2
      begin                                              -- 2
        Int_1_1 := 11; Int_2_2 := 22;                  -- 2
        Proc_1_1; Proc_2_1; Proc_2_2;                  -- 2
      end Proc_2_2;                                     -- -2
  begin                                              -- 1
    Int_1_1 := 11;                                       -- 1
    Proc_1_1; Proc_2_1; Proc_2_2;                      -- 1
  end Proc_1_1;                                       ---1

```

*Note:* The comment after each line indicates the lexical level of the line.

A procedure or function, though introducing a new lexical level, is a declaration of a name at the current lexical level. The parameters of the procedure or function are of course at the next lexical level.

The layout of the above program can be schematically visualized diagrammatically as Figure 24.1:

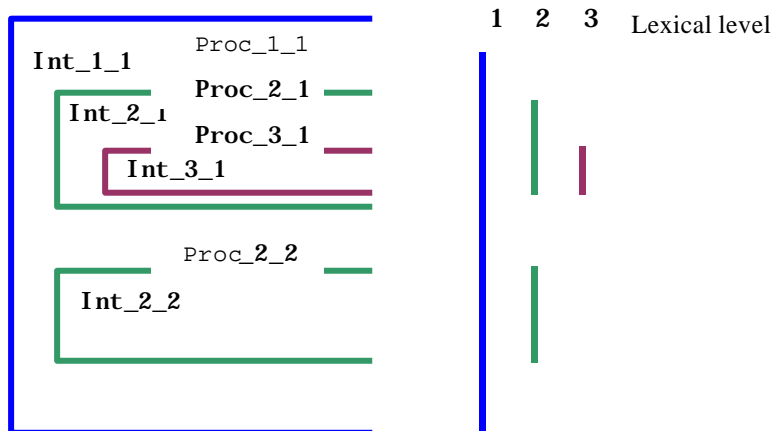


Figure 24.1 Lexical levels in an Ada program.

The procedure Proc\_3\_1 can access the following items:

Procedure	Can access procedures	Can access variables
Proc_3_1	Proc_1_1 Proc_2_1 Proc_3_1	Int_1_1 Int_2_1 Int_3_1

However, the procedure Proc\_2\_2 cannot be accessed as this is declared after the end of Proc\_3\_1. The variable Int\_2\_2 cannot be accessed as even though it is at a lower lexical level it is not in a surrounding lexical level.

The procedure Proc\_2\_2 can access the following items:

Procedure	Can access procedures	Can access variables
-----------	-----------------------	----------------------

Proc_2_2	Proc_1_1	Int_1_1
	Proc_2_1	Int_2_2
	Proc_2_2	

However, the variable `Int_2_1` cannot be accessed as, even though it is at the same lexical level, it is not in a surrounding lexical level.

### 24.1.1 Advantages of using nested procedures

Using a nested procedure structure allows a user to hide names used in the solution of different parts of a problem. This helps reduce the pollution of the global name space with items that only have a very limited scope.

### 24.1.2 Introducing a new lexical level in a procedure or function

The construct **declare ... begin ... end;** introduces a new lexical level in a procedure or function. For example, in the following program:

```

procedure Outer is           --+1
  Outer_Int : Integer;        -- 1
  procedure Inner is         -- 1
    Inner_Int : Integer;      -- +2
  begin                       -- 2
    declare                   -- 2
      I : Integer;            -- +3
    begin                     -- 3
      I := 2;                 -- 3
    end;                     -- -3
    Inner_Int := 1;           -- 2
    Outer_Int := 2;           -- 2
  end Inner;                  -- -2
begin                         -- 1
  Inner;                      -- 1
end Outer;                   ---1

```

the integer object `I` is at lexical level 3 and can only be accessed within the **begin ... end;** of the block defined by the **declare** statement.

### 24.1.3 Holes in visibility

Because names can be overloaded, a hole in the visibility of an item can be created. For example, in the following program the variable `i` declared immediately in the procedure `main` is not visible for the extent of the enclosing **begin end** in the `declare` block.

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use   Ada.Text_IO, Ada.Integer_Text_IO;
procedure Ex5 is
  I : Integer := 1;  --First I declaration
begin
  Put( I );          --Accesses first I => 1
  declare
    I : Integer := 2; --Second I declaration
  begin
    Put( I );        --Accesses second I => 2
  end;
  Put( I );          --Accesses first I => 1
end Ex5;

```

### 24.1.4 Consequences of lexical levels

## 360 *Scope of declared items*

There is a performance penalty to pay for this flexibility in accessing items at different lexical levels. This penalty is mostly evident in the lengthy code required to perform procedure entry and exit. Optimizing compilers can drastically simplify the code required for entry and exit to a procedure or function when only a simple nesting structure is used.

The main code complexity arises because procedures and functions may be called recursively. A recursive procedure or function will create each time it is called a new stack frame.

The use of the class construct can drastically reduce the need to use heavily nested procedures.

### 24.2 Self-assessment

- How might a whole in the visibility of a variable in a program be created.
- Why cannot a procedure or function be called which is in an inner block to the current calling position.
- What procedures and integer variables can code in the body of the procedures Main, Proc\_A, Proc\_B, and Proc\_C access in the following program.

```
procedure Main is
  A : Integer;
  procedure Proc_A is
    B : Integer;
    procedure Proc_B is
      C : Integer;
      begin
        --Code;
      end Proc_B;
    procedure Proc_C is
      D : Integer;
      begin
        --Code;
      end Proc_C;
    begin
      --Code;
    end Proc_A;
    E : Integer;
  begin
    --Code
  end Main;
```

# 25 Mixed language programming

This chapter describes how code written in another language can be called from an Ada program. This allows the Ada programmer to take advantage of the wealth of code previously written in other languages.

## 25.1 Linking to other code

The designers of Ada 95 realised that if the language was to prosper then it must co-exist in a world where code was written in languages other than Ada. Ada provides mechanisms that allow code written in the programming languages C, Fortran and COBOL to be directly called. Other languages such as C++ may also be called using the C interface.

## 25.2 Selected types and functions from Interfaces.C

The following are a selection of types and functions from the package Interfaces.C. This Ada package allows the calling of functions written in the C language. In particular mechanisms are provided to convert between instance of Ada types and instances of C types.

### 25.2.1 Integer, character and floating point types

By using the following types an Ada variable or expression can be converted to a form compatible with the C language.

Integer Types	Char types	Floating point types
Int Short Long Size_T	Char Wchar_T	C_Float Double Long_Double.

For example to pass the integer value `Item` as a parameter to a C function that expects a long double the following expression can be used: `Long_Double(Item)`.

### 25.2.2 C String type

The following declaration:

```
type Char_Array is Array (Size_T range <>) of aliased Char;
```

is used to declare a C array of characters. In C an array of characters terminated by the null character is used to represent a string. For example, the following will declare a string `Name` containing the text "Miranda" that may be passed to a C function that requires a C string parameter.

```
Name : constant Char_Array := "Miranda" & nul;
```

*Note: The use of `nul` to represent the null character. Remember **null** is an Ada reserved word.*

### 25.2.3 Selected functions

Character conversions:

<code>function To_C (Item : in Character) return Char;</code>
<code>function To_Ada (Item : in Char ) return Character;</code>

String conversions:

<code>function To_C ( Item : in String; Append_Nul:in Boolean := True ) return char_array;</code>
<code>function To_Ada ( Item: in char_array; Trim_Nul : in Boolean := True ) return String;</code>

## 25.3 An Ada program calling a C function

The following Ada program calls the C function `triple`. The C function `triple` returns as an integer value triple the integer value passed to it. Firstly an interface function `Triple` is constructed that calls the C function `triple`.

```
with Interfaces.C;
use Interfaces.C;
function Triple( N:in Integer ) return Integer is
  function C_Triple(N:in Int) return Int;
  pragma Import (C, C_Triple, "c_triple");
begin
  return Integer( C_Triple( Int(N) ) );
end Triple;
```

*Note:*     *The use of the type defined in Interfaces.C is:*  
           *Int*                 *Represents a C int*  
           *Integer*            *The Ada integer type.*

In the Ada interface function the **pragma import** is used to request the compiler to link to an externally written procedure or function. In this case the function `c_triple` written in C.

This interface function `triple` is then called from a simple test program.

```
with Ada.Text_IO, Ada.Integer_Text_IO, Triple;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
begin
  Put("3 Tripled is "); Put( Triple(3) ); New_Line;
end Main;
```

The implementation of the function `c_triple` in C is:

```
int c_triple( int n )                 /* function to triple argument */
{
  return n + n + n;
}
```

The above code when compiled and linked will produce the following output.



```
3 Tripled is          9
```

### 25.3.1 Another example

The C library function `strlen` returns the number of characters in a C string. This function is easily called using the Ada function `To_C` which will convert an Ada string into a C string. For example, the following test program calls the C function `strlen` to deliver the number of characters in the Ada string "Brighton".

```
with Interfaces.C, Ada.Text_IO;
use Interfaces.C, Ada.Text_IO;
procedure Main is

  function Strlen( Str:in String ) return Integer is
    function C_Strlen( C_Str:in char_array ) return Int;
    pragma Import (C, C_Strlen, "strlen");
  begin
    return Integer( C_Strlen( To_C( Str, Append_Nul => True ) ) );
  end Strlen;

  place : constant String := "Brighton";
begin
  Put("The length of the string [" & place & "] is " &
    Integer'Image( strlen( place ) ) & " characters long");
  New_Line;
end Main;
```

*Note: The need to specify that a null character is appended to the string before the call to the C `strlen` function.*

The above test program when compiled and run will produce the following results:

```
The length of the string [Brighton] is 8 characters long
```

## 25.4 An Ada package in C

In Section 23.2.2, in the implementation of the class `TUI`, the package `Raw_IO` was used as the interface between the end user and the program. This package is responsible for the raw I/O between the user and the program. It is special in that information cannot be buffered and data must be made immediately available to a program without being echoed back onto the terminal.

This package can be written in part in Ada, but the function `Get_Immediate` that does not echo its character has to be written in C.

Method	Responsibility
<code>Get_Immediate</code>	Read a character from the keyboard, but do not echo it onto the screen.
<code>Put</code>	Write a character or String to the screen with no buffering. Thus the user will see the written text immediately.

The Ada specification of this package is:

## 364 Mixed language programming

```
package Raw_Io is
  procedure Get_Immediate( Ch:out Character );
  procedure Put( Ch:in Character );
  procedure Put( Str:in String );
end Raw_Io;
```

and its implementation is as follows:

```
with Interfaces.C, Ada.Text_Io;
use Interfaces.C, Ada.Text_Io;
package body Raw_Io is

  First_Time : Boolean := True;

  procedure Get_Immediate( Ch:out Character) is
    procedure C_No_Echo;
    pragma Import (C, C_No_Echo, "c_no_echo");    --Turn off echo
  begin
    if First_Time then
      C_No_Echo; First_Time := False;
    end if;
    Ada.Text_Io.Get_Immediate(Ch);
    if Character'Pos(Ch) = 10 then                --Real Return ch
      Ch := Character'Val(13);
    end if;
  end Get_Immediate;

  procedure Put( Ch:in Character ) is             --Raw write
  begin
    Ada.Text_Io.Put( Ch ); Ada.Text_Io.Flush;
  end Put;

  procedure Put( Str:in String ) is               --Raw write
  begin
    Ada.Text_Io.Put( Str ); Ada.Text_Io.Flush;
  end Put;
end Raw_Io;
```

The function `c_no_echo` is a C function that turns off the echoing of characters input to the user program. The implementation in C of this function is as follows:

```

/*
 * Set the terminal mode to -echo -icanon
 * Terminal mode reset when the character ^E is received
 */

#include <termios.h>
#include <unistd.h>

static tcflag_t c_lflag;
static int fd = 1;                               /* STDOUT_FILENO; */
static struct termios termios_data;

void c_no_echo()
{
    tcgetattr( fd, &termios_data );
    c_lflag = termios_data.c_lflag;
    termios_data.c_lflag = termios_data.c_lflag & ( ~(ECHO|ICANON|ECHOCTL) );
    tcsetattr( fd, TCSANOW, &termios_data );
}

```

The function `c_get_char` is a C function that returns the next character input by the user. When the character ^E is received subsequent characters input will be echoed. The implementation in C of this function is as follows:

```

char c_get_char()
{
    char c;

    c = getchar();
    if ( c == '\005' )
    {
        termios_data.c_lflag = c_lflag;
        tcsetattr( fd, TCSANOW, &termios_data );
    }
    return (char) (c & 0xFF);           /* Ordinary character */
}

```

*Note: The use of control-e to return the stream back to its normal state.*

## 25.5 Linking to FORTRAN and COBOL code

By using the package `Interface.Fortran`, subprograms written in FORTRAN may be called. By using the package `Interface.COBOL` subprograms written in COBOL may be called.

# Appendix A: The main language features of Ada 95

## Simple object declarations

```
ch : Character;    -- An 8 bit character
i  : Integer;     -- A whole number
f  : Float;       -- A floating point number
```

## Array declaration

```
Computers_In_Room : array ( 1 .. 10 ) of Natural;
```

## Type and subtype declarations

```
type Money is delta 0.01 digits 8;    --
subtype Pmoney is Money range 0.0 .. Money'Last; --+ve Money
```

## Enumeration declaration

```
type Colour is (Red, Green, Blue);
```

## Simple statements

```
Sum := 2 + 3;
Deposit( Mine, 100.00 );
```

## Block

```
declare
  Ch : Character;
begin
  Ch := 'A'; Put( Ch );
end;
```

## Selection statements

```

if Temp < 15 then Put("Cool"); end if;

if Temp < 15 then Put("Cool"); else Put("Warm"); end if;

case Number is
  when 2+3    => Put("Is 5");
  when 7      => Put("Is 7");
  when others => Put("Not 5 or 7");
end case;

```

## Looping statements

```

while Raining loop      --While raining
  Work;                 -- Perform work
end loop;               --

loop                    --Repeat
  Work;                 -- Perform work
  exit when Sunny;      -- exit from loop when sunny
end loop;               --

for I in 1 .. 10 loop   --Vary I from 1 to 10
  Put(I); New_Line;     -- Write I
end loop;               --

```

## Arithmetic operators

```

Res := A + B;    --plus
Res := A - B;    --minus
Res := A * B;    --multiplication
Res := A / B;    --Division
Res := A mod B;  --Modulus
Res := A rem B;  --Remainder

```

## Conditional expressions

```

if A = B then --Equal to
if A > B then --Greater than
if A < B then --Less than
if A /= B then --Not equal
if A >= B then --Greater
               -- or equal
if A <= B then --Less
               -- or equal
if A in 1 .. 10 then

```

```

if Wet and Jan then -- and
if Dry and Feb then -- or

if Wet and Jan and then --
if Dry and Feb and then --

```

```

if Temp > 15 and Dry then

```

Note: When using **and** **then** or **or** **else** the conditional expression will only be evaluated as far as necessary to produce the result of the condition. Thus in the **if** statement:

```

if fun_one or else fun_two then
fun_two will not be called if fun_one delivered true.

```

## Exits from loops

The following code will execute until the condition sunny is met.

```
loop                --Repeat
  Work;             -- Perform work
  exit when Sunny;  -- exit from loop when sunny
end loop;           --
```

## Class declaration and implementation

```
package Class_Account is
  type Account is tagged private;

  type Money is delta 0.01 digits 8;          --
  subtype Pmoney is Money range 0.0 .. Money'Last; --+ve Money

  procedure Deposit( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account;
                      Amount:in Pmoney; Get:out Pmoney );
  function Balance( The:in Account ) return Money;
private
  type Account is tagged record              --Instance variables
    Balance_Of      : Money := 0.00;        --Amount on deposit
    Min_Balance     : Money := 0.00;        --Minimum Balance
  end record;
end Class_Account;
```

```
package body Class_Account is

  procedure Deposit( The:in out Account; Amount:in Pmoney ) is
  begin
    The.Balance_Of := The.Balance_Of + Amount;
  end Deposit;

  -- Procedures withdraw and balance

end Class_Account;
```

## Inheritance

```
with Class_Account;
use Class_Account;
package Class_Interest_Account is

  type Interest_Account is new Account with private;

  type Imoney is digits Money'digits+2;          --

  procedure End_Of_Day( The:in out Interest_Account );
  procedure Interest_Credit( The:in out Interest_Account );
  procedure Interest_Accumulate( The:in out Interest_Account;
                                Amount: in Imoney );
private
  Daily_Interest_Rate: constant Imoney := 0.000133680617;
  type Interest_Account is new Account with record
    Accumulated_Interest : Imoney := 0.00;        --To date
  end record;
  The_Interest_Rate      : Imoney := Daily_Interest_Rate;
end Class_Interest_Account;
```

```
package body Class_Interest_Account is
  procedure Interest_Credit( The:in out Interest_Account ) is
  begin
    Deposit( The, Money(The.Accumulated_Interest) ); --Rounds
    The.Accumulated_Interest := 0.00;
  end Interest_Credit;

  -- Procedure calc_interest

end Class_Interest_Account;
```

## Program delay

delay n.m seconds	delay until a_time;
<pre>delay n.m;</pre>	<pre>declare   use Ada.Calendar; begin   delay until time_of(2000,1,1,0.0);   -- 24 Hours end;</pre>

## Task

```
task type Task_Factorial is
  entry Start( F:in Positive );
  entry Finish( R:out Positive );
end Task_Factorial;
```

--Specification  
--Rendezvous  
--Rendezvous

```
task body Task_Factorial is
  Fact : Positive; Answer : Positive := 1;
begin
  accept Start( F:in Positive) do Fact := F; end Start;
  for I in 2 .. Fact loop Answer := Answer * I; end loop;
  accept Finish( R:out Positive ) do R:=Answer; end Finish;
end Task_Factorial;
```

--Implementation

## Communication with a task

```
procedure Main is
  Factorial : Task_Factorial;
  Res : Natural;
begin
  Factorial.Start(5); Put( "Factorial 5 is ");
  Factorial.Finish(Res); Put( Res, Width=>4 ); New_Line;
end Main;
```

## Rendezvous

select statement	select with else	select with delay
------------------	------------------	-------------------

<pre> select   accept option1 do     ...   end; or   accept option2 do     ...   end; end select; </pre>	<pre> select   accept ... else   statements; end select; </pre>	<pre> select   accept ... or   delay n.m;   statements; end select; </pre>
--	---	--

## Protected type

```

protected type PT_Buffer is           --Task type specification
  entry Put( Ch:in Character );
  entry Get( Ch:out Character );
private
-- variables which cannot be simultaneous accessed
end PT_Buffer;

```

```

protected body PT_Buffer is

  entry Put( Ch:in Character )
    when No_In_Queue < Queue_Size is
  begin
    . . .
  end Put;

  entry Get( Ch:in out Character )
    when No_In_Queue > 0 is
  begin
    . . .
  end Get;

```



# Appendix B: Components of Ada

## B.1 Reserved words and operators in Ada 95

### B.1.1 Reserved words

abort	abs	abstract	accept	access	aliased
all	and	array	at	begin	body
case	constant	declare	delay	delta	digits
do	else	elsif	end	entry	exception
exit	for	function	generic	goto	if
in	is	limited	loop	mod	new
not	null	of	or	others	out
package	pragma	private	procedure	protected	raise
range	record	rem	renames	requeue	return
reverse	select	separate	subtype	tagged	task
terminate	then	type	until	use	when
while	with	xor			

### B.1.2 Operators

<code>:=</code>	<code>=</code>	<code>/=</code>	<code>&gt;</code>	<code>&lt;</code>	<code>&gt;=</code>
<code>&lt;=</code>	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>rem</code>
<code>mod</code>	<code>**</code>	<code>not</code>	<code>abs</code>	<code>&amp;</code>	<code>and</code>
<code>or</code>	<code>and then</code>	<code>or else</code>	<code>in</code>	<code>not in</code>	<code>xor</code>

*Note:* Some of the operators are represented by reserved words.

## B.2 Attributes of objects and types

### B.2.1 Scalar objects

Attribute		Description	Type of result
S'Max	1	Delivers the max of the two arguments.	S'Base
S'Min	2	Delivers the min of the two arguments.	S'Base

- 1 S'Max denotes a function with specification:  
**function** S'Max( left, right: S'Base) **return** S'Base;  
This is for all scalar types S.
- 2 S'Min may be used in a similar way to S'Max

### B.2.2 Array objects and types

Attribute		Description	Type of result
O'First	1	Delivers the lower bound of the first array index	Type of array index

O'First(n)	1	Delivers the lower bound of the n'th array index	Type of array index
O'Last	1	Delivers the upper bound of the first array index	Type of array index
O'Last(n)	1	Delivers the upper bound of the n'th array index	Type of array index
O'Length	1	Delivers the number of elements in the first array index	Universal Integer
O'Length(n)	1	Delivers the number of elements in the n'th array index	Universal Integer
O'Range	1	Delivers the first array index range	O'First .. O'Last
O'Range(n)	1	Delivers the n'th array index range	O'First(n) .. O'Last(n)

1 Only an instance of an unconstrained array may be interrogated using the attribute.

### B.2.3 Scalar objects and types

Attribute		Description	Type of result
O'First		Delivers the lower bound of the object or type	Of the type of O
O'Last		Delivers the upper bound of the object or type	Of the type of O

### B.2.4 Discrete objects

Attribute		Description	Type of result
O'Succ( val )	1	Delivers the successor of val which is a value in the base type of T. The exception <code>Constraint_error</code> is raised if the successor of O'Last is taken.	Of the base type of T
O'Pred( val )	1	Delivers the predecessor of val which is a value in the base type of T. The exception <code>Constraint_error</code> is raised if the predecessor of O'First is taken.	Of the base type of T

1 The attribute will only work on an instance of discrete object and not on a discrete type.

### B.2.5 Task objects and types

Attribute		Description	Type of result
O'Callable	1	Returns TRUE if the task object is callable	Boolean
O'Storage_size		The storage units required for each activation of the task	Universal Integer
O'Terminated	1	Returns TRUE if the task is terminated	Boolean

1 The attribute will only work on an instance of task object and not on a task type.

### B.2.6 Floating point objects and types

Attribute	Description	Type of result
-----------	-------------	----------------

T'Digits	The decimal precision.	Universal Integer
T'Model_epsilon	The absolute value of the difference between 1.0 and the next representable number above 1.0.	Universal real
T'Safe_first	The lower bound of the safe range of T,	Of type T
T'Safe_last	The upper bound of the safe range of T,	Of type T

## B.3 Literals in Ada

An integer can be expressed in any base from 2 to 16 by prefixing the number by its base. For example, the number 42 to base 10 can be written as:

2#101010#      4#222#      8#52#      10#42#      16#2A#

*Note:*      The use of #'s to bracket the number.

In a number the underscore character can be used to improve readability. Usually this will be used to separate a number into groups of three digits.

1\_00              1\_234.567\_    3.141\_596      1\_000\_000  
8

The number 12.34 can be written as:

0.123\_4E2      1.234E1

a numeric literal is of the type `universal_integer` or `universal_real`, which allows the literal to be used freely with any appropriate type.

## B.4 Operators in Ada 95

Operator	Operand(s)	Result
<b>and</b> <b>or</b> <b>xor</b>	Boolean 1D Boolean array modular	Boolean 1D Boolean array modular
<b>and then</b> <b>or else</b>	Boolean	Boolean
< <= > >=	scalar 1D discrete array	Boolean Boolean
= /=	any non limited operands	Boolean
<b>in</b> <b>not in</b>	scalar <b>in</b> range scalar <b>not in</b> range	Boolean
&	1D array & 1D array 1D array & element element & 1D array element & element	1D array 1D array 1D array 1D array

+ - (monadic)	numeric	Same as operands
+ - (dyadic)	numeric	Same as operands
*	integer * integer floating * floating fixed * integer integer * fixed universal fixed * universal fixed root real * root integer root integer * root real	integer floating fixed fixed universal fixed root real root real

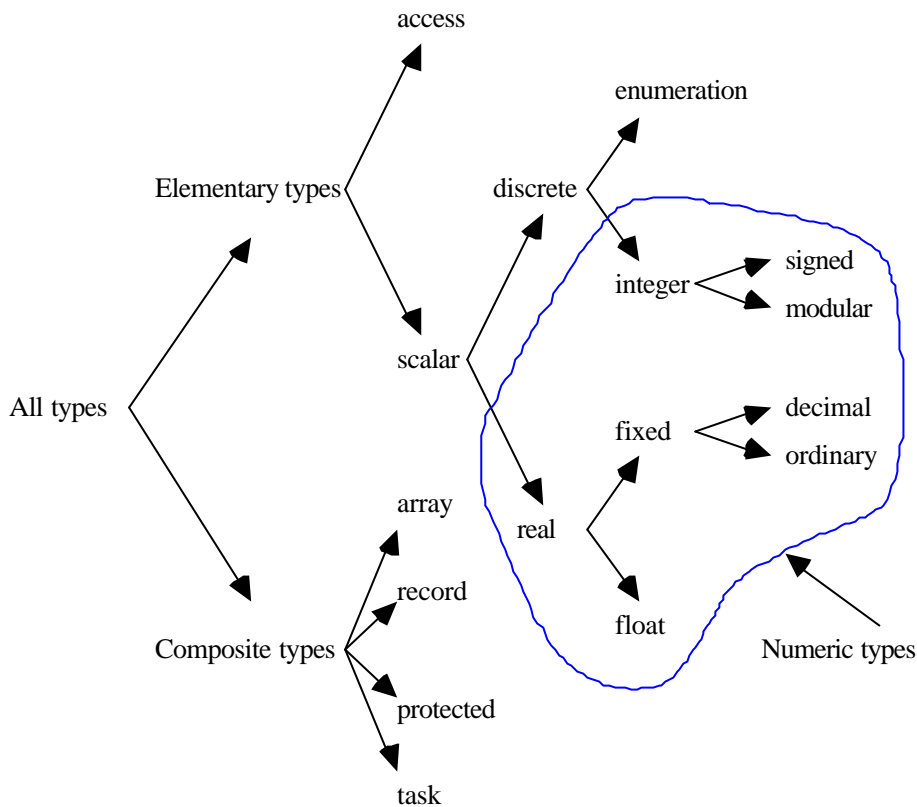
/	integer / integer floating / floating fixed / integer universal fixed / universal fixed root real / root integer	integer floating fixed universal fixed root real
<b>mod rem</b>	integer	integer
**	integer ** integer (>= 0 ) floating ** integer (>= 0 )	integer floating
<b>not</b>	Boolean 1D Boolean array modular	Boolean 1D Boolean array modular
<b>abs</b>	numeric	numeric

*Note:* In the table 1D is a shorthand for a one dimensional array.

#### B.4.1 Priority of operators from high to low

<b>and or xor and then or else</b>	Logical operators	High
= /= < <= > >=	Relational operators	
+ - * / &	Dyadic arithmetic join operator	
+ -	Monadic arithmetic operators	
* / <b>mod rem</b>	Dyadic arithmetic operators	
** <b>abs not</b>	The others	Low

#### B.5 Ada type hierarchy



## B.6 Implementation requirements of standard types

Type	Min value	Max value	Notes
Integer	-32768	32767	
Long_Integer	-2147483648	2147483647	1
Float	6 dec. places		
Long_float	11 dec. places		2

Min Value The minimal value that must be representable (smaller values are allowed).

Max Value The maximum value that must be representable (larger numbers are allowed).

*Note: 1 Should be provided by an implementation if the target machine supports 32bit or longer arithmetic.*

*Note: 2 May be provided.*

## B.7 Exceptions

### B.7.1 Pre-defined exceptions

Exception	Explanation
Constraint_Error	Raised when an attempt is made to assign a value to a variable which does not satisfy a constraint defined on the variable.
Storage_Error	Raised when a program tries to allocate more memory than is available to it.
Program_Error	Raised when an attempt is made to execute an erroneous action. For example, leaving a function without executing a return statement.

Tasking_Error	Raised when an error occurs in a task.
---------------	--

### B.7.2 I/O exceptions

Exception	Explanation
Data_Error	Raised when a get on a numeric object finds the input is not a valid value of this numeric type.
End_Error	Raised when an attempt is made to read past the end of the file.
Mode_Error	Raised when an inappropriate operation is attempted on a file.
Name_Error	Raised if the name used in an open call does not match a file in the external environment.
Status_Error	Raised when an operation is attempted on a file that has not been opened or created.
Use_Error	Raised if the attempted operation is not possible on the external file.

## B.8 Ada 95, the structure

The Ada 95 programming language is split into two distinct sections: a core language that must be implemented and a series of annexes that may or may not be implemented. The annexes extend the language into problem specific areas. The annexes to the language are:

Annex	Name	Contents of annex:
C	System programming	The provision of features that will allow the interfacing of an Ada program to external environments. For example, the interfacing of Ada code to a peripheral device or to components of the operating system interface.
D	Real time systems	The provision of features that will allow the control of real-time processes.
E	Distributed systems	The provision of features that will allow a system to extend beyond a single address space in a single machine.
F	Information systems	The provision of features that will allow an Ada program to communicate with programs or systems written in C or COBOL.
G	Numerics	The provision of features that will allow the construction of numerically intense applications
H	Safety and security	Restrictions to the language to minimize insecurities and areas in which compromises to validation and verification would be made.

*Note: The annexes rarely extend the syntax of the language; rather they provide extra packages to enable the particular area to be performed.*

## B.9 Sources of information

### B.8.1 Copies of the Ada 95 compiler

The main internet site for copies of the GNAT Ada 95 compiler is `cs.nyu.edu`. The latest version of the compiler for a multitude of machines is held in the directory `pub/gnat`.

### B.8.2 Ada information on the World Wide Web

Some of the sites offering information about Ada on the World Wide Web are:

URL (Uniform Resource Locator)	Commentary
<a href="http://lglwww.epfl.ch/Ada/">http://lglwww.epfl.ch/Ada/</a>	A wealth of information about Ada. Has links to other sites.
<a href="http://sw-eng.falls-church.va.us/">http://sw-eng.falls-church.va.us/</a>	The Ada Information Clearinghouse. Many Ada documents, including the reference manual and rational.
<a href="http://wuarchive.wustl.edu/languages/ada/">http://wuarchive.wustl.edu/languages/ada/</a>	The PAL (Public Ada Library): lots of Ada-related software.
<a href="http://www.acm.org/sigada/">http://www.acm.org/sigada/</a>	The ACM SIGAda home page

*Note: The URL should be typed all on one line:*

### B.8.3 News groups

The usenet newsgroup `comp.lang.ada` contains a lively discussion about Ada related topics.

### B.8.4 CD ROMs

Walnut Creek produce a CD ROM of Ada-related information including the GNAT compiler. For more information e-mail [info@cdrom.com](mailto:info@cdrom.com). Alternatively see the WWW page <http://www.cdrom.com/>.

### B.8.5 Additional information on this book

The WWW page <http://www.brighton.ac.uk/ada95/home.html> contains additional information and programs not in this book.

# Appendix C: Library functions and packages

The list of library functions and packages is reproduced from the *Ada 95 Reference Manual* ANSI/ISO/IEC-8652:1995. The following copyright notice appears in the manual:

*Copyright © 1992,1993,1994,1995 Intermetrics, Inc.*

*This copyright is assigned to the U.S. Government. All rights reserved.*

*This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Compiled copies of standard library units and examples need not contain this copyright notice so long as the notice is included in all copies of source code and documentation.*

In reproducing the subset of the library functions and packages in the Ada library, the author has made changes to layout, case, and font. This was accomplished by using various software tools and so minor changes may have been introduced without the author's knowledge.

## C.1 Generic function `Unchecked_Conversion`

The generic function `Unchecked_Conversion` performs a conversion between two dissimilar types. The size of the storage occupied by an instance of the types must be the same.

```
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);
```

## C.2 Generic function `Unchecked_Deallocation`

The generic function `Unchecked_Deallocation` releases storage for storage claimed by an allocator back to the pool of free storage.

```
generic
  type Object(<>) is limited private;
  type Name is access Object;
procedure Ada.Unchecked_Deallocation(X : in out Name);
pragma Convention(Intrinsic, Ada.Unchecked_Deallocation);
pragma Preelaborate(Ada.Unchecked_Deallocation);
```

## C.4 The Package Standard



```

package Standard is
  pragma Pure(Standard);

  type Boolean is (False, True);

  --The predefined relational operators for this type are as follows:

  --function "=" (Left, Right : Boolean) return Boolean;
  --function "/=" (Left, Right : Boolean) return Boolean;
  --function "<" (Left, Right : Boolean) return Boolean;
  --function "<=" (Left, Right : Boolean) return Boolean;
  --function ">" (Left, Right : Boolean) return Boolean;
  --function ">=" (Left, Right : Boolean) return Boolean;

  --The predefined logical operators and the predefined logical
  --negation operator are as follows:

  --function "and" (Left, Right : Boolean) return Boolean;
  --function "or" (Left, Right : Boolean) return Boolean;
  --function "xor" (Left, Right : Boolean) return Boolean;

  --function "not" (Right : Boolean) return Boolean;

  --The integer type root_integer is predefined.
  --The corresponding universal type is universal_integer.
end package Standard;

type Integer is range implementation-defined;

subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;

--The predefined operators for type Integer are as follows:
--function "=" (Left, Right : Integer'Base) return Boolean;
--function "/=" (Left, Right : Integer'Base) return Boolean;
--function "<" (Left, Right : Integer'Base) return Boolean;
--function "<=" (Left, Right : Integer'Base) return Boolean;
--function ">" (Left, Right : Integer'Base) return Boolean;
--function ">=" (Left, Right : Integer'Base) return Boolean;

--function "+" (Right : Integer'Base) return Integer'Base;
--function "-" (Right : Integer'Base) return Integer'Base;
--function "abs" (Right : Integer'Base) return Integer'Base;

--function "+" (Left, Right : Integer'Base) return Integer'Base;
--function "-" (Left, Right : Integer'Base) return Integer'Base;
--function "*" (Left, Right : Integer'Base) return Integer'Base;
--function "/" (Left, Right : Integer'Base) return Integer'Base;
--function "rem" (Left, Right : Integer'Base) return Integer'Base;
--function "mod" (Left, Right : Integer'Base) return Integer'Base;

--function "***" (Left : Integer'Base; Right : Natural) return Integer'Base;

--The specification of each operator for the type
--root_integer, or for any additional predefined integer
--type, is obtained by replacing Integer by the name of the type
--in the specification of the corresponding operator of the type
--Integer. The right operand of the exponentiation operator
--remains as subtype Natural.

--The floating point type root_real is predefined.
--The corresponding universal type is universal_real.

```

```

type Float is digits implementation-defined;

--The predefined operators for this type are as follows:

--function "=" (Left, Right : Float) return Boolean;
--function "/=" (Left, Right : Float) return Boolean;
--function "<" (Left, Right : Float) return Boolean;
--function "<=" (Left, Right : Float) return Boolean;
--function ">" (Left, Right : Float) return Boolean;
--function ">=" (Left, Right : Float) return Boolean;

--function "+" (Right : Float) return Float;
--function "-" (Right : Float) return Float;
--function "abs" (Right : Float) return Float;

--function "+" (Left, Right : Float) return Float;
--function "-" (Left, Right : Float) return Float;
--function "*" (Left, Right : Float) return Float;
--function "/" (Left, Right : Float) return Float;

--function "***" (Left : Float; Right : Integer'Base) return Float;

--The specification of each operator for the type root_real, or for
--any additional predefined floating point type, is obtained by
--replacing Float by the name of the type in the specification of the
--corresponding operator of the type Float.

--In addition, the following operators are predefined for the root
--numeric types:

```

```

function "*" (Left : root_integer; Right : root_real)
  return root_real;

function "*" (Left : root_real; Right : root_integer)
  return root_real;

function "/" (Left : root_real; Right : root_integer)
  return root_real;

--The type universal_fixed is predefined.
--The only multiplying operators defined between
--fixed point types are

function "*" (Left : universal_fixed; Right : universal_fixed)
  return universal_fixed;

function "/" (Left : universal_fixed; Right : universal_fixed)
  return universal_fixed;

--The declaration of type Character is based on the standard ISO 8859-1 character set.

--There are no character literals corresponding to the positions for control characters.
--They are indicated in italics in this definition. See 3.5.2.

```

```

type Character is

(nul, soh, stx, etx, eot, enq, ack, bel,
 bs, ht, lf, vt, ff, cr, so, si,

 dle, dc1, dc2, dc3, dc4, nak, syn, etb,
 can, em, sub, esc, fs, gs, rs, us,

 ' ', '!', '"', '#', '$', '%', '&', '\'',
 '(', ')', '*', '+', ',', '-', '.', '/',

 '0', '1', '2', '3', '4', '5', '6', '7',
 '8', '9', ':', ';', '<', '=', '>', '?',

 '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',

 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
 'X', 'Y', 'Z', '[', '_ '], '^', '_',

 '\', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',

 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
 'x', 'y', 'z', '|', '~', del,

 reserved_128, reserved_129, bph, nbh,
 reserved_132, nel, ssa, esa,

 hts, htj, vts, pld, plu, ri, ss2, ss3,

 dcs, pul, pu2, sts, cch, mw, spa, epa,

 sos, reserved_153, sci, csi,
 st, osc, pm, apc,

... );

--The predefined operators for the type Character are the same as for
--any enumeration type.

--The declaration of type Wide_Character is based on the standard ISO 10646 BMP character set.
--The first 256 positions have the same contents as type Character. See 3.5.2.

type Wide_Character is (nul, soh ... FFFE, FFFF);

package ASCII is ... end ASCII; --Obsolescent; see J.5

--Predefined string types:

type String is array(Positive range <>) of Character;
pragma Pack(String);

--The predefined operators for this type are as follows:

-- function "=" (Left, Right: String) return Boolean;
-- function "/=" (Left, Right: String) return Boolean;
-- function "<" (Left, Right: String) return Boolean;
-- function "<=" (Left, Right: String) return Boolean;
-- function ">" (Left, Right: String) return Boolean;
-- function ">=" (Left, Right: String) return Boolean;

```

```

--  function "&" (Left: String;   Right: String)   return String;
--  function "&" (Left: Character; Right: String)   return String;
--  function "&" (Left: String;   Right: Character) return String;
--  function "&" (Left: Character; Right: Character) return String;

type Wide_String is array(Positive range <>) of Wide_Character;
pragma Pack(Wide_String);

--The predefined operators for this type correspond to those for String

type Duration is delta implementation-defined range implementation-defined;

--The predefined operators for the type Duration are the same as for
--any fixed point type.

--The predefined exceptions:

Constraint_Error: exception;
Program_Error   : exception;
Storage_Error   : exception;
Tasking_Error   : exception;

end Standard;

```

## C.5 The Package `Ada.Text_IO`

In the package `Ada.Standard` the following parameter arguments are used:

Parameter name	Purpose
<code>aft</code>	The number of digits after the decimal place
<code>base</code>	The base of the number.
<code>exp</code>	The number of characters in the exponent. For the number 123.45678: exp=>0 would give a format of => 123.45678 exp=>2 would give a format of => 1.2345678E+2 exp=>4 would give a format of => 1.2345678E+002
<code>file</code>	<code>file_type</code> : The file descriptor which is read from or written to. <code>file_access</code> : The access value of the <code>file_type</code> .
<code>fore</code>	The number of digits before the decimal place
<code>form</code>	Form of the created output file.
<code>item</code>	The item to be read / written.
<code>last</code>	The last character read from the string.
<code>mode</code>	The mode of the operation read, write or append.
<code>name</code>	The name of the file as a String.
<code>width</code>	The number of characters to be read / written

```
package Ada.IO_Exceptions is
  pragma Pure(IO_Exceptions);

  Status_Error : exception;
  Mode_Error   : exception;
  Name_Error    : exception;
  Use_Error     : exception;
  Device_Error  : exception;
  End_Error     : exception;
  Data_Error    : exception;
  Layout_Error  : exception;

end Ada.IO_Exceptions;
```

```

with Ada.IO_Exceptions;
package Ada.Text_IO is

  type File_Type is limited private;

  type File_Mode is (In_File, Out_File, Append_File);

  type Count is range 0 .. implementation-defined;
  subtype Positive_Count is Count range 1 .. Count'Last;
  Unbounded : constant Count := 0; --line and page length

  subtype Field is Integer range 0 .. implementation-defined;
  subtype Number_Base is Integer range 2 .. 16;

  type Type_Set is (Lower_Case, Upper_Case);

  --File Management

  procedure Create (File : in out File_Type;
                   Mode : in File_Mode := Out_File;
                   Name : in String := "";
                   Form : in String := "");

  procedure Open (File : in out File_Type;
                 Mode : in File_Mode;
                 Name : in String;
                 Form : in String := "");

  procedure Close (File : in out File_Type);
  procedure Delete (File : in out File_Type);
  procedure Reset (File : in out File_Type; Mode : in File_Mode);
  procedure Reset (File : in out File_Type);

  function Mode (File : in File_Type) return File_Mode;
  function Name (File : in File_Type) return String;
  function Form (File : in File_Type) return String;

  function Is_Open (File : in File_Type) return Boolean;

  --Control of default input and output files

  procedure Set_Input (File : in File_Type);
  procedure Set_Output (File : in File_Type);
  procedure Set_Error (File : in File_Type);

  function Standard_Input return File_Type;
  function Standard_Output return File_Type;
  function Standard_Error return File_Type;

  function Current_Input return File_Type;
  function Current_Output return File_Type;
  function Current_Error return File_Type;

  type File_Access is access constant File_Type;

  function Standard_Input return File_Access;
  function Standard_Output return File_Access;
  function Standard_Error return File_Access;

  function Current_Input return File_Access;
  function Current_Output return File_Access;
  function Current_Error return File_Access;

  --Buffer control
  procedure Flush (File : in out File_Type);
  procedure Flush;

```

```

--Specification of line and page lengths

procedure Set_Line_Length(File : in File_Type; To : in Count);
procedure Set_Line_Length(To : in Count);

procedure Set_Page_Length(File : in File_Type; To : in Count);
procedure Set_Page_Length(To : in Count);

function Line_Length(File : in File_Type) return Count;
function Line_Length return Count;

function Page_Length(File : in File_Type) return Count;
function Page_Length return Count;

--Column, Line, and Page Control

procedure New_Line (File : in File_Type;
                    Spacing : in Positive_Count := 1);
procedure New_Line (Spacing : in Positive_Count := 1);

procedure Skip_Line (File : in File_Type;
                    Spacing : in Positive_Count := 1);
procedure Skip_Line (Spacing : in Positive_Count := 1);

function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;

procedure New_Page (File : in File_Type);
procedure New_Page;

procedure Skip_Page (File : in File_Type);
procedure Skip_Page;

function End_Of_Page(File : in File_Type) return Boolean;
function End_Of_Page return Boolean;

function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;

procedure Set_Col (File : in File_Type; To : in Positive_Count);
procedure Set_Col (To : in Positive_Count);

procedure Set_Line(File : in File_Type; To : in Positive_Count);
procedure Set_Line(To : in Positive_Count);

function Col (File : in File_Type) return Positive_Count;
function Col return Positive_Count;

function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;

function Page(File : in File_Type) return Positive_Count;
function Page return Positive_Count;

--Character Input-Output

procedure Get(File : in File_Type; Item : out Character);
procedure Get(Item : out Character);

procedure Put(File : in File_Type; Item : in Character);
procedure Put(Item : in Character);

procedure Look_Ahead (File : in File_Type;
                    Item : out Character;
                    End_Of_Line : out Boolean);

```

```

procedure Look_Ahead (Item      : out Character;
                      End_Of_Line : out Boolean);

procedure Get_Immediate(File      : in File_Type;
                      Item       : out Character);
procedure Get_Immediate(Item      : out Character);

procedure Get_Immediate(File      : in File_Type;
                      Item       : out Character;
                      Available  : out Boolean);
procedure Get_Immediate(Item      : out Character;
                      Available  : out Boolean);

--String Input-Output

procedure Get(File : in File_Type; Item : out String);
procedure Get(Item : out String);

procedure Put(File : in File_Type; Item : in String);
procedure Put(Item : in String);

procedure Get_Line(File : in File_Type;
                  Item  : out String;
                  Last  : out Natural);
procedure Get_Line(Item : out String; Last : out Natural);

procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);

--Generic packages for Input-Output of Integer Types

generic
  type Num is range <>;
package Integer_IO is

  Default_Width : Field := Num'Width;
  Default_Base  : Number_Base := 10;

  procedure Get(File : in File_Type;
              Item  : out Num;
              Width : in Field := 0);
  procedure Get(Item : out Num;
              Width : in Field := 0);

  procedure Put(File : in File_Type;
              Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);
  procedure Put(Item : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);

  procedure Get(From : in String;
              Item  : out Num;
              Last  : out Positive);
  procedure Put(To   : out String;
              Item  : in Num;
              Base  : in Number_Base := Default_Base);

end Integer_IO;

generic
  type Num is mod <>;
package Modular_IO is

  Default_Width : Field := Num'Width;
  Default_Base  : Number_Base := 10;

```



```

procedure Get(File : in File_Type;
               Item : out Num;
               Width : in Field := 0);
procedure Get(Item : out Num;
               Width : in Field := 0);

procedure Put(File : in File_Type;
               Item : in Num;
               Width : in Field := Default_Width;
               Base : in Number_Base := Default_Base);
procedure Put(Item : in Num;
               Width : in Field := Default_Width;
               Base : in Number_Base := Default_Base);
procedure Get(From : in String;
               Item : out Num;
               Last : out Positive);
procedure Put(To : out String;
               Item : in Num;
               Base : in Number_Base := Default_Base);

end Modular_IO;
--Generic packages for Input-Output of Real Types

generic
  type Num is digits <>;
package Float_IO is

  Default_Fore : Field := 2;
  Default_Aft  : Field := Num'Digits-1;
  Default_Exp  : Field := 3;

  procedure Get(File : in File_Type;
               Item : out Num;
               Width : in Field := 0);
  procedure Get(Item : out Num;
               Width : in Field := 0);

  procedure Put(File : in File_Type;
               Item : in Num;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
  procedure Put(Item : in Num;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);

  procedure Get(From : in String;
               Item : out Num;
               Last : out Positive);
  procedure Put(To : out String;
               Item : in Num;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
end Float_IO;

generic
  type Num is delta <>;
package Fixed_IO is

  Default_Fore : Field := Num'Fore;
  Default_Aft  : Field := Num'Aft;
  Default_Exp  : Field := 0;

  procedure Get(File : in File_Type;
               Item : out Num;

```

```

        Width : in Field := 0);
    procedure Get(Item : out Num;
        Width : in Field := 0);

    procedure Put(File : in File_Type;
        Item : in Num;
        Fore : in Field := Default_Fore;
        Aft : in Field := Default_Aft;
        Exp : in Field := Default_Exp);
    procedure Put(Item : in Num;
        Fore : in Field := Default_Fore;
        Aft : in Field := Default_Aft;
        Exp : in Field := Default_Exp);

    procedure Get(From : in String;
        Item : out Num;
        Last : out Positive);
    procedure Put(To : out String;
        Item : in Num;
        Aft : in Field := Default_Aft;
        Exp : in Field := Default_Exp);
end Fixed_IO;

generic
    type Num is delta <> digits <>;
package Decimal_IO is

    Default_Fore : Field := Num'Fore;
    Default_Aft : Field := Num'Aft;
    Default_Exp : Field := 0;

    procedure Get(File : in File_Type;
        Item : out Num;
        Width : in Field := 0);
    procedure Get(Item : out Num;
        Width : in Field := 0);

    procedure Put(File : in File_Type;
        Item : in Num;
        Fore : in Field := Default_Fore;
        Aft : in Field := Default_Aft;
        Exp : in Field := Default_Exp);
    procedure Put(Item : in Num;
        Fore : in Field := Default_Fore;
        Aft : in Field := Default_Aft;
        Exp : in Field := Default_Exp);

    procedure Get(From : in String;
        Item : out Num;
        Last : out Positive);
    procedure Put(To : out String;
        Item : in Num;
        Aft : in Field := Default_Aft;
        Exp : in Field := Default_Exp);
end Decimal_IO;

--Generic package for Input-Output of Enumeration Types

generic
    type Enum is (<>);
package Enumeration_IO is

    Default_Width : Field := 0;
    Default_Setting : Type_Set := Upper_Case;

    procedure Get(File : in File_Type;
        Item : out Enum);

```

```

procedure Get(Item : out Enum);

procedure Put(File : in File_Type;
              Item : in Enum;
              Width : in Field := Default_Width;
              Set : in Type_Set := Default_Setting);

procedure Put(Item : in Enum;
              Width : in Field := Default_Width;
              Set : in Type_Set := Default_Setting);

procedure Get(From : in String;
              Item : out Enum;
              Last : out Positive);

procedure Put(To : out String;
              Item : in Enum;
              Set : in Type_Set := Default_Setting);

end Enumeration_IO;

--Exceptions

Status_Error : exception renames IO_Exceptions.Status_Error;
Mode_Error : exception renames IO_Exceptions.Mode_Error;
Name_Error : exception renames IO_Exceptions.Name_Error;
Use_Error : exception renames IO_Exceptions.Use_Error;
Device_Error : exception renames IO_Exceptions.Device_Error;
End_Error : exception renames IO_Exceptions.End_Error;
Data_Error : exception renames IO_Exceptions.Data_Error;
Layout_Error : exception renames IO_Exceptions.Layout_Error;

private
... --not specified by the language
end Ada.Text_IO;

```

## C.6 The Package Ada.Sequential\_io

In the package Ada.Sequential\_io the following parameter arguments are used:

Parameter name	Purpose
file	file_type: The file descriptor which is read from or written to.
form	Form of the created output file.
mode	The mode of the operation read, write or append.
name	The name of the file as a String.

```

with Ada.IO_Exceptions;
generic
  type Element_Type(<>) is private;
package Ada.Sequential_IO is

  type File_Type is limited private;

  type File_Mode is (In_File, Out_File, Append_File);

  --File management

  procedure Create(File : in out File_Type;
                  Mode : in File_Mode := Out_File;
                  Name : in String := "";
                  Form : in String := "");

  procedure Open (File : in out File_Type;
                 Mode : in File_Mode;
                 Name : in String;
                 Form : in String := "");

  procedure Close (File : in out File_Type);
  procedure Delete(File : in out File_Type);
  procedure Reset (File : in out File_Type; Mode : in File_Mode);
  procedure Reset (File : in out File_Type);

  function Mode (File : in File_Type) return File_Mode;
  function Name (File : in File_Type) return String;
  function Form (File : in File_Type) return String;

  function Is_Open(File : in File_Type) return Boolean;

  --Input and output operations

  procedure Read (File : in File_Type; Item : out Element_Type);
  procedure Write (File : in File_Type; Item : in Element_Type);

  function End_Of_File(File : in File_Type) return Boolean;

  --Exceptions

  Status_Error : exception renames IO_Exceptions.Status_Error;
  Mode_Error   : exception renames IO_Exceptions.Mode_Error;
  Name_Error   : exception renames IO_Exceptions.Name_Error;
  Use_Error    : exception renames IO_Exceptions.Use_Error;
  Device_Error : exception renames IO_Exceptions.Device_Error;
  End_Error    : exception renames IO_Exceptions.End_Error;
  Data_Error   : exception renames IO_Exceptions.Data_Error;

private
  ... --not specified by the language
end Ada.Sequential_IO;

```

## C.7 The Package `Ada.Characters.Handling`

```

package Ada.Characters.Handling is
  pragma Preelaborate(Handling);

  --Character classification functions

  function Is_Control      (Item : in Character) return Boolean;
  function Is_Graphic      (Item : in Character) return Boolean;
  function Is_Letter       (Item : in Character) return Boolean;
  function Is_Lower        (Item : in Character) return Boolean;
  function Is_Upper        (Item : in Character) return Boolean;
  function Is_Basic        (Item : in Character) return Boolean;
  function Is_Digit        (Item : in Character) return Boolean;
  function Is_Decimal_Digit (Item : in Character) return Boolean
    renames Is_Digit;
  function Is_Hexadecimal_Digit (Item : in Character) return Boolean;
  function Is_Alphanumeric (Item : in Character) return Boolean;
  function Is_Special      (Item : in Character) return Boolean;

  --Conversion functions for Character and String

  function To_Lower (Item : in Character) return Character;
  function To_Upper (Item : in Character) return Character;
  function To_Basic (Item : in Character) return Character;

  function To_Lower (Item : in String) return String;
  function To_Upper (Item : in String) return String;
  function To_Basic (Item : in String) return String;

  --Classifications of and conversions between Character and ISO 646

  subtype ISO_646 is
    Character range Character'Val(0) .. Character'Val(127);

  function Is_ISO_646 (Item : in Character) return Boolean;
  function Is_ISO_646 (Item : in String) return Boolean;

  function To_ISO_646 (Item      : in Character;
    Substitute : in ISO_646 := ' ')
    return ISO_646;

  function To_ISO_646 (Item      : in String;
    Substitute : in ISO_646 := ' ')
    return String;

  --Classifications of and conversions between Wide_Character and Character.

  function Is_Character (Item : in Wide_Character) return Boolean;
  function Is_String    (Item : in Wide_String) return Boolean;

  function To_Character (Item      : in Wide_Character;
    Substitute : in Character := ' ')
    return Character;

  function To_String    (Item      : in Wide_String;
    Substitute : in Character := ' ')
    return String;

  function To_Wide_Character (Item : in Character) return Wide_Character;

  function To_Wide_String    (Item : in String) return Wide_String;

end Ada.Characters.Handling;

```

## C.8 The Package Ada.Strings.Bounded

In the package Ada.Strings.Bounded the following parameter arguments are used:

Parameter name	Purpose
drop	= Left (Compressing to the right) = Right (Compressing to the left) = Error (Strings Length_error propagated)
going	= Forward (Forward search)

```

with Ada.Strings.Maps;
package Ada.Strings.Bounded is
  pragma Preelaborate(Bounded);

  generic
    Max : Positive;    --Maximum length of a Bounded_String
  package Generic_Bounded_Length is

    Max_Length : constant Positive := Max;

    type Bounded_String is private;

    Null_Bounded_String : constant Bounded_String;

    subtype Length_Range is Natural range 0 .. Max_Length;

    function Length (Source : in Bounded_String) return Length_Range;

```

```

--Conversion, Concatenation, and Selection functions

function To_Bounded_String (Source : in String;
                           Drop   : in Truncation := Error)
  return Bounded_String;

function To_String (Source : in Bounded_String) return String;

function Append (Left, Right : in Bounded_String;
                Drop       : in Truncation := Error)
  return Bounded_String;

function Append (Left  : in Bounded_String;
                Right  : in String;
                Drop   : in Truncation := Error)
  return Bounded_String;

function Append (Left  : in String;
                Right  : in Bounded_String;
                Drop   : in Truncation := Error)
  return Bounded_String;

function Append (Left  : in Bounded_String;
                Right  : in Character;
                Drop   : in Truncation := Error)
  return Bounded_String;

function Append (Left  : in Character;
                Right  : in Bounded_String;
                Drop   : in Truncation := Error)
  return Bounded_String;

procedure Append (Source  : in out Bounded_String;
                New_Item : in Bounded_String;
                Drop      : in Truncation := Error);

procedure Append (Source  : in out Bounded_String;
                New_Item : in String;
                Drop      : in Truncation := Error);

procedure Append (Source  : in out Bounded_String;
                New_Item : in Character;
                Drop      : in Truncation := Error);

function "&" (Left, Right : in Bounded_String)
  return Bounded_String;

function "&" (Left : in Bounded_String; Right : in String)
  return Bounded_String;

function "&" (Left : in String; Right : in Bounded_String)
  return Bounded_String;

function "&" (Left : in Bounded_String; Right : in Character)
  return Bounded_String;

function "&" (Left : in Character; Right : in Bounded_String)
  return Bounded_String;

function Element (Source : in Bounded_String;
                Index  : in Positive)
  return Character;

procedure Replace_Element (Source : in out Bounded_String;
                Index  : in Positive;
                By     : in Character);

```

```

function Slice (Source : in Bounded_String;
               Low    : in Positive;
               High   : in Natural)
  return String;

function "=" (Left, Right : in Bounded_String) return Boolean;
function "=" (Left : in Bounded_String; Right : in String)
  return Boolean;

function "<=" (Left : in String; Right : in Bounded_String)
  return Boolean;

function "<" (Left, Right : in Bounded_String) return Boolean;
function "<" (Left : in Bounded_String; Right : in String)
  return Boolean;

function "<" (Left : in String; Right : in Bounded_String)
  return Boolean;

function "<=" (Left, Right : in Bounded_String) return Boolean;
function "<=" (Left : in Bounded_String; Right : in String)
  return Boolean;

function "<=" (Left : in String; Right : in Bounded_String)
  return Boolean;

function ">" (Left, Right : in Bounded_String) return Boolean;
function ">" (Left : in Bounded_String; Right : in String)
  return Boolean;

function ">" (Left : in String; Right : in Bounded_String)
  return Boolean;

function ">=" (Left, Right : in Bounded_String) return Boolean;
function ">=" (Left : in Bounded_String; Right : in String)
  return Boolean;

function ">=" (Left : in String; Right : in Bounded_String)
  return Boolean;

--Search functions

function Index (Source : in Bounded_String;
               Pattern : in String;
               Going   : in Direction := Forward;
               Mapping  : in Maps.Character_Mapping
                       := Maps.Identity)
  return Natural;

function Index (Source : in Bounded_String;
               Pattern : in String;
               Going   : in Direction := Forward;
               Mapping  : in Maps.Character_Mapping_Function)
  return Natural;

function Index (Source : in Bounded_String;
               Set      : in Maps.Character_Set;
               Test     : in Membership := Inside;
               Going    : in Direction := Forward)
  return Natural;

function Index_Non_Blank (Source : in Bounded_String;

```



```

        Going : in Direction := Forward)

    return Natural;

function Count (Source : in Bounded_String;
                Pattern : in String;
                Mapping : in Maps.Character_Mapping
                    := Maps.Identity)

    return Natural;

function Count (Source : in Bounded_String;
                Pattern : in String;
                Mapping : in Maps.Character_Mapping_Function)

    return Natural;

function Count (Source : in Bounded_String;
                Set : in Maps.Character_Set)

    return Natural;

procedure Find-Token (Source : in Bounded_String;
                    Set : in Maps.Character_Set;
                    Test : in Membership;
                    First : out Positive;
                    Last : out Natural);

--String translation subprograms

function Translate (Source : in Bounded_String;
                  Mapping : in Maps.Character_Mapping)

    return Bounded_String;

procedure Translate (Source : in out Bounded_String;
                  Mapping : in Maps.Character_Mapping);

function Translate (Source : in Bounded_String;
                  Mapping : in Maps.Character_Mapping_Function)

    return Bounded_String;

procedure Translate (Source : in out Bounded_String;
                  Mapping : in Maps.Character_Mapping_Function);

--String transformation subprograms

function Replace_Slice (Source : in Bounded_String;
                      Low : in Positive;
                      High : in Natural;
                      By : in String;
                      Drop : in Truncation := Error)

    return Bounded_String;

procedure Replace_Slice (Source : in out Bounded_String;
                      Low : in Positive;
                      High : in Natural;
                      By : in String;
                      Drop : in Truncation := Error);

function Insert (Source : in Bounded_String;
                Before : in Positive;
                New_Item : in String;
                Drop : in Truncation := Error)

    return Bounded_String;

procedure Insert (Source : in out Bounded_String;
                Before : in Positive;
                New_Item : in String;
                Drop : in Truncation := Error);

```

```

function Overwrite (Source : in Bounded_String;
                   Position : in Positive;
                   New_Item : in String;
                   Drop      : in Truncation := Error)
return Bounded_String;

procedure Overwrite (Source : in out Bounded_String;
                   Position : in Positive;
                   New_Item : in String;
                   Drop      : in Truncation := Error);

function Delete (Source : in Bounded_String;
                From     : in Positive;
                Through   : in Natural)
return Bounded_String;

procedure Delete (Source : in out Bounded_String;
                From     : in Positive;
                Through   : in Natural);

--String selector subprograms

function Trim (Source : in Bounded_String;
              Side     : in Trim_End)
return Bounded_String;
procedure Trim (Source : in out Bounded_String;
              Side     : in Trim_End);

function Trim (Source : in Bounded_String;
              Left     : in Maps.Character_Set;
              Right    : in Maps.Character_Set)
return Bounded_String;

procedure Trim (Source : in out Bounded_String;
              Left     : in Maps.Character_Set;
              Right    : in Maps.Character_Set);

function Head (Source : in Bounded_String;
              Count    : in Natural;
              Pad       : in Character := Space;
              Drop      : in Truncation := Error)
return Bounded_String;
procedure Head (Source : in out Bounded_String;
              Count    : in Natural;
              Pad       : in Character := Space;
              Drop      : in Truncation := Error);

function Tail (Source : in Bounded_String;
              Count    : in Natural;
              Pad       : in Character := Space;
              Drop      : in Truncation := Error)
return Bounded_String;

procedure Tail (Source : in out Bounded_String;
              Count    : in Natural;
              Pad       : in Character := Space;
              Drop      : in Truncation := Error);

--String constructor subprograms
function "*" (Left : in Natural;
             Right : in Character)
return Bounded_String;

function "*" (Left : in Natural;
             Right : in String)
return Bounded_String;

function "*" (Left : in Natural;

```

```

        Right : in Bounded_String)
    return Bounded_String;

    function Replicate (Count : in Natural;
                        Item   : in Character;
                        Drop   : in Truncation := Error)
    return Bounded_String;

    function Replicate (Count : in Natural;
                        Item   : in String;
                        Drop   : in Truncation := Error)
    return Bounded_String;

    function Replicate (Count : in Natural;
                        Item   : in Bounded_String;
                        Drop   : in Truncation := Error)
    return Bounded_String;
private
    ... --not specified by the language
end Generic_Bounded_Length;
end Ada.Strings.Bounded;

```

## C.9 The Package Interfaces.C

```

package Interfaces.C is
    pragma Pure(C);

    --Declarations based on C's <limits.h>

    CHAR_BIT  : constant := implementation-defined; --typically 8
    SCHAR_MIN : constant := implementation-defined; --typically -128
    SCHAR_MAX : constant := implementation-defined; --typically 127
    UCHAR_MAX : constant := implementation-defined; --typically 255

    --Signed and Unsigned Integers
    type int      is range implementation-defined;
    type short   is range implementation-defined;
    type long    is range implementation-defined;

    type signed_char is range SCHAR_MIN .. SCHAR_MAX;
    for signed_char'Size use CHAR_BIT;

    type unsigned      is mod implementation-defined;
    type unsigned_short is mod implementation-defined;
    type unsigned_long  is mod implementation-defined;

    type unsigned_char is mod (UCHAR_MAX+1);
    for unsigned_char'Size use CHAR_BIT;

    subtype plain_char is implementation-defined;

    type ptrdiff_t is range implementation-defined;

    type size_t is mod implementation-defined;

```

```

--Floating Point

type C_float      is digits implementation-defined;

type double       is digits implementation-defined;

type long_double  is digits implementation-defined;

--Characters and Strings

type char is <implementation-defined character type>;

nul : constant char := char'First;

function To_C    (Item : in Character) return char;

function To_Ada  (Item : in char) return Character;

type char_array is array (size_t range <>) of aliased char;
pragma Pack(char_array);
for char_array'Component_Size use CHAR_BIT;

function Is_Nul_Terminated (Item : in char_array) return Boolean;

function To_C    (Item      : in String;
                  Append_Nul : in Boolean := True)
  return char_array;

function To_Ada  (Item      : in char_array;
                  Trim_Nul  : in Boolean := True)
  return String;

procedure To_C (Item      : in String;
                Target    : out char_array;
                Count     : out size_t;
                Append_Nul : in Boolean := True);

procedure To_Ada (Item      : in char_array;
                  Target    : out String;
                  Count     : out Natural;
                  Trim_Nul  : in Boolean := True);

--Wide Character and Wide String

type wchar_t is implementation-defined;

wide_nul : constant wchar_t := wchar_t'First;

function To_C    (Item : in Wide_Character) return wchar_t;
function To_Ada  (Item : in wchar_t) return Wide_Character;

type wchar_array is array (size_t range <>) of aliased wchar_t;

pragma Pack(wchar_array);

function Is_Nul_Terminated (Item : in wchar_array) return Boolean;

function To_C    (Item      : in Wide_String;
                  Append_Nul : in Boolean := True)
  return wchar_array;

function To_Ada  (Item      : in wchar_array;
                  Trim_Nul  : in Boolean := True)
  return Wide_String;

procedure To_C (Item      : in Wide_String;
                Target    : out wchar_array;

```

```

        Count      : out size_t;
        Append_Nul : in Boolean := True);

    procedure To_Ada (Item      : in wchar_array;
                     Target    : out Wide_String;
                     Count     : out Natural;
                     Trim_Nul  : in Boolean := True);

    Terminator_Error : exception;

end Interfaces.C;

```

## C.10 The Package Ada.Numerics

```

package Ada.Numerics is
    pragma Pure(Numerics);
    Argument_Error : exception;
    Pi : constant :=
        3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
    e : constant :=
        2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;

```

## C.11 The Package Ada.Numerics.generic\_elementary\_functions

```

generic
    type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
    pragma Pure(Generic_Elementary_Functions);

    function Sqrt      (X          : Float_Type'Base) return Float_Type'Base;
    function Log       (X          : Float_Type'Base) return Float_Type'Base;
    function Log       (X, Base    : Float_Type'Base) return Float_Type'Base;
    function Exp       (X          : Float_Type'Base) return Float_Type'Base;
    function "***"     (Left, Right : Float_Type'Base) return Float_Type'Base;

    function Sin       (X          : Float_Type'Base) return Float_Type'Base;
    function Sin       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
    function Cos       (X          : Float_Type'Base) return Float_Type'Base;
    function Cos       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
    function Tan       (X          : Float_Type'Base) return Float_Type'Base;
    function Tan       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
    function Cot       (X          : Float_Type'Base) return Float_Type'Base;
    function Cot       (X, Cycle   : Float_Type'Base) return Float_Type'Base;

    function Arcsin    (X          : Float_Type'Base) return Float_Type'Base;
    function Arcsin    (X, Cycle   : Float_Type'Base) return Float_Type'Base;
    function Arccos    (X          : Float_Type'Base) return Float_Type'Base;
    function Arccos    (X, Cycle   : Float_Type'Base) return Float_Type'Base;
    function Arctan    (Y          : Float_Type'Base;
                       X          : Float_Type'Base := 1.0) return Float_Type'Base;
    function Arctan    (Y          : Float_Type'Base;
                       X          : Float_Type'Base := 1.0;
                       Cycle      : Float_Type'Base) return Float_Type'Base;

```

```

function Arccot (X      : Float_Type'Base;
                  Y      : Float_Type'Base:= 1.0) return Float_Type'Base;

function Arccot (X      : Float_Type'Base;
                  Y      : Float_Type'Base := 1.0;
                  Cycle   : Float_Type'Base) return Float_Type'Base;

function Sinh (X      : Float_Type'Base) return Float_Type'Base;
function Cosh (X      : Float_Type'Base) return Float_Type'Base;
function Tanh (X      : Float_Type'Base) return Float_Type'Base;
function Coth (X      : Float_Type'Base) return Float_Type'Base;
function Arcsinh (X    : Float_Type'Base) return Float_Type'Base;
function Arccosh (X    : Float_Type'Base) return Float_Type'Base;
function Arctanh (X    : Float_Type'Base) return Float_Type'Base;
function Arccoth (X    : Float_Type'Base) return Float_Type'Base;

end Ada.Numerics.Generic_Elementary_Functions;

```

## C.12 The Package `Ada.Command_line`

```

package Ada.Command_Line is
  pragma Preelaborate(Command_Line);

  function Argument_Count return Natural;

  function Argument (Number : in Positive) return String;

  function Command_Name return String;

  type Exit_Status is implementation-defined integer type;

  Success : constant Exit_Status;
  Failure : constant Exit_Status;

  procedure Set_Exit_Status (Code : in Exit_Status);

private
  ... --not specified by the language
end Ada.Command_Line;

```

## C.13 The Package `Ada.Finalization`

```

package Ada.Finalization is
  pragma Preelaborate(Finalization);

  type Controlled is abstract tagged private;

  procedure Initialize(Object : in out Controlled);
  procedure Adjust      (Object : in out Controlled);
  procedure Finalize    (Object : in out Controlled);

  type Limited_Controlled is abstract tagged limited private;

  procedure Initialize(Object : in out Limited_Controlled);
  procedure Finalize    (Object : in out Limited_Controlled);
private
  ... --not specified by the language
end Ada.Finalization;

```

## C.14 The Package Ada.Tags

```

package Ada.Tags is
  type Tag is private;

  function Expanded_Name(T : Tag) return String;
  function External_Tag(T : Tag) return String;
  function Internal_Tag(External : String) return Tag;

  Tag_Error : exception;

private
  ... --not specified by the language
end Ada.Tags;

```

## C.15 The Package Ada.Calendar

```

package Ada.Calendar is
  type Time is private;

  subtype Year_Number is Integer range 1901 .. 2099;
  subtype Month_Number is Integer range 1 .. 12;
  subtype Day_Number is Integer range 1 .. 31;
  subtype Day_Duration is Duration range 0.0 .. 86_400.0;

  function Clock return Time;

  function Year (Date : Time) return Year_Number;
  function Month (Date : Time) return Month_Number;
  function Day (Date : Time) return Day_Number;
  function Seconds(Date : Time) return Day_Duration;

  procedure Split (Date : in Time;
                  Year : out Year_Number;
                  Month : out Month_Number;
                  Day : out Day_Number;
                  Seconds : out Day_Duration);

  function Time_Of(Year : Year_Number;
                  Month : Month_Number;
                  Day : Day_Number;
                  Seconds : Day_Duration := 0.0)
    return Time;

```

```

function "+" (Left : Time;   Right : Duration) return Time;
function "+" (Left : Duration; Right : Time) return Time;
function "-" (Left : Time;   Right : Duration) return Time;
function "-" (Left : Time;   Right : Time) return Duration;

function "<" (Left, Right : Time) return Boolean;
function "<=" (Left, Right : Time) return Boolean;
function ">" (Left, Right : Time) return Boolean;
function ">=" (Left, Right : Time) return Boolean;

Time_Error : exception;

private
... --not specified by the language
end Ada.Calendar;

```

## C.16 The Package System

```

package System is
  pragma Preelaborate(System);

  type Name is implementation-defined-enumeration-type;
  System_Name : constant Name := implementation-defined;

  --System-Dependent Named Numbers:

  Min_Int           : constant := root_integer'First;
  Max_Int           : constant := root_integer'Last;

  Max_Binary_Modulus : constant := implementation-defined;
  Max_Nonbinary_Modulus : constant := implementation-defined;

  Max_Base_Digits    : constant := root_real'Digits;
  Max_Digits         : constant := implementation-defined;

  Max_Mantissa       : constant := implementation-defined;
  Fine_Delta         : constant := implementation-defined;

  Tick              : constant := implementation-defined;

```



```

--Storage-related Declarations:
type Address is implementation-defined;
Null_Address : constant Address;

Storage_Unit : constant := implementation-defined;
Word_Size    : constant := implementation-defined * Storage_Unit;
Memory_Size  : constant := implementation-defined;

--Address Comparison:
function "<" (Left, Right : Address) return Boolean;
function "<=" (Left, Right : Address) return Boolean;
function ">" (Left, Right : Address) return Boolean;
function ">=" (Left, Right : Address) return Boolean;
function "=" (Left, Right : Address) return Boolean;
--function "/=" (Left, Right : Address) return Boolean;
--"/=" is implicitly defined
pragma Convention(Intrinsic, "<");
--and so on for all language-defined subprograms in this package

--Other System-Dependent Declarations:
type Bit_Order is (High_Order_First, Low_Order_First);
Default_Bit_Order : constant Bit_Order;

```

```

--Priority-related declarations (see D.1):
subtype Any_Priority is Integer range implementation-defined;
subtype Priority is Any_Priority
    range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority
    range Priority'Last+1 .. Any_Priority'Last;

Default_Priority : constant Priority :=
    (Priority'First + Priority'Last)/2;

private
    ... --not specified by the language
end System;

```

# Appendix D: Answers to selected exercises

## From chapter 2

A program to print the first 20 numbers.

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
begin
  for I in 1 .. 20 loop
    Put( Integer'Image( I ) ); New_Line;
  end loop;
end Main;
```

A program to print the 8 times table.

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
begin
  for I in 1 .. 12 loop
    Put( " 8 * " ); Put( Integer'Image(I) ); Put( " = " );
    Put( Integer'Image( I*8 ) ); New_Line;
  end loop;
end Main;
```

A program to print numbers in the Fibonacci series.

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  First, Second, Next : Integer;
begin
  First  := 0;
  Second := 1;
  Put( Integer'Image(1) ); New_Line;
  while Second < 10000 loop
    Put( Integer'Image( Second ) ); New_Line;
    Next := First + Second;
    First := Second;
    Second := Next;
  end loop;
end Main;
```

A program to print a character table.

```

with Ada.Text_IO;
use   Ada.Text_IO;
procedure Main is
begin
  for I in 32 .. 127 loop
    Put( "Character " ); Put( Character'Val(I) );
    Put( " is represented by code " ); Put( Integer'Image(I) );
    New_Line;
  end loop;
end Main;

```

## From chapter 3

A program to print an arbitrary time table in the range 1 .. 20.

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use   Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  subtype Valid_Times_Table is Integer range 1 .. 20;
  Table, Last : Integer;
begin
  if Argument_Count >= 1 then
    Get( Argument(1), Table, Last );
    if Table in Valid_Times_Table then
      Put("The "); Put( Table, Width=>2 );
      Put(" times table is"); New_Line;
      for I in 1 .. 12 loop
        Put( Table, Width=>2 ); Put( " * " );
        Put( I, Width=>2 ); Put( " = " );
        Put( I*Table, Width=>3 ); New_Line;
      end loop;
    else
      Put("Number not valid"); New_Line;
    end if;
  else
    Put("No argument specified"); New_Line;
  end if;
end Main;

```

A program to determine if a number is prime or not.

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  Num : Integer;
begin
  Put("Please enter number : "); Get( Num );
  Put("Number is ");
  if Num in Positive then
    for I in 2 .. Num-1 loop
      if (Num/I)*I = Num then
        Put("not ");
        exit;
      end if;
    end loop;
    Put("prime"); New_Line;
  else
    Put("Require a positive number"); New_Line;
  end if;
end Main;

```

A program to covert a temperature in Fahrenheit to centigrade.

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  subtype Centigrade is Float range -32.0/1.8 .. 212.0/1.8;
  subtype Fahrenheit is Float range 0.0 .. 212.0;
  Temp      : Float;
begin
  Put("Please enter temperature in Fahrenheit ");
  Get( Temp );
  if Temp in Fahrenheit then
    Put("Temperature in Centigrade is ");
    Put( ( Temp -32.0 ) / 1.8, Exp=>0, Aft=>2 ); New_Line;
  else
    Put("Temperature not valid"); New_Line;
  end if;
end Main;

```

A program to print student marks as grades.

```

with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Name_Length : constant Positive := 20;
  type Name_Range is range 1 .. Name_Length;
  Ch : Character;
  Mark: Integer;
begin
  while not End_Of_File loop
    for I in Name_Range loop
      Get(Ch); Put(Ch);
    end loop;
    Get( Mark );
    case Mark is
      when 0 .. 39 => Put("F");
      when 40 .. 49 => Put("D");
      when 50 .. 59 => Put("C");
      when 60 .. 69 => Put("B");
      when 70 ..100 => Put("A");
      when others => Put("Invalid data");
    end case;
    Skip_Line; New_Line;
  end loop;
end Main;

```

## From chapter 4

A program to print statistics on the number of different types of character in a file.

```

with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  type Char is ( Digit, Punctuation, Letter, Other_Ch );

  function What_Is_Char( Ch:in Character ) return Char is
  begin
    case Ch is
      when 'a' .. 'z' | 'A' .. 'Z' => return Letter;
      when '0' .. '9'              => return Digit;
      when ',' | '.' | ';' | ':'   => return Punctuation;
      when others                  => return Other_Ch;
    end case;
  end What_Is_Char;

  No_Letters      : Natural := 0;
  No_Digits       : Natural := 0;
  No_Punct_Ch     : Natural := 0;
  No_Other_Ch     : Natural := 0;
  Ch              : Character;
begin

```

```

while not End_Of_File loop
  while not End_Of_Line loop
    Get( Ch );
    case What_Is_Char( Ch) is
      when Letter      => No_Letters := No_Letters + 1;
      when Digit       => No_Digits := No_Digits + 1;
      when Punctuation => No_Punct_Ch := No_Punct_Ch + 1;
      when Other_Ch    => No_Other_Ch := No_Other_Ch + 1;
    end case;
  end loop;
  Skip_Line;
end loop;
Put("Letters are      "); Put( No_Letters ); New_Line;
Put("Digits are      "); Put( No_Digits ); New_Line;
Put("Punctuation chs are "); Put( No_Punct_Ch ); New_Line;
Put("Other chs are    "); Put( No_Other_Ch ); New_Line;
end Main;

```

*Note:* This only works for the English character set.

A program to print the average of three rainfall readings.

```

with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
  procedure Order3( A,B,C:in out Float ) is
    procedure Order2( F,S:in out Float ) is
      Tmp : Float;
    begin
      if F > S then
        Tmp := F; F := S; S := Tmp;
      end if;
    end Order2;
  begin
    Order2( A, B ); --S L ? (a, b, correct order)
    Order2( B, C ); --? ? L ( c is largest)
    Order2( A, B ); --S M L (a, b, c ordered )
  end Order3;
  First,Second,Third : Float;
begin
  Put("Input 3 rainfall reading ");
  Get( First ); Get( Second ); Get( Third ); --Data
  Put("Rainfall average is      : ");
  Put( (First+Second+Third)/3.0, Exp=>0, Aft=>2 ); --Average
  New_Line;
  Order3( First, Second, Third ); --Order
  Put("Data values (sorted) are : "); --List
  Put( First, Exp=>0, Aft=>2 ); Put(" ");
  Put( Second, Exp=>0, Aft=>2 ); Put(" ");
  Put( Third, Exp=>0, Aft=>2 ); Put(" ");
  New_Line;
end Main;

```

## From chapter 5

A class Performance that represents the number of seats at a cinema performance.

```

package Class_Performance is

  type Performance is private;
  subtype Money is Float;

  procedure Book_Seats( The:in out Performance; N:in Natural );
  procedure Cancel( The:in out Performance; N:in Natural );
  function Sales( The:in Performance ) return Money;
  function Seats_Free( The:in Performance ) return Natural;

private
  Max_Seats : constant Natural := 200;
  Seat_Price: constant Money := 4.50;
  type Performance is record
    Seats_Left : Natural := Max_Seats;
  end record;
end Class_Performance;

```

```

package body Class_Performance is

  procedure Book_Seats( The:in out Performance; N:in Natural ) is
  begin
    if The.Seats_Left >= N then
      The.Seats_Left := The.Seats_Left - N;
    end if;
  end Book_Seats;

  procedure Cancel( The:in out Performance; N:in Natural ) is
  begin
    The.Seats_Left := The.Seats_Left + N;
  end Cancel;

  function Sales( The:in Performance ) return Money is
  begin
    return Float(Max_Seats-The.Seats_Left) * Seat_Price;
  end Sales;

  function Seats_Free( The:in Performance ) return Natural is
  begin
    return The.Seats_Left;
  end Seats_Free;

end Class_Performance;

```

A program to deal with the day-to-day administration for a cinema which has three performances.

```

with Ada.Text_IO, Ada.Float_Text_IO, Class_Performance, Class_Tui;
use  Ada.Text_IO, Ada.Float_Text_IO, Class_Performance, Class_Tui;
procedure Main is
  procedure Process(Per:in out Performance; Name:in String) is
    function Money_Image( M:in Money ) return String is
      Res : String( 1 .. 10 );           --String of 10 characters
    begin
      Put( Res, M, Aft=>2, Exp=>0 );    --2 digits - NO exp
      return Res;
    end Money_Image;
    Screen : Tui;                       --The TUI screen
    Tickets : Integer;                  --Tickets being processed
  begin
    loop
      Message( Screen, "Performance is " & Name );
      Menu( Screen, "Book", "Cancel", "Seats free", "Sales" );
      case Event( Screen ) is
        when M_1 =>                     --Book
          Dialog(Screen, "Number of seats to book", Tickets);
          if Tickets>0 and then Tickets<=Seats_Free(Per) then
            Book_Seats( Per, Tickets );
          else
            Message(Screen, "Not a valid number of tickets");
          end if;
        when M_2 =>                     --Cancel
          Dialog(Screen, "Number of seats to return", Tickets);
          if Tickets > 0 then
            Cancel( Per, Tickets );
          else
            Message( Screen, "Not a valid number of tickets" );
          end if;
        when M_3 =>                     --Free
          Message( Screen, "Number of seats free is " &
            Integer'Image( Seats_Free(Per) ) );
        when M_4 =>                     --Value
          Message( Screen, "Value of seats sold is f" &
            Money_Image( Sales(Per) ) );
        when M_Quit =>                  --Exit
          exit;
        end case;
      end loop;
    end Process;

    Afternoon, Early_Evening, Evening : Performance;
    Main_Menu : Tui;
  begin
    loop
      Menu(Main_Menu, "Afternoon", "Early Evening", "Evening", "");
      case Event( Main_Menu ) is
        when M_1 => Process( Afternoon, "Afternoon" );
        when M_2 => Process( Early_Evening, "Early evening" );
        when M_3 => Process( Evening, "Evening" );
        when M_Quit => exit;
        when others => Message( Main_Menu, "Try again");
      end case;
    end loop;
  end Main;

```

## From chapter 6

A fragment of code showing a data structure that represents a computer system.



```

declare
  Kb : constant := 1;           --In Kilobyte units
  Mb : constant := 1024;        --In Kilobyte units
  Gb : constant := 1024*Mb;     --In Kilobyte units
  type Main_Memory is range 0 .. 64*Mb;
  type Cache_Memory is range 0 .. 2*Mb;
  type Disk_Memory is range 0 .. 16*Gb;
  type Video_Memory is range 0 .. 8*Mb;
  type Computer is (Pc, Workstation, Multimedia );
  type Network is (Either, Ring );

  type Computer_System(Type_Of:Computer:=Pc) is record
    Main : Main_Memory;         --In Megabytes
    Cache: Cache_Memory;        --In Kilobytes
    Disk : Disk_Memory;         --In Megabytes
    case Type_Of is
      when Workstation =>
        Connection : Network;
      when Multimedia =>
        Display_Memory: Video_Memory;
      when Pc
        =>
        null;
      end case;
  end record;
  My_Computer: Computer_System(Pc);
  At_Work : Computer_System;
begin
  My_Computer := ( Pc, 256*Mb, 512*Kb, 18*Gb );
  At_Work := ( Pc, 128*Mb, 512*Kb, 6*Gb );
end;

```

Note: Kb, Mb, and Gb are scaled so that the number is representable.

## From chapter 9

```

package Pack_Types is
  subtype Money is Float range 0.0 .. Float'Last;
  subtype Hours is Float range 0.0 .. 24.0*7.0;
  Tax : constant Float := 0.8;
end Pack_Types;

```

```

with Pack_Types; use Pack_Types;
package Class_Emp_Pay is

  type Emp_Pay is tagged private;

  procedure Set_Hourly_Rate(The:in out Emp_Pay; R:in Money);
  procedure Add_Hours_Worked(The:in out Emp_Pay; H:in Hours);
  function Pay( The:in Emp_Pay ) return Money;
  procedure Reset(The:in out Emp_Pay);
  function Hours_Worked( The:in Emp_Pay ) return Hours;
  function Pay_Rate( The:in Emp_Pay ) return Money;

private
  type Emp_Pay is tagged record
    Worked : Hours := 0.0;    --Hours worked in week
    Rate : Money := 0.0;      --Rate per hour
  end record;
end Class_Emp_Pay;

```

```

package body Class_Emp_Pay is
  procedure Set_Hourly_Rate(The:in out Emp_Pay; R:in Money) is
  begin
    The.Rate := R;
  end Set_Hourly_Rate;

  procedure Add_Hours_Worked(The:in out Emp_Pay; H:in Hours) is
  begin
    The.Worked := The.Worked + H;
  end Add_Hours_Worked;

  function Pay( The:in Emp_Pay ) return Money is
  begin
    return The.Rate * The.Worked * Tax;
  end Pay;

  procedure Reset(The:in out Emp_Pay) is
  begin
    The.Rate := 0.0; The.Worked := 0.0;
  end Reset;

  function Hours_Worked( The:in Emp_Pay ) return Hours is
  begin
    return The.Worked;
  end Hours_Worked;

  function Pay_Rate( The:in Emp_Pay ) return Money is
  begin
    return The.Rate;
  end Pay_Rate;
end Class_Emp_Pay;

```

```

with Pack_Types, Class_Emp_Pay;
use Pack_Types, Class_Emp_Pay;
package Class_Better_Emp_Pay is

  type Better_Emp_Pay is new Emp_Pay with private;

  procedure Set_Overtime_Pay(The:in out Better_Emp_Pay;
                             R:in Money);
  procedure Normal_Pay_Hours(The:in out Better_Emp_Pay;
                              H:in Hours);
  function Pay( The:in Better_Emp_Pay ) return Money;

private
  type Better_Emp_Pay is new Emp_Pay with record
    Normal_Hours : Hours := 0.0;    --Normal pay hours
    Over_Time_Pay : Money := 0.0;   --Overtime rate
  end record;
end Class_Better_Emp_Pay;

```

```

package body Class_Better_Emp_Pay is

  procedure Set_Overtime_Pay(The:in out Better_Emp_Pay;
                             R:in Money) is
  begin
    The.Over_Time_Pay := R;
  end Set_Overtime_Pay;

  procedure Normal_Pay_Hours(The:in out Better_Emp_Pay;
                              H:in Hours) is
  begin
    The.Normal_Hours := H;
  end Normal_Pay_Hours;

  function Pay( The:in Better_Emp_Pay ) return Money is
    Get : Money;
  begin
    if Hours_Worked(The) > The.Normal_Hours then
      Get := The.Normal_Hours * Pay_Rate(The) +
        (Hours_Worked(The)-The.Normal_Hours)*The.Over_Time_Pay;
      return Get * Tax;
    else
      return Pay( Emp_Pay(The) );
    end if;
  end Pay;

```

```

end Class_Better_Emp_Pay;
with Ada.Text_IO, Ada.Float_Text_IO, Class_Emp_Pay, Class_Better_Emp_Pay;
use Ada.Text_IO, Ada.Float_Text_IO, Class_Emp_Pay, Class_Better_Emp_Pay;
procedure Main is
  Mike      : Emp_Pay;
  Corinna   : Better_Emp_Pay;
begin
  Set_Hourly_Rate( Mike, 10.00 );
  Add_Hours_Worked( Mike, 40.0 );
  Put( "Mike gets      : " ); Put( Pay(Mike), Exp=>0, Aft=>2 );
  New_Line;
  Set_Hourly_Rate( Corinna, 10.00 );
  Set_Overtime_Pay( Corinna, 11.00 );
  Normal_Pay_Hours( Corinna, 30.0 );
  Add_Hours_Worked( Corinna, 40.0 );
  Put( "Corinna gets : " ); Put( Pay(Corinna), Exp=>0, Aft=>2 );
  New_Line;
end Main;

```

## From chapter 13

A program to use a generic data store. The specification for the class Store is

```

generic
  type Store_Index    is private;      --
  type Store_Element  is private;      --
package Class_Store is
  type Store is limited private;       --NO copying
  Not_There, Full : exception;

  procedure Add    ( The:in out Store;
    Index:in Store_Index;
    Item:in Store_Element);
  function Deliver( The:in Store;
    Index:in Store_Index )
    return Store_Element;
private
  Max_Store : constant := 10;
  type      Store_R_Index is range 0 .. Max_Store;
  subtype Store_R_Range is Store_R_Index range 1 .. Max_Store;
  type Store_Record is record
    Index: Store_Index;      --Index
    Item : Store_Element;    --Data item
  end record;
  type Store_Array is array( Store_R_Range ) of Store_Record;
  type Store is limited record
    Data : Store_Array;
    Items: Store_R_Index := 0;
  end record;
end Class_Store;

```

A possible implementation of the Class Store is:

```

package body Class_Store is

  procedure Add    ( The:in out Store;
    Index:in Store_Index;
    Item:in Store_Element) is
  begin
    if The.Items < Max_Store then
      The.Items := The.Items + 1;
      The.Data(The.Items) := ( Index, Item );
    else
      raise Full;
    end if;
  end Add;

  function Deliver( The:in Store;
    Index:in Store_Index)
    return Store_Element is
  begin
    for I in 1 .. Store_R_Range(Max_Store) loop
      if The.Data(I).Index = Index then
        return The.Data(I).Item;
      end if;
    end loop;
    raise Not_There;
  end Deliver;
end Class_Store;

```

```

package Pack_Types is
  subtype Name is String( 1..5 );
end Pack_Types;

```

The instantiation of a store package to hold student names and exam marks is:

```
with Class_Store, Pack_Types;
package Class_Store_Int_Str is
new Class_Store( Pack_Types.Name, Integer );
```

A simple test program for this package is:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Store_Int_Str;
use  Ada.Text_IO, Ada.Integer_Text_IO, Class_Store_Int_Str;
procedure Main is
  Marks : Store;
begin
  Add( Marks, "Andy ", 50 );
  Add( Marks, "Bob  ", 65 );
  Add( Marks, "Clark", 73 );
  Add( Marks, "Dave ", 54 );
  Put("Mark for Bob  is " );
  Put( Deliver(Marks, "Bob  "), Width=> 3 ); New_Line;
  Put("Mark for Dave is " );
  Put( Deliver(Marks, "Dave "), Width=> 3 ); New_Line;
end Main;
```

## From chapter 14

A queue implemented using dynamically allocated storage. The specification of the class Queue:

```
with Ada.Finalization;
use  Ada.Finalization;
generic
  type T is private;           --Can specify any type
package Class_Queue is
  type Queue is new Limited_Controlled with private;
  Queue_Error: exception;

  procedure Add( The:in out Queue; Item:in T );
  procedure Sub( The:in out Queue; Item:out T );
  procedure Finalize( The:in out Queue );
private
  type Node;                  --Mutually recursive def
  type P_Node is access Node;  --Pointer to a Queue
  pragma Controlled( P_Node ); --We do deallocation

  type Node is record          --Node holds the data
    Item    : T;                --The stored item
    P_Next  : P_Node;           --Next in list
  end record;

  type Queue is new Limited_Controlled with record
    Head    : P_Node := null;   --Head of Queue
    Tail    : P_Node := null;   --Tail of Queue
    No_Of    : Natural:= 0;      --Number in queue
  end record;
end Class_Queue;
```

The implementation of the class Queue.

```

with Unchecked_Deallocation;
package body Class_Queue is

  procedure Dispose is
    new Unchecked_Deallocation( Node, P_Node );

  procedure Add( The:in out Queue; Item:in T ) is
    Tmp : P_Node := new Node'( Item, null );
  begin
    if The.No_Of > 0 then
      The.Tail.P_Next := Tmp;           --Chain in
    else
      The.Head        := Tmp;           --Also head
    end if;
    The.Tail          := Tmp;           --New Tail
    The.No_Of         := The.No_Of + 1; --Inc no.
  end Add;

  procedure Sub( The:in out Queue; Item :out T ) is
    Tmp : P_Node;
  begin
    if The.No_Of > 0 then
      Item      := The.Head.Item;       --Recovered item
      Tmp       := The.Head;             --Node finished with
      The.Head := The.Head.P_Next;       --new head
      Dispose( Tmp );                   --Free storage
      The.No_Of := The.No_Of - 1;        --1 less in queue
    else
      raise Queue_Error;                 --Error
    end if;
  end Sub;

```

```

  procedure Finalize( The:in out Queue ) is
    Discard : T;
  begin
    for I in 1 .. The.No_Of loop      --Free storage
      Sub( The, Discard );
    end loop;
  end Finalize;

end Class_Queue;

```

The instantiation of an integer instance of the class Queue.

```

with Class_Queue;
package Class_Queue_Int is new Class_Queue(Integer);

```

A small test program to test the class Queue.

```

with Ada.Text_IO, Ada.Integer_Text_IO, Class_Queue_Int;
use   Ada.Text_IO, Ada.Integer_Text_IO, Class_Queue_Int;
procedure Main is
  Number_Queue : Queue;           --Queue of numbers
  Action       : Character;       --Action
  Number       : Integer;         --Number processed
begin
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is
          when '+' =>
            Get( Number ); Add(Number_Queue,Number);
            Put("add number = "); Put(Number); New_Line;
          when '-' =>
            Sub(Number_Queue,Number);
            Put("remove number = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      exception
        when Queue_Error =>
          Put("Exception Queue_error"); New_Line;
        when Data_Error =>
          Put("Not a number"); New_Line;
        when End_Error =>
          Put("Unexpected end of file"); New_Line; exit;
      end;
    end loop;
    Skip_Line;
  end loop;
end Main;

```

## From chapter 19

A task type which allows repeated calculations of a factorial value to be made is:

```

package Pack_Factorial is
  task type Task_Factorial is
    entry Calculate( F:in Natural );
    entry Deliver( Res:out Natural );
    entry Finish;
  end Task_Factorial;
end Pack_Factorial;

```

```

package body Pack_Factorial is
  task body Task_Factorial is
    --Implementation
    Factorial : Natural;
    Answer    : Natural := 1;
    --Initial value
  begin
    loop
      select
        accept Calculate( F:in Natural ) do
          --Store in buffer
          --Factorial
          Factorial := F;
        end Calculate;
        Answer := 1;
      begin
        for I in 2 .. Factorial loop
          --Calculate
          Answer := Answer * I;
        end loop;
      exception
        when Constraint_Error =>
          Answer := 0;
        end;
      accept Deliver( Res:out Natural ) do
        --Return answer
        Res := Answer;
      end Deliver;
    or
      accept Finish;
      --Get from buffer
      --Finished
      exit;
    end select;
  end loop;

  end Task_Factorial;
end Pack_Factorial;

```

A short test program for this task is:

```

with Ada.Text_IO, Ada.Integer_Text_IO, Pack_Factorial;
use   Ada.Text_IO, Ada.Integer_Text_IO, Pack_Factorial;
procedure Main is
  Fac    : Task_Factorial;
  Num    : Integer;
  Answer : Integer;
  --Factorial task
  --To calculate
  --Result of calculation
begin
  while not End_Of_File loop
    while not End_Of_Line loop
      Get( Num );
      Fac.Calculate( Num );
      Put("Factorial "); Put( Num, Width=>2 ); Put(" is ");
      Fac.Deliver( Answer );
      Put( Answer, Width=> 2 ); New_Line;
    end loop;
    Skip_Line;
  end loop;
  Fac.Finish;
  --Terminate task
end Main;

```



# References

Intermetrics (1995) *Ada 95 Rational*, Intermetrics, Inc, Cambridge, Massachusetts.

Intermetrics (1995) *Ada 95 Reference Manual*, Intermetrics, Inc, Cambridge, Massachusetts.

Taylor, B. (1995) Ada 95 Compatibility Guide in *Ada Yearbook 1995* (Ed Mark Ratcliffe), IOS press, pp. 260-313.

Whitaker, W.A. (1993) Ada - The Project. *ACM SIGPLAN Notices*, **28**(3), 299-331.

# Index

- , 54
- & operator, 119
- \*, 53
- \*\*, 54
- ..
  - case statement, 32
- /, 53
- /=, 56
- |
  - case statement, 32
- +, 54
- <, 56
- <=, 56
- =, 56
- >, 56
- >=, 56
- abstract class, 153
- accept, 292
- access, 216
  - all, 211, 216
    - Class', 233
  - constant, 216
  - value of a function, 223
- access constant, 211
- actual parameter, 64
- Ada
  - case sensitivity, 26
  - format of a program, 26
- adjust, 255, 267
  - assignment, 267
- aggregate
  - record, 95
- aliased
  - example, 209
- all
  - access, 211, 216
- allocator, 212
  - example, 212
- and, 56
- and then, 56
- append
  - to file, 276
- array
  - initializing, 115
  - slice, 118
  - unconstrained, 117
- array dynamic, 119
- arrays, 102
- assignment
  - adjust, 267
- attribute
  - Access', 209
  - Callable', 372
  - Class', 231
  - Digits', 373
  - First', 371
  - Float'Digits, 39
  - Float'First.i.attribute
    - Float'Last, 39
  - Float'Last, 39
  - Float'Size.i.attribute
    - Float'Digits, 39
  - Integer'First.i.attribute
    - Integer'Last, 38
  - Integer'Last, 38
  - Integer'Size, 38
  - Last', 372
  - Length', 372
  - Max', 371
  - Min', 371
  - Model\_epsilon', 373
  - Pos', 306
  - Pred', 372
  - Pred', 306
  - Range', 372
  - Safe\_first', 373
  - Safe\_last', 373
  - Storage\_size', 372
  - Succ', 372
  - Tag', 228
  - Terminated', 372
  - Unchecked\_Access', 225
  - Val', 306
- attributes
  - on a discrete object, 372
  - on a floating point object and type, 372
  - on a scalar object and type, 372
  - on a task object and type, 372
  - on an array, 104, 371
- base class, 148
- bitwise operator
  - and, 57
  - or, 57
- Boolean
  - example, 56, 57
- case, 31
- child library, 166
  - example, 167
  - generic, 205
  - visibility, 169
- class, 80
  - abstract, 153
  - base, 148
  - derived, 148
  - hiding the structure, 220
  - instance attribute, 79
  - instance method, 79
  - UML notation, 21, 78

- 'class, 233
- Class
  - attribute, 152
  - method, 152
- collection
  - heterogeneous, 232
- Command line
  - arguments, 34
- compile-time
  - consistency check, 49
- composition
  - UML notation, 20
- conflict
  - use of names in package, 82
- constant
  - access, 216
  - data structure, 95
  - declaration, 40
    - typed, 40
- constant Integer, 30
- constrained
  - record, 99
- constrained and unconstrained types
  - scalar, 48
- construct
  - declare, 45
- controlled
  - adjust, 255
  - finalization, 255
- controlled object, 159
  - adjust, 267
  - example, 160
  - finalize, 161
  - initialize, 161
- conversion
  - derived -> base class, 239
  - derived to base class, 152
  - Float to Integer, 43
  - Integer to Float, 43
  - scalar types, 40
  - view, 230
- converting
  - base class -> derived class, 240
- copy
  - deep, 261
  - input to output, 33
- create
  - file, 275
- Currency converter*, 327
- data structure
  - constant, 95
- data structures
  - access to members, 94
  - record, 94
- data\_error
  - exception, 183
- declaration
  - tentative, 213
- declare
  - construct, 45
- deep copy, 261
- delay
  - accept, 301
- derived class, 148
  - visibility rules, 152
- design
  - identifying objects, 128
- discriminant
  - default value, 97
  - record structure, 96
- downcasting
  - example, 239
- dynamic allocation of storage, 212
- dynamic array, 119
- dynamic binding, 228
- else, 28
  - select, 301
- elsif, 29
- encapsulation, 74
- end\_error
  - exception, 183
- end\_of\_file, 33, 34, 35
- end\_of\_line, 33, 34
- enumeration, 50
  - Character, 51
  - io, 273
- exception
  - Constraint\_error, 375
  - data\_error, 183
  - Data\_error, 376
  - end\_error, 183
  - End\_error, 376
  - example, 183
  - Mode\_error, 376
  - name\_error, 275
  - Name\_error, 186, 376
  - others, 184
    - name of, 184
  - program\_error, 375
  - Status\_error, 186, 376
  - Storage\_error, 375
  - Use\_error, 376
- exit, 31
- finalization, 159
  - Controlled, 162
  - Limited\_Controlled, 162
- finalize, 267
- first'
  - attribute array, 104
- fixed
  - io, 273
- float
  - io, 273
- Float, 38

## 422 *Appendix D*

- for, 29
- formal parameter, 64
- function, 60
  - access value, 223
  - local variables, 61
  - program unit, 60
- fusion, 128
- generic
  - child library, 205
  - formal subprograms, 198
  - inheritance, 206
  - instantiation, 191, 199
  - package, 195
  - procedure, 191
  - procedure example, 193
  - with, 198, 199
- guard to entry, 298
- heterogeneous collections, 232
- hiding base class methods, 163
- identifying objects, 128
- if, 28
- in, 54
  - parameter, 64, 66
- in out
  - parameter, 64, 66
- inheritance, 147
  - generic, 206
  - initialization & finalization, 159
  - multiple, 156
  - UML notation, 23
- initialization, 159
  - Controlled, 162
  - Limited\_Controlled, 162
  - using
    - assignment, 85
    - discriminant, 84
- initializing
  - array, 115
- input
  - character, 33
- input output
  - detailed examples, 273
- inspector, 82
- instance
  - method, 79
- Instance
  - attribute, 79
- instantiation
  - generic function, 191
  - generic package, 199
- integer
  - io, 273
- Integer
  - constant, 30
- intermediate results in expression, 47
- io
  - append to file, 276
  - create file, 275
  - of data structures, 277
  - open file, 275
- iteration
  - printing list, 214
- iterator for list, 249
- last'
  - attribute array, 104
- length'
  - attribute array, 104
- Lexical levels
  - declare, 359
  - example, 357
  - wholes in visibility, 359
- library package
  - unchecked\_deallocation, 217
- limited
  - record, 100
- limited private, 83
- list, 249
- local variables, 61
- loop, 31
- message, 74
- methodology
  - fusion, 128
- mixed language
  - program, 362
- multidimensional arrays, 113
- multiple inheritance, 156
- mutator, 82
- name\_error
  - exception, 275
- natural
  - subtype, 50
- new\_line, 33
- not, 57
- not in, 54
- object, 74
  - example of use, 75
  - UML notation, 20, 21
- Observable, 241, 242
- Observe-observer
  - implementation, 242
  - specification, 242
- observe-observer pattern, 240
- Observer, 241
- open
  - file, 275
- operator
  - dyadic, 53
  - monadic, 54
  - &, 119
  - \*, 53
  - \*\*, 54
  - ., 94
  - /, 53
  - /=, 56

- +
  - dyadic, 53
    - monadic, 54
  - <, 56
  - <=, 56
  - =, 56
  - >, 56
  - >=, 56
  - and, 56, 57
  - and then, 56
  - in, 54
  - mod, 53
  - not, 57
  - not in, 54
  - or, 56, 57
  - or else, 56
  - overloading, 171
  - rem, 53
- or, 56
- or (select), 300
- or else, 56
- others
  - case statement, 31
  - exception, 184
- out
  - parameter, 64, 66
- output
  - string, 32
- overloading, 67
  - operators, 171
  - renames, 68
- package
  - as a class, 80
  - child library, 166
  - example
    - Class\_account, 220
    - Class\_account\_ot, 160
    - Class\_board, 109, 111, 138
    - Class\_board (TUI), 321
    - Class\_building, 235
    - Class\_cell, 137
    - Class\_counter, 136
    - Class\_dialog, 349
    - Class\_histogram, 105
    - Class\_input\_manager, 335
    - Class\_interest\_account, 150
    - Class\_list, 250, 252, 253
    - Class\_menu, 351
    - Class\_menu\_title, 354
    - Class\_named\_account, 157
    - Class\_object\_rc, 265
    - Class\_Office, 230
    - Class\_piggy\_bank, 123
    - Class\_player, 144
    - Class\_rational, 172
    - Class\_Restricted\_account, 163
    - Class\_room, 229
    - Class\_root\_window, 335
    - Class\_screen, 135, 333
    - Class\_Set, 269
    - Class\_stack, 187, 195
    - Class\_string, 177
    - Class\_tui, 87, 89
    - Class\_window, 341
    - Class\_window\_control, 336
  - Pack\_factorial, 289
  - Pack\_is\_a\_prime, 289
  - Pack\_md\_io, 331
  - Pack\_threads, 296
  - raw\_io, 331
  - implementation, 76, 78
  - specification, 76, 77
  - standard, 81
  - use, 81
    - Ada.Characters, 106
  - with, 81
- parameter
  - actual, 64
  - by name, 70
  - by position, 70
  - default values to, 70
  - formal, 64
  - in, 64, 66
  - in out, 64, 66
  - out, 64, 66
  - variable number, 69
- polymorphism, 228
  - package names, 237
  - parameter to procedure, 231
- Pos'
  - attribute, 306
- positive
  - subtype, 50
- Pred'
  - attribute, 306
- private, 83
  - in a class, 77
- procedure, 62
  - example
    - sort, 202
  - program unit, 62
- program
  - case sensitivity, 26
  - hello world, 25
  - mixed language, 362
- protected type, 296
- put
  - float parameters, 42, 43
  - integer parameters, 42
- range'
  - attribute array, 104
- record
  - limited, 100
  - variant, 99

## 424 *Appendix D*

- record aggregate, 95
- record structure
  - discriminant, 96
  - nested, 96
- recursion, 66
  - printing list, 214
- reference counting, 262
- renames, 68
- rendezvous, 291
- representation clause
  - physical address, 308, 309
  - specific value enumeration, 306
- reverse, 29
- root integer, 46
- root real, 46
- run time dispatch, 232
- run-time
  - consistency check, 49
- scalar
  - Image, 41
  - type hierarchy, 52
- scientific notation, 38
- select, 300
  - or, 300
- sequential\_io
  - package, 277
- skip\_line, 33
- slice of an array, 118
- standard
  - package, 81
- standard types, 375
- statement
  - accept, 292
  - case, 31
  - for, 29
    - reverse, 29
  - if, 28
  - if else, 28
    - nested, 28
  - if elsif, 29
  - loop, 31
    - exit, 31
  - select, 300
  - select delay, 301
  - select else, 301
  - when entry, 298
  - while, 28
- storage
  - dynamic allocation, 212
- storage pool, 212
- string
  - type, 118
- subprogram
  - generic, 198
- subtype, 45
  - natural, 50
  - positive, 50
- Succ'
  - attribute, 306
- tagged type, 148
- task
  - example of use, 289
  - rendezvous, 291
- task type, 289
- tentative declaration, 213
- thread, 289, 290
- TUI
  - skeleton layout, 312
- type, 45, 74
  - protected, 296
  - tagged, 148
  - task, 289
  - the intermediate results, 47
  - type safety, 44
    - example, 44
- type string, 118
- types
  - implementation size, 375
- UML
  - class notation, 21
  - composition notation, 20
  - inheritance notation, 23
  - object notation, 20, 21
- unchecked\_deallocation, 217
  - use of, 221
- unconstrained
  - record, 97, 99
- unconstrained array, 117
- universal integer, 40
- use
  - example, 81
  - positioning in a package, 82
  - use type, 181
- Val'
  - attribute, 306
- variant
  - record, 99
- view conversion, 230
- visibility
  - wholes, 359
- visibility rules
  - derived class, 152
- when
  - case statement, 31
- while, 28
- with
  - example, 81
  - generic, 198, 199
  - positioning in a package, 82
  - record extension, 240
- www
  - information, 377