University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

J-Serializer – Java Programs Serializer

Ph.D. Study Report

Jaroslav Kačer

Technical Report No. DCSE/TR-2001-07
December, 2001

Distribution: public

Technical Report No. DCSE/TR-2001-07
December 2001

J-Serializer – Java Programs Serializer

Jaroslav Kačer

Abstract

This paper describes an early version J-Serializer, a Java tool for serialization of concurrent programs written in the Java language. It shows some basic concepts it is based on as well as some details of internal functioning and the conversion process.

This work was supported by the grant of the Czech Grant Agency "Developing SW Components for Distributed Environments", No. 201/99/0244.

Contents

List of Figures

1 Introduction to J-Serializer

J-Serializer is intended to be a platform-independent Java software for serialization of Java programs. Currently, only an initial version is available, performing only the basic tasks and showing that the ideas it is based on are realizable. In this short chapter, the basic characteristics of concurrent and serial (or serialized) programs will be explained, together with behavior of serialized programs generated by J-Serializer.

1.1 The Purpose of Serialization

The Java language provides a very easy and efficient way to write concurrent programs. Every program may consist of one or more *threads*. Threads are, in general, sequences of code which can be run concurrently to other threads. Whether they run really concurrently or not depends both on hardware (mainly the number of processors) and software (mainly the abilities of operating system used). It is the Java interpreter which manages thread execution and switching, according to the characteristics and constraints given by the programmer, e.g. threads' priorities, synchronized sections, thread suspension and reactivation and some others.

The first (initial, default) thread is always present in a Java program. It is created automatically by the JVM[1] when the program starts up and its code corresponds to the code placed in the method `main()` of the class given as the first parameter to the Java interpreter. From this thread, other threads can be created and started, either directly from the `main()` method or indirectly from a method invoked from `main()`. All such threads must be instances of a class derived[2] from the `Thread` class. A new thread is created in the same way as all other objects are, i.e. using the operator `new`. It is then started when its method `start()` is invoked, which tells the JVM to execute the code placed in the thread's `run()` method. After being started, the thread runs concurrently (or pseudo-concurrently) to other threads, as explained above, unless specified otherwise by the programmer. A thread finishes its execution by reaching the end of the `run()` method, by returning from this method using the `return` keyword or by being killed by another thread which invokes its `stop()` method[3].

However, the ease of writing concurrent programs in Java does not guarantee

[1]Java Virtual Machine

[2]The `extends` keyword serves this purpose.

[3]This method has been deprecated since JDK 1.2 and should not be used anymore.

that they will be free of errors. An example of such an error is a situation called *deadlock*, when all threads of a program are suspended because of cyclic dependencies, usually caused by shared resources.

In addition, it is very difficult to trace or debug concurrent programs. Thread switching is controlled by the JVM, unless special behavior is imposed to the program, using locks and signals. Usually, the moments and the points in the code where switching will take place are not predictable in any way. Also, they differ considerably from computer to computer, from OS to OS, from JVM to JVM. Even on the same computer, using the same OS and the same JVM, it is very unlikely to get the same sequence of switching operations at the same points of code and at the same time. The momentary system load also plays an important role, as well as the number of processors being available to the program. A program having constantly two threads will behave in a completely different manner on a two-processors computer than on a computer having a single processor.

The aim of J-Serializer is to suppress this natural behavior and replace it with a deterministic one, allowing to define rules that all threads within a Java program will obey.

1.2 The Model of Java Concurrent Programs

Now, let's put our focus on a typical Java concurrent program, its structure and behavior.

The program consists ot the following entities:

- *Threads* – A dynamic set of runnable objects. New threads may be created in another thread and they may die[4] during the execution of the program and they may be automatically destroyed by the garbage collector when there is not a reference to them anymore. Therefore, the number of threads in a program changes in time. Threads cannot be killed by other threads.

- *Monitors* – A dynamic set of objects holding global data. Methods manipulating with a monitor's data are synchronized, which prevents the data from becoming inconsistent because of concurrent reading and writing operations. Threads keep references to monitors whose data have to be shared among them. The number of monitors also changes

[4]A Java thread dies when it reaches its `run()` method's end.

in time since they can be dynamically created and automatically destroyed.

- *Local data* – Threads' attributes, their methods' local variables, and objects referenced to by threads (either an attribute or a method's local variable can be the reference).

A thread's code can be thought of as a sequence of 'local computing' blocks (there is no interaction with monitors), divided by calls to monitors' methods. These methods are possibly blocking, i.e. threads may get suspended inside a monitor's method and woken up later. They can also return a value which can be stored into a local variable.

```
Thread born
Local computing
A call to a monitor's method
Local computing
A call to a monitor's method
. . .
Thread dead
```

The state of a monitor can be defined as a set of its all data's values. The state of a thread can be defined as a set of its local data's values.

A monitor is in *consistent state* when no thread is currently executing its code, that is:

- when no thread is currently running inside its method, and/or
- when one or more threads are passivated inside one of its methods, using `wait()`.

A thread is in *consistent state* when its data are not being changed, that is:

- at the point when the thread calls a monitor's method, or
- when the thread is in passive state, or
- when the thread has died already but its data are still valid[5].

[5]It has not been destroyed by the garbage collector yet.

A program's behavior can be visualized using a *graph* whose nodes represent threads' consistent states and whose transitions represent changes made to local data between two consequent call to a monitor's methods. Since the code between two consequent calls can contain `if-else` statements and `switch` statements which can split program flow into several branches, more than one transition can lead from a node. This situation is depicted in figure 1 on page 6.

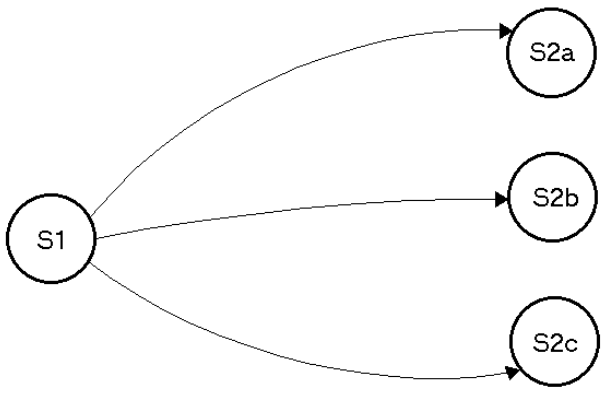

Figure 1: A Node with Three Transitions

The transitions of the graph correspond to the code working with local data and results of prior calls to monitors' methods. Therefore, each transition can be expressed as a function

$$next\_state = f(local\_data, last\_result\_of\_monitor's\_method)$$

All threads' graphs can be merged together into one, usually huge, graph – the *graph of program's states.* The purpose of J-Serializer is to assure that a deterministic algorithm will control the advancement from state to state in the whole graph. Some of the startegies how such an algorithm can make decisions are described later in chapter 1.4.

1.3 Two Parts of J-Serializer

In fact, J-Serializer is not a single program, doing all necessary work at once. Instead, it is composed of two more or less independent parts, each of which has a specific task to accomplish. They are as follows:

- J-Serializer's *internal classes, providing runtime support* for user programs. This part assures that programs using J-Serializer will behave according to a chosen strategy of program execution. It is located in Java package `cz.zcu.fav.kiv.jserializer`. Chapter 2 discusses this part in detail.
- J-Serializer's *converter.* This part converts original user's source files into a modified form which is forced to use services of the former part. It is located in package `cz.zcu.fav.kiv.jsercompiler` and will be discussed in chapter 3.

The runtime part can be thought of as a software layer placed between the application and the JVM. The situation before and after conversion is depicted in figure 2 on page 7.

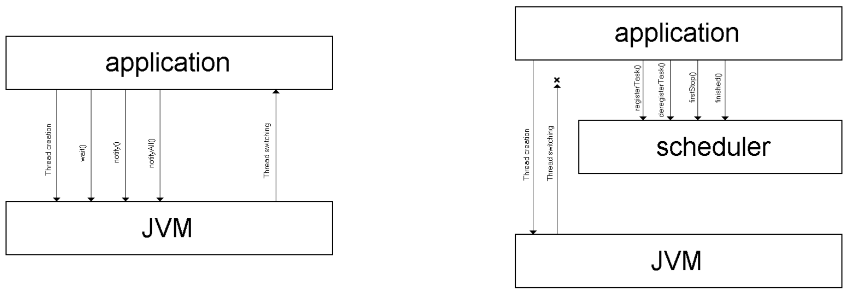

Figure 2: An Application Before and After Conversion

The situation after conversion differs from the former one in several points:

- The JVM does not switch threads anymore, causing them to be interrupted in a incosistent state. This is due to the fact that just one thread is running in the program after conversion and every attempt to switch from the currently running thread to another is thus ineffective.

- All calls to methods possibly changing state of one or more threads (`wait()`, `notify()`, `notifyAll()`) have been removed. Threads do not communicate directly with the JVM, they use public methods of a scheduler. The scheduler is an object controlling thread activity instead of the JVM. Interaction between the scheduler and the JVM is hidden for the user who does not need to care about its internal functioning.

The *scheduler* shown in the figure is a substantial part of the runtime package. It has a well defined interface that threads use. The scheduler has a built-in algorithm which determines the way how threads will be switched. As mentioned later in chapter 1.4, different strategies can be applied to the same original program, resulting in different results obtained when the converted program runs. The scheduler also maintains information about threads currently present in the program.

1.4 Expected Behavior of Serialized Programs

As said above, the scheduler can apply different rules when controlling program flow. Different strategies require different decision mechanisms built in the scheduler, therefore slightly different versions of the scheduler must be available. The following is a list of some possible strategies:

1. The *FIFO strategy.* Every thread has a unique ID number. Whenever the scheduler is asked to switch from the running thread to another, it selects the thread with the least ID possible, but bigger than the active thread's ID. If there is no thread fulfilling this condition, the thread with the least ID is selected.

2. The *strategy based on time measurement.* Threads measure elapsed time between two consequent switching points and report it to the scheduler. There exists a *calendar*, holding wake-up *events*, each of which contains information about time and a thread to be woken up. A new event is created when a thread asks the scheduler to switch. The events in the calender are ordered according to their time. When the scheduler is asked to switch, it takes the first event in the calendar and activates its corresponding thread. Then, the event is destroyed.

3. A *modified time measurement strategy.* Instead of the real time, an equivalent can be used. It can be a random number computed using the measured value (e.g. an exponential-distribution random number,

having $1/\lambda$ equal to the measured time) or another random numbers generator, using the measured time, can be used.

4. A *pseudo-random strategy.* Threads are switched in an order generated by a random numbers generator. If the generator is initialized always in the same way, the same results should be obtained.

5. The *user-driven strategy.* The user can interactively decide which thread will be selected to run next.

Note: Currently, J-Serializer supports the time measurement strategy only.

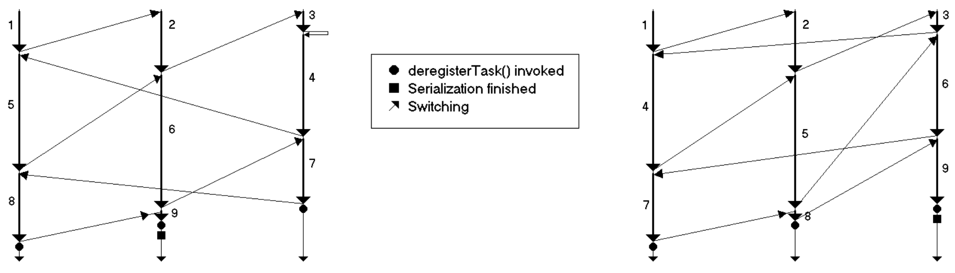

Figure 3: Different Strategies of Thread Switching

The difference between the first and second strategy mentioned above is depicted in figure 3 on page 9. There are three threads in the system. Each thread has three blocks of local computing, divided by two global actions (calls to a monitor's method). The fact that they all contain the same number of local computing blocks is a pure accident and has no special meaning. On the left, the time measurement strategy is shown, the FIFO strategy is on the right.

The decision rules of a chosen strategy are used at the very beginning of the program, and later at any time the scheduler decides to switch from a thread to another one. This includes also the situation when a thread is about to terminate and it is necessary to switch to another thread not to cause deadlock. The scheduler itself does not decide when to switch, it is asked by the currently running thread to do so. The thread invokes scheduler's `finished()` method[6] which suspends it and gives control to another thread.

In order to see what exactly happens during a serialized program's execution and to get a brief overview of how threads are switched, figure 4 should be examined.

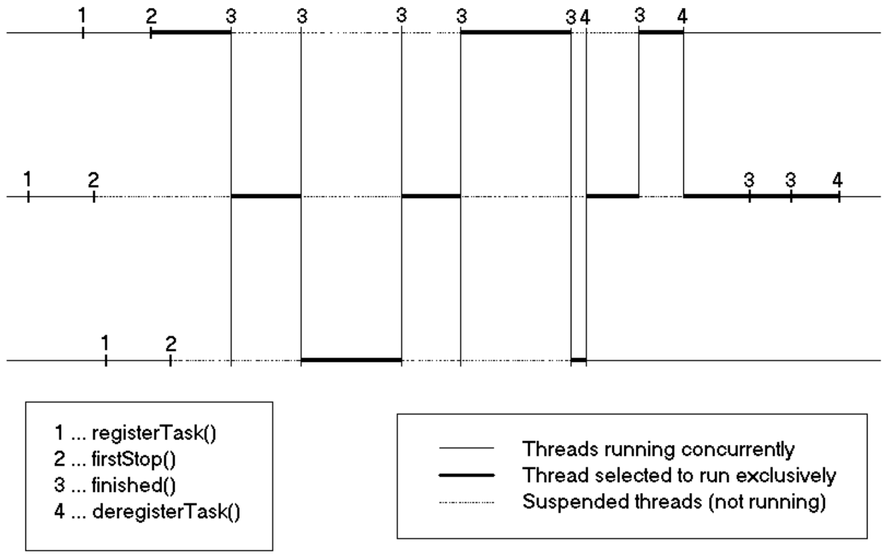
1
2
3
3
3
3
3 4
3
4
1
2
3
3
4
1
2
1 ... registerTask()
2 ... firstStop()
3 ... finished()
4 ... deregisterTask()
Threads running concurrently
Thread selected to run exclusively
Suspended threads (not running)

Figure 4: A Serialized Program in Real Time

The first thing which has to be done when a thread is run is to register it using the `registerTask()` method. This method returns a number indicating the thread's ID number which is unique within the whole program. Next, there is a point where all running threads passivate themselves, except of one which continues running – the scheduler's method `firstStop()`.

When the selected thread leaves `firstStop()`, it is running as the only active thread within the program until it reaches a call to the `finished()` method. A new evaluation is then made which results in activation of another thread and suspension of the currently running one.

At the end of a thread's life, it is 'automatically' removed from the scheduling mechanism by invoking scheduler's method `deregisterTask()`. This

[6]The switching mechanism is also implemented in methods `firstStop()` and `deregisterTask()`.

method does not passivate the calling thread, only activates another one according to the chosen strategy. Immediately after that, its `run()` method's end is reached and the thread becomes a passive object, like all standard Java objects. `deregisterTask()` also implements the switching mechanism as all other threads in the system are suspended and one of them must necessarily be activated. Otherwise, a deadlock would occur. Therefore, just before dying, every thread activates another thread in order for the program not to stop before finishing naturally. If there is no other thread in the system to be activated (the currently running thread is the only one), the thread simply returns from its `run()` method and the serialization is terminated. After that, no other thread should run in the system and the program should terminate.

2 Internal Functioning

This chapter describes the internal functioning of programs serialized with J-Serializer. The main focus is put on the mechanism of thread switching, implemented in classes of the runtime part of J-Serializer, but some other important topics – like shared objects and time measurement – will also be discussed. All classes mentioned in this chapter are part of the package `cz.zcu.fav.kiv.jserializer` which must be available for the Java interpreter when a program generated by J-Serializer is run.

2.1 Thread Switching

After having been started using its `start()` method, every thread runs completely independently from the others as well as from the thread which created and started it. According to the number of real processors, either one or more threads can run at a given time. If there are more threads than processors, the JVM switches them in a unpredictable way. In order to switch them deterministicly and replace thus the JVM's natural behavior, *methods of synchronization* must be used.

2.1.1 Unusable Synchronization Methods

The `Thread` class provides three very promising methods allowing to manipulate with a thread's execution state: `suspend()`, `resume()`, and `stop()`. The `suspend()` method temporarily suspends execution of a thread until `resume()` is invoked on the same thread. The last method interrupts execution of a thread and causes it to exit its `run()` method. However, all of them have been marked as deprecated since JDK[7] 1.2. There were very serious reasons to do so, mainly concerning security. The reasons can be found in [Lea].

2.1.2 Usable Synchronization Methods

However, Java provides a very efficient way of thread synchronization, although it is not obvious for the first sight. A method of a class can be marked as `synchronized` (using the `synchronized` keyword in its header) which will prevent all threads from entering method's body if there is already

[7]Java Development Kit

another thread. A *lock* is used to keep information whether it is possible to enter the method. If there are more protected methods within a class, they share the same lock. Note that the lock belongs to an object (instance of a class) not to a class. Only static synchronized methods use class locks. Here is a simple example of two synchronized methods sharing the same lock:

```
public synchronized void a()
{ /* ... something is happening here ... */ }

public synchronized void b()
{ /* ... and something else here ... */ }
```

When a thread enters the `a()` method, the lock is acquired and no other thread can enter this nor the `b()` method. All threads attempting to do so are temporarily suspended and one of them is resumed when the first thread leaves the `a()` method and the lock is released.

However, methods or just their parts can be synchronized by another lock than the implicit one (`this`). The following piece of code protects the methods' code in exactly the same way as the last one, but uses an explicit lock which must have been properly initialized. The lock can be an instance of any class.

```
public void a()
{
 synchronized (theLock)
 { /* ... something is happening here ... */ }
}

public void b()
{
 synchronized (theLock)
 { /* ... and something else here ... */ }
}
```

The latter (more complicated) possibility is not used in J-Serializer since switching takes place only in the scheduler's methods. Since the scheduler is shared by all threads (and non-threads too), there is no need for creating a shared lock and the scheduler itself serves as the lock.

There are techniques offered by Java that allow a thread to become temporarily suspended inside a synchronized method (or just a block of code) in order to release the lock and thus allow another thread to enter the synchronized method or block. Method `wait()`, introduced already in the class `Object`, provides this service. The suspended thread must be woken up sometimes not to spend all its life in passive state, being not able to do anything. There exist two methods usable for this purpose: `notify()` and `notifyAll()`. They do not differ in behavior if there is just one thread suspended. If there are more than one thread suspended, the `notify()` method wakes up just one of them, it is not specified which one. The `notifyAll()` method wakes up all of them. However, they cannot start to run parallelly since they wake up inside a synchronized block of code. Only one of them is allowed to run (not specified which one) and all others must wait until the selected thread leaves the synchronized block of code. As soon as it does so, another already woken-up thread is selected and allowed to run and so on. A typical example on how to use these methods (`notify()` is omitted here but the principle is the same) is as follows:

```
public void sleepUntilWokenUp()
{
 synchronized (theLock)
 {
  // This piece of code will be executed right after entering
  // the synchronized block. This can take a while, too,
  // if there is already another thread.
  theLock.wait();  // suspended
  // This piece of code will be executed
  // after the suspended thread
  // is woken up and the thread that woke him
  // up in wakeThemUp()
  // leaves the synchronized block.
 }
}

public void wakeThemUp()
{
 synchronized (theLock)
 {
  // This piece of code will be executed right after entering
  // the synchronized block. This can take a while, too,
  // if there is already another thread.
  theLock.notifyAll();  // wake up all suspended threads
  // This piece of code will be executed right after waking
  // other processes up, there is no time delay.
 }
}
```

Classes having these synchronized methods are not usually threads but *passive data containers*. They use synchronization to protect their data from being overwritten or read in a bad way by two concurrently running threads trying to change and read the value of a variable in the same moment. Such classes are called *monitors*.

2.1.3 Points of Switching

Before explaining the switching technique itself, it would be convenient to state where exactly the switching takes place. It is always in a synchronized method of the shared scheduler which is an instance of the class `Scheduler`. The methods are as follows:

- `Scheduler.firstStop()` – The main purpose of this method is to disable Java's natural switching from a thread to another one. This method is entered just once by every thread present in the system after obtaining its thread number from the scheduler. All threads, except of one, are passivated here and remain passive until woken up by a signal (sent by `notifyAll()`) from the only active thread. Whether a thread will be passivated here or selected to run depends on the criteria mentioned in chapter 1.4. Since all times are equal to zero (no thread has run yet), thread #0 is selected.
- `Scheduler.finished()` – The only currently running thread invokes this method in order to passivate itself and activate another thread according to the criteria from chapter 1.4. The thread calling the method is (in most cases) passivated and another thread is activated. Note that the thread being activated need not necessarily have been passivated in the same method. It may remain passive since its first passivation in `firstStop()`. This behavior is possible thanks to the fact that all three methods share the same lock for synchronization – the implicit scheduler's lock.

  In fact, a method named `finished()` does not exist. Instead, `finishedWithoutException()` and `finishedWithException()` are used. The only difference between them is the possibility of throwing `InterruptedException` from the body of `finishedWithException()`.

This difference is quite unimportant and has no influence on the principles of J-Serializer's functioning. Therefore, these two names will not be distinguished and the name '`finished()`' will be used for both of them. More information on these two functions will be provided in chapter 3.5.

- `Scheduler.deregisterTask()` – When a thread about to die (at the end of `run()`), it is necessary to wake up another thread which is in passive state. The thread to be woken up is selected in the same manner as it is in the `firstStop()` or `finished()` method. The only difference between `finished()` and `deregisterTask()` is that the calling thread is not passivated in order for it to be able to terminate.

2.1.4 Commands Used to Switch

The deterministic switching[8] used in J-Serializer is possible thanks to the following factors:

- A common lock, available to all threads, exists. It is the implicit lock of the shared scheduler.
- Every thread in the system has a unique ID number. The scheduler keeps information about the currently running thread in variable called `runningThread`. This variable is updated whenever the scheduler selects a thread to be run. If no thread is running, the constant `NO_THREAD` is set.

Let's have a look now at how the switching process is implemented in Java. A piece of code taken out from the `finished()` method is shown here, however, the principles remain the same in the two other methods.

First of all, ID numbers of the currently running thread is stored to the variable `myThread`. It is important to save this value before it changes inside the method as it will be needed later when a signal is received from another thread which wants to activate a passive thread. Only the scheduler keeps this information, not the threads themselves. Since the switching can take place in any object (including all non-threads), the threads would have to pass this information as parameters to every method of a non-thread which would be inefficient and hard to add to the source code during the conversion process.

[8]Such a kind of switching where there are exact rules that the switching algorithm must obey, e.g. rules mentioned in chapter 1.4.

```
myThread = getRunningThread();
```

Right after that, the elapsed time of the running thread is updated and the scheduler is informed that the thread is getting suspended.

```
 addTime(time);
 setRunningThread(NO_THREAD);
```

Then, it is necessary to select the next thread to run, respecting the criteria from chapter 1.4. All threads are asked to report the time they have elapsed and the thread having the minimum time is selected – its ID number is stored into `threadToRun`.

```
for (i = 0; i < MAX_THREADS; i++)
 if ((threadActive[i] == true) && (threads[i] != null) &&
     (minTime > threads[i].getTime()))
 {
  minTime = threads[i].getTime();
  threadToRun = i;
 } // if
```

When a thread is selected, its ID number is saved into `runningThread`.

```
runningThread = threadToRun;
```

Then, all threads present in the system, suspended with the same lock, are woken up by a signal sent by `notifyAll()`. The calling thread is not affected in any way and continues normally. It is necessary that `notifyAll()` be placed before the thread suspends itself. Otherwise, the waking-up signal would never be sent and all threads would remain passive forever.

```
notifyAll();
```

Finally, the thread can passivate itself using `wait()`. When it is later woken up by a signal sent by `notifyAll()` from another thread (which invoked either `finished()` or `deregisterTask()`), it compares its' numbers, saved at the very beginning, with the numbers set by the scheduler. If the numbers are identical, it leaves the while cycle and returns from the `finished()` method (or wherever this algorithm is used). Otherwise, the thread passivates itself again and waits for another signal sent during next switching.

```
while (runningThread != myThread)
{
 try { wait(); }
 catch (InterruptedException e)
 { System.err.println("Thread interrupted!"); }
} // while
```

2.1.5 A Switching Example

To better illustrate what has been said about the principles of switching, figure 5 on page 18 gives an explanation.

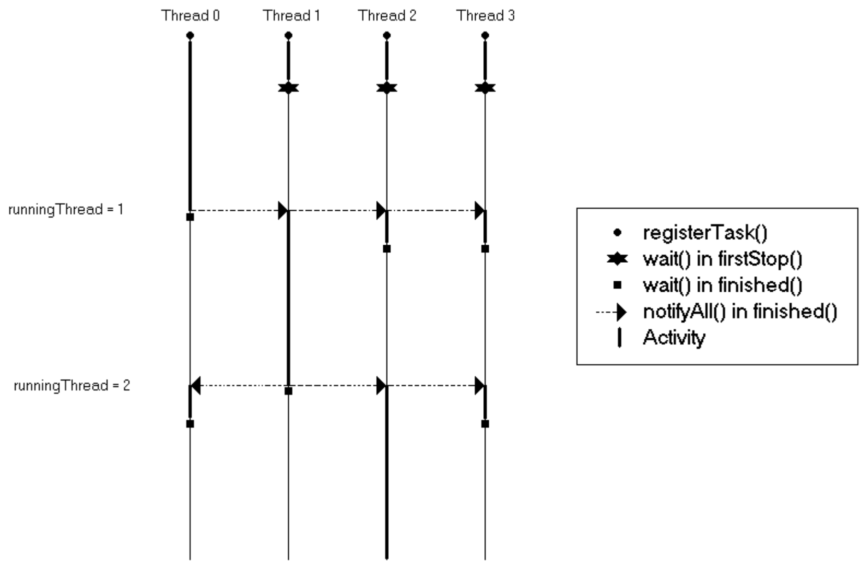

Figure 5: The Principles of Switching Illustrated

Four threads are present in the system. At the very beginning, threads #1, #2, and #3 are suspended in `firstStop()` while thread #0 is allowed to run. When it invokes `finished()` for the first time, thread #1 is selected by the scheduler to run. Therefore, number 1 is stored to `runningThread`.

Then, a signal sent by `notifyAll()` wakes up all passive threads. Immediately after that, thread #0 which sent the signal, passivates itself by using `wait()`. When other threads are woken up, they must decide whether to keep running or passivate themselves again. Only one of them can decide to run, here it is thread #1. The others (threads #2 and #3), after comparing their numbers to the numbers set by the scheduler, passivate themselves again by executing the body of the while cycle[9].

The selected thread runs as the only active thread within the program, which prevents the JVM to switch to another one – there is no thread to swich to. When it later calls the `finished()` method or the `deregisterTask()` method, the same algorithm is used again.

2.2 Access to Shared Objects

As already explained, it is necessary for all threads within a program to have access to a scheduler which controls thread switching. In fact, it is not the only shared object. Another one is an object measuring time between two consequent switching points – a *timekeeper*.

Since switching may occur in any method of a thread but in any method of a non-thread[10] as well, it is necessary for the two shared objects to be available to all Java objects in the same fashion. Figure 6 on page 20 shows the solution to this problem.

The principles used for both objects are identical and thus will be explained at once. The classes providing their instances for sharing (`Scheduler` and `StandardJavaTimeKeeper`) have static fields of the same type. The class `Scheduler` has the field `defaultScheduler` and the class `StandardJavaTimeKeeper` has the field `sharedStdJavaTK`. These two fields are created in the static block of their respective class, i.e. at the time the class is loaded by the JVM. Because of their access modifier (`private`), they cannot be accessed directly nor modified nor destroyed from outside their class. A reference to them can be obtained only by using the static method `getDefaultScheduler()` or `getSharedTimeKeeper()`, respectively. Invoked from any context, these two methods always return the same results, references to the same objects.

A short piece of code (extracted from the file `Scheduler.java`) should clarify possible obscurity:

```
public class Scheduler
{
```

[9]See chapter 2.1.4.

[10]An instance of a class being not derived from `Thread`.

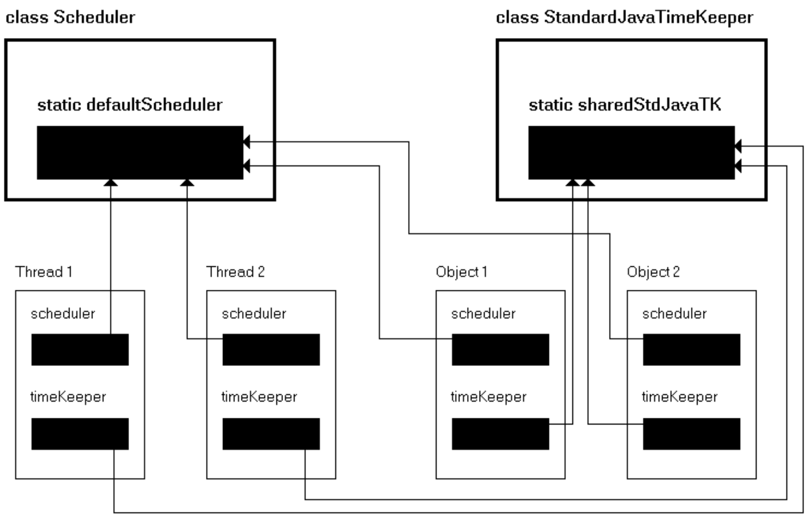

Figure 6: Access to Shared Objects

```
 . . . // Other fields
 private static Scheduler defaultScheduler;

 static
 { defaultScheduler = new Scheduler(); }

 public static Scheduler getDefaultScheduler()
 { return defaultScheduler; }
 . . . // Other methods, constuctor
} // class Scheduler
```

In case of the shared timekeeper, the situation is almost the same. A piece of code from `StandardJavaTimeKeeper.java` is as follows:

```
public class StandardJavaTimeKeeper
{
 . . . // Other fields
 private static StandardJavaTimeKeeper sharedStdJavaTK;

 static
 { sharedStdJavaTK = new StandardJavaTimeKeeper(); }

 public static AbstractTimeKeeper getSharedTimeKeeper()
 { return sharedStdJavaTK; }
 . . . // Other methods, constructor
} // class StandardJavaTimeKeeper
```

There are several possibilities where the two shared objects may be needed:

- In a method of a thread class[11].
- In a static method of a thread class.
- In a method of a non-thread class.
- In a static method of a non-thread class.

Normal (non-static) methods of a class may access both non-static and static fields of the class. However, static methods can access static fields only – they can be invoked even if there exists no instace of the class. Therefore, it is necessary for the shared objects to be marked as `static`. A static reference is present in every class converted by J-Serializer:

```
public class X
{
 private static AbstractTimeKeeper timeKeeper = null;
 private static Scheduler scheduler = null;
 . . . // Other fields and methods, constructor
}
```

Please ignore the fact that `timeKeeper` is typed as `AbstractTimeKeeper`. This will be explained in chapter 2.3.

The references just mentioned have to be initialized somehow. In threads, it is done in their constructors. In non-threads, it is done in every method because a method of a non-thread object can be invoked at any time without any other method having been called before or without the constructor having been called – it may be an implicit one. Updating the references to the scheduler and the timekeeper (they are static variables!) every time a new thread is created or a method of a non-thread class is invoked is not a dangerous

[11]A class derived from `Thread`, directly or indirectly.

operation since the same value is obtained every time the access methods `getDefaultScheduler()` and `getSharedTimeKeeper()` are called.

A better solution would be to put the initialization code directly to the place where these references are declared. This would simplify the conversion process and improve performace of the serialized program too, by omitting all unnecessary initializations in threads' constructors and all non-threads' methods.

2.3 Time Measurement

When the `finished()` method of the shared scheduler is invoked, a real number is passed to it, determining the amount of time which has passed since last call to `finished()`. This time is added to the sum of all thread's elapsed time which is the main criterion for selection of the next thread to run. The object which has to provide the time amount is called timekeeper. As said in chapter 2.2, the timekeeper is shared in the same way the scheduler is.

In this version of J-Serializer, a standard function for time measurement is used. It is `System.currentTimeMillis()` which returns the number of milliseconds since 0:00:00 of January 1, 1970. However, this function does not provide exact time measurement. There is always a certain limit. If a real time difference is less than the limit, the JVM reports it as equal to zero. A typical example follows:

```
long time1 = System.currentTimeMillis();
System.out.println("Hello");
long time2 = System.currentTimeMillis();
System.out.println("Difference = " + (time2-time1));
```

In most cases, `Difference = 0` is printed out although the real difference is certainly bigger. Meaningful results are returned when the real difference is around 50 milliseconds and longer. Therefore, to protect the decision mechanism from switching to the same thread again, J-Serializer adds 1.0 to the number passed as parameter from the timekeeper. In fact, using the real time measured by the timekeeper is not the only possibility how to provide a meaningful switching mechanism, as described in chapter 1.4.

2.3.1 Abstract Timekeeper

In order to provide the possibility of replacing the standard timekeeper by a user's own one, the abstract class `AbstractTimeKeeper` was introduced. Since the shared timekeeper is typed to `AbstractTimeKeeper`, an instance of any subclass of `AbstractTimeKeeper` can replace the standard one, if needed. For example, a more precise time measurement could be implemented by using native methods written in the C language. However, replacing the standard timekeeper requires rewriting some of the `.insert` files. More information can be found in chapter 3.4.

`AbstractTimeKeeper` has four methods which must be implemented or overwritten in order for an alternative timekeeper to function properly:

- `public abstract void start()` – This method starts counting time.
- `public abstract void stop()` – This method stops counting.
- `public abstract double lastDelta()` – This method returns the last interval measured, i.e. the difference of results returned by last calls to `stop()` and `start()`.
- `public static AbstractTimeKeeper getSharedTimeKeeper()` – This method returns a reference to the shared timekeeper. It should always return a reference to the same object. In this class, `null` is always returned.

2.3.2 Actually Used Timekeeper

The actual timekeeper used in J-Serializer is an instance of the class `StandardJavaTimeKeeper`. This class is derived from `AbstractTimeKeeper`. It overwrites `getSharedTimeKeeper()` and implements the four remaining methods. The time measurement is implemented in the way that Java programs normaly use – a call to the `System.currentTimeMillis()`. As described in 2.3, it has some features not allowing exact time measurement, however, it is still usable.

An instance of this class is created in the class' static block and returned from the overwritten `getSharedTimeKeeper()` method every time it is called. It assures that all objects have access to the same timekeeper, unique within the whole program.

3 The Conversion Process

So far, only the already serialized program has been discussed. However, before a program becomes serialized, it must be converted from its original form to a slightly modified form which uses package `cz.zcu.fav.kiv.jserializer`, discussed in chapter 2. This chapter describes the process of conversion and the package `cz.zcu.fav.kiv.jsecompiler` whose classes are a substantial part of the converter.

3.1 Prerequisites and Limitations

In order to run the converter succesfully, the user must assure the following condition of his computing environment:

1. A Java interpreter, at least version 1.2, must be installed and able to run. This includes the file `java` (or `java.exe`) and many support libraries, according to the operating system used.
2. A Java compiler, at least version 1.2, must be installed and able to run. This includes the file `javac` (or `javac.exe`) and many other files, usually shared with the interpreter. The compiler is not necessary for the conversion itself but the results (Java source files) would not be usable after the conversion process finishes.
3. The package `cz.zcu.fav.kiv.jsercompiler` must be located in a directory where the environment variable `CLASSPATH` points to.
4. The main converter's file – `SerMain.class` – must be placed in an arbitrary directory.
5. All `.insert` files must be placed in a subdirectory named `insert`, which is located in the same directory as `SerMain.class` is.

Failing to meet any of these requirements will result in an error during the conversion process.

When the converter loads a class and parses its source code (as described in chapter 3.3), it does not recognize all symbols of the Java grammar, as they are described in [JLS]. Instead, it searches for a certain sequence of symbols in the source code to get to an important point where a piece of code needs to be deleted, added, or replaced. Sometimes, there are more possibilities how symbols in a sequence may be ordered. However, to simplify the conversion process, not all expressions allowed in a standard Java source text are supported. In such a case, the conversion process will probably report an error and stop without converting all classes. The following list shows all known limitations of this version of J-Serializer, together with examples of correct and incorrect syntax:

1. If the class being converted belongs to a package, the `package` keyword and the name of the package must be placed on the same line.
   Correct: `package com.mycompany.mypackage;`
   Incorrect: `package` < *newline* > `com.mycompany.mypackage;`

2. If the class being converted is derived from another class, the `extends` statement must not contain the name of the package that the superclass is a part of.
   Correct: `public class MyClass extends String`
   Incorrect: `public class MyClass extends java.lang.String`

3. If the class being converted is derived from `Object`, the extends statement must not be used.
   Correct: `public class MyClass`
   Incorrect: `public class MyClass extends Object`

4. If two or more classes have the same source file (one of them is marked as `public`, the others have no class access modifier), they must not have methods with the same names.
   Correct: `ClassA.methodA()`, `ClassB.methodB()`
   Incorrect: `ClassA.methodA()`, `ClassB.methodA()`

5. If a method of a class is synchronized, the following order of the method's modifiers must strictly be used:
   ```
   public
   private
   protected  static  final  synchronized  strictfp
   ```
   Any modifier can be omitted but the order of the remaining ones must be respected. The `native` and `abstract` modifiers are not listed since native and abstract methods are never processed.

6. If the return type of a method is not a primitive type but a class, then its name must not contain the package where the class belongs to, if any.
   Correct: `public MyClass aMethod()`
   Incorrect: `public com.mycompany.mypackage.MyClass aMethod()`

7. The methods `wait()`, `notify()`, `notifyAll()`, and the keyword `synchronized` must have their corresponding parenthesis on the same line.
   Correct: `wait();`
   Incorrect: `wait` < *newline* > `();`
   Incorrect: `wait (` < *newline* > `);`

8. No methods of a class can be overloaded. If there are such methods, they will not be converted correctly.
   Correct: `public void meth(int i)`
   Incorrect: `public void meth(int i)` and `public void meth(double d)`

9. The constructor must not be overloaded. If it is, only one version is converted which may (and probably will) result in code not able to get compiled.
   Correct: `public MyClass()`
   Incorrect: `public MyClass()` and `public MyClass(int i)`

3.2 Parameters of the Converter

The converter itself is not part of any package already mentioned. It consists of one 'executable' file named `SerMain.class` and ten `.insert` files which will be described in detail in chapter 3.4. All `.insert` files must reside in the `insert` subdirectory of the directory where `SerMain.class` is placed. `SerMain.class` cannot be executed directly by the host operating system since it contains *bytecode*[12], not a target environment's machine code. `SerMain` takes one mandatory parameter and two optional ones. The first parameter denotes the main class of the program being converted. The main class is the class that the user gives as parameter to the Java interpreter and whose `main()` method is thus executed automatically after start-up. In J-Serializer, the main class is supposed to create and start a number of threads, doing no other important work. Therefore, it is not converted in the usual way that all other classes are.

The two other parameters denote the input and output directory. If they are not entered, implicit values (subdirectories `input` and `output` of the current directory) are used. If the directory entered is not absolute, it is appended to the current directory.

[12]Bytecode is a platform-independent code generated by Java compilers which needs to interpreted in the target environment.

If no parameters are passed to `SerMain`, then a help message is printed out and the program terminates. Some examples follow:

```
java SerMain MainClass
```

This command tells J-Serializer to convert all files from `./input` to `./output`, using `MainClass` as the main class. Note: '`.`' is the current directory.

```
java SerMain MainA sources result
```

This command tells J-Serializer to convert all files from `./sources` to `./result`, using `MainA` as the main class. Note: '`.`' is the current directory.

```
java SerMain MainB /home/john/dir_a /home/john/dir_b
```

This command tells J-Serializer to convert all files from `/home/john/dir_a` to `/home/john/dir_b`, using `MainB` as the main class.

The format of directories entered by the user is platform-dependent. Under MS-Windows, '\' will be used instead of Unix '/' and a drive letter followed by a colon will be used at the beginning of an absolute path.

3.3 Steps of the Conversion

When the converter is started, it first tries to determine what are the input and output directories. If these parameters are not supplied by the user, implicit values are used. After that, the real conversion process can start.

Since it would be quite complicated to analyze Java source texts, J-Serializer uses another approach. Instead of using `.java` files, it uses their compiled equivalents – `.class` files. A class must be compiled and its bytecode must be put to the input directory together with its source file in order for a class to be converted by J-Serializer. If its bytecode is not present in the input directory, the source file itself is not touched by the converter and the class is not converted.

In the conversion process, a mechanism called *reflection* plays an important role. It is a possibility of a Java program to get information about itself and its parts (classes, their fields, methods, constructors, objects, return types, parameters of methods, etc.) at runtime. Tools being part of the reflection subsystem are located in the `java.lang.reflect` package. The binary image of a class is loaded into memory and converted to an instance of the `Class` class[13]. This conversion is done by a *class loader* – instance of the `FileClassLoader` class. It is necessary to use this non-standard class loader,

[13]A class whose name is `Class`

subclassed from the `ClassLoader` class, since the standard one, present implicitly in every Java program, has certain limitations:

- It does not load classes which are not in the current directory or in the directory pointed to by the `CLASSPATH` environment variable.
- If a class belongs to a package, the standard class loader loads it only if its bytecode is stored in a file located in a subdirectory of the current directory or the `CLASSPATH` directory and the subdirectory's name corresponds to the package's name.

The class loader used in J-Serializer ignores these limitations, i.e. it allows to load any class being part of any package. However, all classes loaded by this class loader must reside in the same directory.

After all classes are successfully loaded, they must be sorted into four categories. As described in chapters 3.5.1, 3.5.2, 3.5.3, and 3.5.4, there are different rules how to handle the conversion of a public thread, of a public non-thread, etc. The four categories are as follows:

- Public Threads – Classes derived (directly or indirectly) from the `Thread` class. The class is public, i.e. it has the `public` modifier in its declaration and it is stored in a source file having the name of the class (+ `.java`).
- Non-public Threads – Classes derived (directly or indirectly) from the `Thread` class. The class is not public, i.e. it has no access modifier in its declaration and it is stored in a source file together with another public class.
- Public Non-threads – Classes derived (directly or indirectly) from `Object`, but not derived from `Thread`. The class is public, i.e. it has the `public` modifier in its declaration and it is stored in a source file having the name of the class (+ `.java`).
- Non-public Non-threads – Classes derived (directly or indirectly) from `Object`, but not derived from `Thread`. The class is not public, i.e. it has no access modifier in its declaration and it is stored in a source file together with another public class.

Note: In the rest of the chapter, non-threads are often referred to as 'normal classes' or just 'classes' while threads and their subclasses are reffered to as 'threads'.

The main class is completely excluded from this sorting, as it is always converted in a special way. Also, interfaces are excluded since they have no real code and thus it is not necessary to convert them. Abstract classes are not excluded because they can contain non-abstract methods which are not overwritten in their subclasses.

After having sorted all loaded classes, the converter takes one after another from each category and converts them according to the algorithm specific to the category. The conversion process stops either if all classes are successully converted or if an error is encoutered during a class' conversion.

3.4 The `.insert` Files

The `.insert` files are text files containing pieces of Java code which are inserted by the converter to well defined places of user's source files. They must reside in the `insert` directory of the directory where `SerMain.class` is put.

This concept brings several advantages when compared to the case when patching information are stored directly in the source code of the converter:

- The patching information for a specific purpose is located at one place only. Therefore, if there is a need to modify it, it is possible to do it without affecting the converter's source code where the same piece of inserted code might be located more than once.
- If a part of the `cz.zcu.fav.kiv.jserializer` package changes, the changes have usually no influence on the converter itself, only on the `.insert` files.

Currently, there are 10 different `.insert` files. Their purpose and content are explained in the following paragraphs.

1. `all_all_finished_notify.insert`

   The code of this file replaces any call to the `notify()` or `notifyAll()` method of any lock. It stops the timekeeper, calls the `finishedWithoutException()` method of the scheduler and then starts the timekeeper again. `finishedWithoutException()` is used because the original code does not throw any exception.

   ```
   timeKeeper.stop();
   ```

```
scheduler.finishedWithoutException
          (timeKeeper.lastDelta()+1.0);
timeKeeper.start();
```

2. `all_all_finished_wait.insert`

   This file is almost identical with the last one but one important change must be notified. The `finishedWithoutException()` method is replaced with `finishedWithException()` because the original code (a call to `wait()`) may throw `InterruptedException`. If the other version of the method were used, the converter would have to remove all code concerning handling the exception which would be a complicated process. This file replaces any call to the `wait()` method of any lock.

   ```
   timeKeeper.stop();
   scheduler.finishedWithException
             (timeKeeper.lastDelta()+1.0);
   timeKeeper.start();
   ```

3. `all_class.insert`

   The content of this file is inserted at the beginning of all classes' definitions. It assures that every class can access the shared scheduler and the timekeeper.

   ```
   private static AbstractTimeKeeper timeKeeper = null;
   private static Scheduler scheduler = null;
   ```

4. `all_import.insert`

   The content of this file is inserted at the beginning of every converted file. It assures that all classes from the `cz.zcu.fav.kiv.jserializer` package will be available to the converted program.

   ```
   import cz.zcu.fav.kiv.jserializer.*;
   ```

5. `others_all_beginning.insert`

   The content of this file is inserted at the beginning of every method of all non-thread classes (or normal classes). It assures that the scheduler and the timekeeper will always be properly initialized in normal classes' methods.

```
scheduler = Scheduler.getDefaultScheduler();
timeKeeper = StandardJavaTimeKeeper.getSharedTimeKeeper();
```

6. `Thread_class.insert`

   The content of this file is inserted at the beginning of all threads' definitions. It assures that every thread knows its ID number.

   ```
   private int myThreadNo = NO_THREAD;
   ```

7. `Thread_constructor_end.insert`

   This file contains code which is inserted at the end of all threads' constructors. It initializes the scheduler and the timekeeper once per an instance of a thread class.

   ```
   scheduler = Scheduler.getDefaultScheduler();
   timeKeeper = StandardJavaTimeKeeper.getSharedTimeKeeper();
   ```

8. `Thread_run_beginning.insert`

   This file is inserted at the beginning of the `run()` method of every thread class. It registers every new thread (by a call to `registerTask()`) and calls the `firstStop()` method which synchronizes all running threads at the beginning of a program's execution. When a thread is selected to run and it leaves the `firstStop()` method, the timekeeper is started.

   ```
   myThreadNo = scheduler.registerTask(this);
   scheduler.firstStop(myThreadNo);
   timeKeeper.start();
   ```

9. `Thread_run_end.insert`

   This file is inserted at the end of the `run()` method of every thread class. It deregisters every thread which is about to finish and switches to another one.

   ```
   scheduler.deregisterTask();
   ```

3.5 Conversion of a Class

As described in chapter 3.3, the conversion of a class is one of many steps J-Serializer does when it is used. After all classes are sorted into the four categories, they are taken one by one and converted separately. The conversion of one class is what this chapter puts focus on.

The mechanism used for conversion of a class is depicted in figure 7 on page 32. Data inputs, outputs, and tools used are represented by rectangles, actions by bubbles.

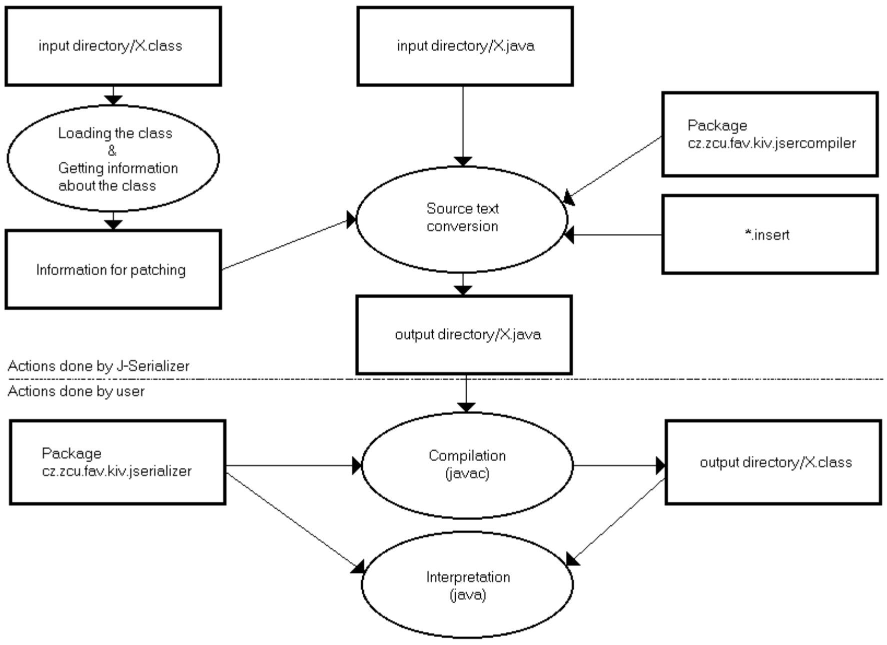

Figure 7: The Conversion Mechanism

First of all, a `.class` file is taken from the input directory and loaded into memory. At this moment, the source file (`.java`) is not touched at all. Loading the `.class` file first assures that only classes being properly written and compiled without problems will be converted. When binary data are loaded from the file, the class loader (instance of `FileClassLoader`) converts them to an instance of `Class`. The `Class` class is a part of the Java reflection package, as mentioned on page 27. Using various methods of the `Class` class, the following information is retrieved:

- Information about *class modifiers.* This includes access modifiers, but also the `abstract` and `interface` modifiers. The method `getModifiers()` is used for this purpose.

- The name of the class' *source file*. Since there is not a method to return the source file's name, the following approach is used: If the class has the `public` modifier, its source file's name is identical to the class' name, plus '`.java`'. If the `public` modifier is not set, `javap`[14] is executed as a separate process and its output is redirected to an `InputStreamReader`. The first line of `javap`'s output contains the name of the source file.

- Information about all class' *constructors* and *methods*. Two arrays (one array of type `Constructor` and one array of type `Method`) are returned. `getDeclaredMethods()` is used to get the array of methods and `getDeclaredConstructors()` is used to get the array of constructors. In the case of normal classes, the information about constructors is not needed to convert the class, therefore `getDeclaredConstructors()` is not called.

When all necessary information is available, the real conversion process can start. From the input directory, the class' source file is taken, modified in several steps and written to the output directory. When the output file is written, the class is supposed to be properly converted and the converter can pick up another class for conversion.

As shown in figure 8 on page 34, several stages can be distinguished during the conversion of a source file. They are as follows:

1. Comments are removed from the source file. This allows the converter to work more quickly and precisely.

2. Braces (characters '{' and '}'), which denote beginning and end of blocks of code in Java programs, are put on separate lines. At most one brace is placed on a line and nothing than the brace itself. This is due to the source code parser which must be able to find beginnings and ends of blocks easily.

3. Spaces are inserted between consequent tokens of the code. Spaces are inserted between any two consequent identifiers, operators or separators, only square brackets '`[`' and '`]`' are left stuck together. Again, this is due to the source code parser which must be able to find sequences of strings in the source code.

[14]Java decompiler, distributed as a standard part of JDK.

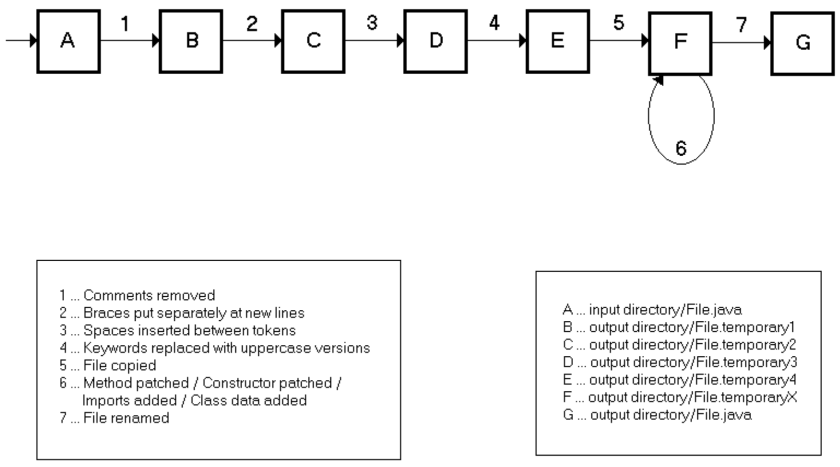
A
B
C
D
E
F
G
1
2
3
4
5
6
7
1 ... Comments removed
2 ... Braces put separately at new lines
3 ... Spaces inserted between tokens
4 ... Keywords replaced with uppercase versions
5 ... File copied
6 ... Method patched / Constructor patched / Imports added / Class data added
7 ... File renamed
A ... input directory/File.java
B ... output directory/File.temporary1
C ... output directory/File.temporary2
D ... output directory/File.temporary3
E ... output directory/File.temporary4
F ... output directory/File.temporaryX
G ... output directory/File.java

Figure 8: Stages of Source File Conversion

4. Some important keywords[15], such as `wait`, `notify`, `notifyAll`, and `synchronized`, are replaced with their uppercase version if they are inside double quotes. This is due to the converter which tries to replace these keywords with an `.insert` file. However, these words may be put in a string (inside double quotes) which means that they must not be replaced.

5. Either a method or a constructor is *patched*, i.e. its beginning if found, the method's or constructor's body is parsed and some important changes are made – a keyword is replaced, a piece of code is added from an `.insert` file etc. This point repeats as needed, i.e. until all methods and constructors of the class are patched. Sometimes, some other changes are made, e.g. an `.insert` file is inserted at the beginning of the file or at the beginning of the class' declaration. The way how the converter behaves at this point is the principal difference between the converter of normal classes and the converter of threads. This point will be described in more detail in subchapters 3.5.1 and 3.5.3.

[15]They are not real keywords of the Java language, they are only called so in this text.

3.5.1 Conversion of a Public Thread

There are many places in a thread's source code where important changes are made. At the very beginning of the source file, all classes of the package `cz.zcu.fav.kiv.jserializer` are imported – the file `all_import.insert` is inserted. New fields are added to the fields already present in the class: the shared scheduler, the shared timekepper (file `all_class.insert`), and thread ID number (file `Thread_class.insert`).

The *constructor* is patched in special way, differently from methods. A code initializing the shared scheduler and timekeeper is added to the end of the constructor – file `Thread_constructor_end.insert`. If there are more than one version of constructor (it is overloaded), only the first one is patched. This is due to the algorithm used for sequence finding. Overloaded constructors (and overloaded methods) will be probably patched correctly in a next version of J-Serializer.

Also the `run()` method is patched in a special way because this method is called automatically by the JVM when a thread is started. At the beginning, the patched version contains code performing registration and synchronization of all started threads which is inserted from `Thread_run_beginning.insert`. At the end, a deregistration routine is inserted from `Thread_run_end.insert`. The rest of the method is patched in exactly the same way that other methods are.

An *ordinary method* is probably the most complicated part the converter must deal with. In threads, nothing is added to the original code, neither to the beginning nor to the end. But quite many things have to be replaced or deleted:

1. If the `synchronized` keyword is present in the method's header (as modifier), it must be removed from there.

2. The `synchronized` keyword must also be removed from the method's body, wherever it is present. The parenthesis surrounding the mandatory lock must be removed too. However, the opening and closing braces of the block belonging to the `synchronized` statement must be preserved because object declaration can take place in any block of code, possibly hiding an object of the same name declared in an outer block.

3. A call to the `wait()` method of any object must be replaced with the content of `all_all_finished_wait.insert` which performs switching

from a thread to another one. The version of `finished()` used in this `.insert` file has `InterruptedException` declared in its header to be possibly thrown out.

4. A call to the `notify()` or `notifyAll()` methods of any object must be replaced with the content of `all_all_finished_notify.insert` which performs switching from a thread to another one. No exception is declared to be thrown out from the version of `finished()` used in this `.insert` file.

3.5.2 Conversion of a Non-public Thread

In general, non-public threads should be converted in exactly the same way as public threads. But one important difference must be remarked. Non-public threads share their source file (`.java`) together with at least one other class, a thread or a normal class. This might confuse the converter and therefore it is necessary to cut the corresponding part out of the source file, convert it aside and paste it back to the source file.

This feature is not implemented yet in J-Serializer and therefore non-public threads are not converted. They are left unchanged in the original state and a warning message is printed out to the console:

```
Converting non-public thread NonPubT...
Non-public threads are not supported
in this version of J-Serializer.
```

3.5.3 Conversion of a Public Class

A public normal class is a slightly easier job for the converter than a public thread. The `cz.zcu.fav.kiv.jserializer` package is also inserted from `all_import.insert`. A little difference is that only the shared scheduler and timekeeper are added to the class' fields, nothing else. Therefore, only the file `all_class.insert` is inserted.

It is not necessary to patch the constructors, if there are any. They are all preserved in the original state. Methods are patched in almost the same way as methods of public threads (see chapter 3.5.1), with one important difference: to the beginning of every method, the content of `others_all_beginning.insert` is added, performing initialization of the shared scheduler and timekeeper. For reasons of doing this, see chapter 2.2.

The replacements and deletions inside a method are identical to the actions described in 3.5.1.

3.5.4 Conversion of a Non-public Class

Everything what was written about non-public threads in 3.5.2 applies to non-public classes. Therefore, non-public classes are not converted and the following text appears instead of the conversion:

```
Converting non-public class NonPubC...
Non-public classes are not supported
in this version of J-Serializer.
```

3.5.5 Conversion of the Main Class

The main class is converted in a slightly different manner that classes and threads already mentioned. In fact, it can be considered as a thread, created and started automatically by the JVM, but threads' `run()` method is replaced with `main()`. Therefore, the initial registration routine must be used at the beginning of `run()`, as well as the deregistration routine at the end.

Since the main class has usually no constructor, the initialization of shared objects must also be placed in `main()`, before the first call to a method of the scheduler. Even if the main class had a constructor, the method of thread patching would not be usable. The `main()` method must always be marked as static which means that it can be invoked even if no instance of the class exists. Therefore, no instance is created and, consequently, no constructor is invoked.

Usually, the main class only creates and starts up other threads, then dies. This behavior does not affect the principles of J-Serializer at all since threads created in `main()` can be properly synchronized even after the main thread dies. If the main class remained not converted, the behavior would not differ much unless the `main()` method calls a shared monitor's methods.

3.5.6 Conversion of Nested Classes

Since nested classes share the source file together with their top-level class, the same problems appear as in the case of non-public classes. (See 3.5.2 and 3.5.4 for more details.) J-Serializer has not been tested yet with nested classes but it is very likely that the conversion process would fail if a nested class had to be converted.

3.6 Hierarchy of Convertors

As mentioned in chapter 3.5, different entities (threads, normal classes, . . .) of a Java program are converted in different ways. In J-Serializer, every entity has its corresponding converter which is responsible its conversion. Since all converters do some elementary actions (such as source text preprocessing) in the same manner, it is advantageous to build up a hierarchy of converter classes, having as its root the most general converter which implements all common actions. The hierarchy is shown in figure 9.

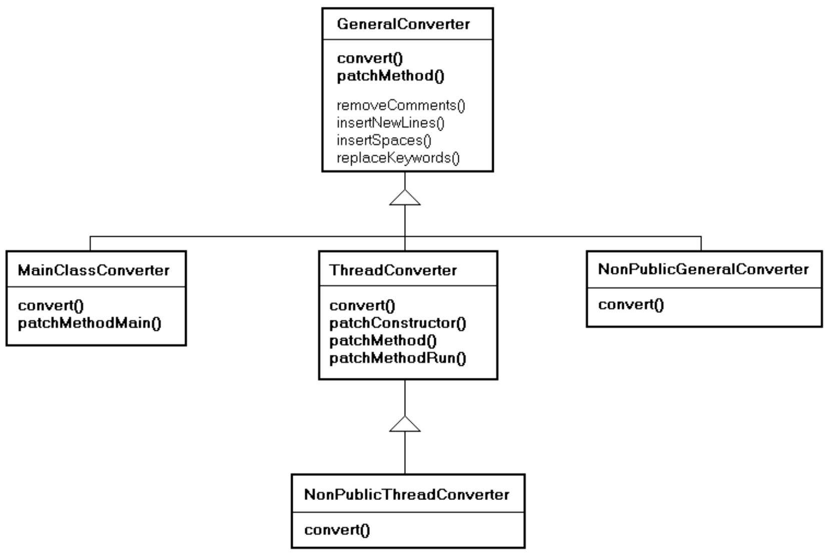

Figure 9: Hierarchy of Converters

The root of the hierarchy is the `GeneralConverter` class which is responsible for conversion of 'normal' classes. It introduceses some auxiliary methods, used for source text preprocessing – `removeComments()`, `insertNewLines()`, `insertSpaces()`, `replaceKeywords()` – and for basic text manipulation. Then, it introduces the `patchMethod()` method which implements the algorithm used to convert a method of a class. More details can be found in chapter 3.5.3. The `convert()` method controls the conversion process of one class. All converters derived from `GeneralConverter` overwrite this method. Usually, there is a need to exclude a particular method from the normal conversion process and convert it in a special manner – for `MainClassConverter`, it is the `main()` method, for `ThreadConverter`, it is the `run()` method. Converters of non-public classes remove all useful actions from `convert()`, they just print out an info message.

The `MainClassConverter` class adds the `patchMethodMain()` method, responsible for conversion of the main class' method `main()`. (See 3.5.5.) Thread converter adds `patchConstructor()` for conversion of threads' constructors, `patchMethodRun()` for conversion of threads' `run()` method and it overwrites the `patchMethod()` method for proper conversion of other methods. (See 3.5.1.)

4 Conclusions and Future Work

In this paper, the initial version of J-Serializer, Java programs serializer, was presented. First, the structure of a typical Java concurrent program was shown and its model was suggested. The mechanisms internally used by J-Serializer were also shown, as well as the process of source texts conversion.

This version is not (and cannot be) a reliable, 100-percent functioning program. It has many limitations and not-implemented-yet features which will (hopefully) be removed during further research. Despite these problems, it has been proven that J-Serializer is built up upon good design decisions and that the objectives fixed at the beginning of the research are not unreal.

J-Serializer is available at `http://home.zcu.cz/~jkacer/jserializer`.

In future, an exact mathematical model (only partially defined in chapter 1) will have to be constructed. According to the model, the tool will have to be modified if it is shown that it does not meet all needs of the complete model.

A better implementation of string sequences searching will be necessary in order to properly convert classes having overloaded constructors or methods. Also, convertors of non-public classes and threads will have to be fully implemented.

References

[Lea] *Lea, D.*: **Concurrent Programming in Java – Design Principles and Patterns**, Second Edition.
Addison-Wesley, U.S.A., November 2000, ISBN 0-201-31009-0

[MaKr] *Magee, J. + Kramer, J.*: **Concurrency – State Models and Java Programs**. John Wiley & Sons, U.S.A., 1999, ISBN 0-471-98710-7

[Her] *Herout, P.*: **Učebnice jazyka Java**.
Kopp, Czech Republic, June 2000, ISBN 80-7232-115-3

[Java] *Sun*: **Java Home Page**
`java.sun.com`

[Mag] *MageLang Institute*: **Writing Your Own Class Loader**
`developer.java.sun.com/developer/onlineTraining/-`
`Security/Fundamentals/contents.html`

[JLS] *Sun*: **The Java Language Specification**
`java.sun.com/docs/books/jls/second_edition/html/-`
`j.title.doc.html`