Java™ & Real-Time
École IN2P3 d'informatique: Temps Réel

Christos Kloukinas

Verimag
http://www-verimag.imag.fr
Centre Équation
2, av. de Vignate
38610 GIÈRES
Telephone/Fax: +33 (0) 4 76 63 48 32 / 50
E-mail: Christos.Kloukinas@imag.fr

29 May 2003

Abstract

Java is a language with a lot of merits, whose use for Real-Time systems could help in increasing their quality.

In this tutorial we will examine its characteristics, its advantages & disadvantages for Real-Time programming, as well as, the RTSJ proposal for rendering Java more suitable for Real-Time programming.

Contents

1 Introduction

Java is a language with a lot of merits and is quickly gaining acceptance by developers. In this tutorial we will examine its characteristics which make it advantageous for Real-Time systems, those which pose a problem, as well as some of the proposals which have been made for better adjusting it to a Real-Time setting.

For the purposes of this tutorial, we use the term Real-Time as referring to *Hard* Real-Time systems, *i.e.*, those where missing a deadline is considered as a system failure. This in contrast to the case of *Soft* Real-Time systems, where missing a deadline can sometimes be an acceptable situation, where the operation of the system is simply degraded (*e.g.*, losses of some image frames in a video conferencing application). Even in the setting of Hard Real-Time systems, we can identify those which are of a large size (*e.g.*, an air traffic control system) or of a small size (*e.g.*, an embedded controller of the breaks of a car). This tutorial has been developed with the second case in mind. However, both these cases share a number of constraints such as a need for determinism, fast execution, an ability to specify the scheduling of tasks in the system, *etc.*

This tutorial starts with a quick introduction to the Java language and to Object-Oriented programming. Then, we examine the existing mechanisms in Java for doing multi-threaded programming, as well as the Java Native Interface. Following this, we identify a number of problems which make it difficult to use Java in a Real-Time setting and look at the different solutions (either existing or proposed) which allow one to program a Real-Time system using (a variant of) Java.

2 What's Java?

Among the characteristics of Java [5] that make it an attractive language are the following: Object-Orientation (O-O), type-safety, security, portability, GUI/Net/concurrency awareness, and automatic garbage collection (GC). Being O-O means that it is easier to structure a program, break up the interfaces and the implementations of the different data structures, *etc.* Java is more type-safe than C/C++. This, in addition to the fact that it imposes checks on array bounds and disallows pointer arithmetic, make it a lot more safe and easier to analyse and to optimise by a compiler. It comes with a large standard library for doing GUI's, network programming and has built-in support for threads and synchronisation on objects shared among different threads. Its automatic garbage collection removes most of the memory-related problems that are so often in programs written using C/C++.

Portability is guaranteed by fixing the size and byte order of the basic data-types, using a common format (bytecode) for Java programs which is interpreted by a Java virtual machine (JVM) [12] and explicitly defining aspects such as floating point operations, the memory model, *etc.*

2.1 Moving from C to Java in 5 Minutes

Java has no `struct` construct. One creates new data types by using the `class` concept. A class has fields just like a structure. Unlike a structure, a class also has methods, *i.e.*, functions which operate on its fields, thus making it clearer which are the available functions on a particular data object. In fact, this is the cornerstone of O-O programming; it allows one to move the specific implementation of a function away from its uses and hides the selection mechanism at the site of each use. This greatly facilitates code development

(since the selection mechanism is done automatically by the O-O language) and the maintenance of libraries (since the O-O language is capable of selecting different/new function definitions). In other words, *O-O is a high level* `if`.

In addition, a class allows one to declare how access to its fields and methods should be performed. One can declare a field (method) as `public` (*i.e.*, accessible by every other class which can access the package of this class), `protected` (*i.e.*, accessible by classes which belong to the same package, or those which inherit from it), or as `private` (*i.e.*, accessible only by methods of the same class). Other field (& method) modifiers are the `static` one, which declares that the field is shared by all object instances of the class (*i.e.*, it is a global variable), and the `final` one, used for preventing sub-classes from overriding or hiding this field. Class methods can additionally be declared as `synchronized` (*i.e.*, the method body is as if it was inside a `synchronized(this)` block), and `strictfp` to indicate that all expressions in the method are *FP-strict*. Quoting from the [12, section 2.18]:

> *Within an FP-strict expression, all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented using single and double formats. Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results; the net effect, roughly speaking, is that a calculation might produce "the correct answer" in situations where exclusive use of the float value set or double value set might result in overflow or underflow.*

In other words, an FP-strict expression uses a `float` (or `double` according to the context) type for all intermediate results, while a non FP-strict expression allows using larger precision intermediate values than `float` (resp. `double`). In Java, classes can have inner classes and implement `interfaces` (a way to get multiple inheritance).

Java has no `union` construct. A union is obtained in an O-O language through polymorphism (see Table 1).

Table 1: Declaring a `union` in Java

C code:	Java code:

`union A {`
` int i;`
` float f;`
`};`
	`abstract class A { abstract public process(); }`
	
	`class intA extends A { public int i; public process() {/* Use the int */};}`
	
	`class floatA extends A { public float f; public process(){/* Use the float */};}`

`struct B {`
`  char c;`
`  union A {`
`  int  i;`
`  float f;`
`  };`
`};`
	`abstract class B { public char c; abstract public process();}`
	
	`class intB extends B { public int i; public process() {/* Use c & i */};}`
	
	`class floatB extends B { public float f; public process() {/* Use c & f */};}`

The `enum` construct is missing as well; one can either use constant fields of a class for each of the enumerated values (which will now be *typed*) or use polymorphism to create a different class for each of them, overriding their method functions accordingly.

Java offers two ways to do *multiple inheritance*: extension of *interfaces* and *composition*. In the latter we declare objects of the classes we wish to inherit from as member fields of the current class and then implement the methods of the current class by explicitly calling the methods of these member fields.

Java also offers certain reflection mechanisms. For example, given an object `o` we can use the method `o.getClass()` to retrieve its class `c`, from which we can retrieve its methods (`c.getMethods()`), its package

(`c.getPackage()`), *etc.* Given a class `c` and an object `o`, we can query the class to see if the object is an instance of it, through `c.isInstance(o)`.

In order to further allow structuring of code and provision of libraries from third parties, Java offers `packages` which are used as a compilation unit. Inside a `package` only one class can be declared as `public`. Through packages we can ensure that class names of different libraries will be different (*e.g.*, `LasVegas.Roulette` *versus* `Russian.Roulette`).

Last but not least, Java *has no object destructors*! This is because, in pure Java programs there is no need for such methods. Java objects are garbage collected automatically and, therefore, we do not have to free the children of an object ourselves. The `finalize()` method which one may assume is Java's name for destructors, is to be used *only* when programming with the Java Native Interface (JNI) [11] and, even then, should be used sparingly (more on this on sub-section 3.4).

3 Concurrent Programming with Java

Java has built-in support for multi-threaded programming. It allows one to create new *threads* and *thread groups* (see `java.lang.Thread` & `ThreadGroup`) so as to easily describe the parallelism inherent in a system. In order to allow mutual exclusive access by different threads when they are accessing shared objects, Java offers built-in support for *monitors* and, in doing so, ensures that objects will be unlocked when a thread exits a critical region, even if it does so abruptly. This greatly simplifies multi-threaded programming with respect to C/C++. The latter languages do not offer any direct support for multi-threaded programming or for mutual exclusion; one is thus forced to use an add-on library which offers these mechanisms (*e.g.*, Posix threads, see [13]). However, simulating monitors with the `lock()` and `unlock()` functions is difficult and error-prone, since the cases where a thread can exit a critical region due to an exception are numerous and difficult to control. By incorporating direct support for monitors this problem disappears, since it is now the language itself which is ensuring the correct behaviour automatically.

Another interesting feature of Java is its support for timers (see `java.util. Timer` and its companion class `java.util.TimerTask`). Timers are used for scheduling tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Finally, Java offers support for calling a function written in some other language (*e.g.*, C, assembly), through the Java Native Interface (JNI). The rest of this section is devoted to examining these mechanisms in more detail.

3.1 Threads in Java

As aforementioned, Java has built-in support for multi-threaded programming. That is, it supports multiple threads, synchronisation on objects shared among different threads, timers, *etc.* Threads in Java have to implement the `interface java.lang.Runnable`, that is, implement the method `void run()`. This method provides the code which will be used as the `main()` function of the thread. Threads have priorities, whose values belong to the integer interval defined by the `MIN_PRIORITY` and `MAX_PRIORITY` static fields of the class `java.lang.Thread`. The default priority of a thread is respectively given by the static field `NORM_PRIORITY`. One should note that these values are *not predefined* and depend on your particular implementation. In addition, the standard does not guarantee that these priorities will be honoured by the scheduler. As stated in [5, section 17.12]:

> *When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.*

In addition, user threads are by definition of lower priority than the GC. That is, even if a thread has as priority `java.lang.Thread.MAX_PRIORITY` it can be stopped by the GC and be forced to wait until the GC finishes, which is an undetermined period of time. This is because the standard does not impose any particular constraint on the implementation of the GC, so an implementation (*i.e.*, a JVM or a compiler)

must be conservative and assume that the particular GC used is not interruptible, to avoid threads accessing the heap when it is in an inconsistent state.

A Java thread can be characterised as a *dæmon* thread, meaning that it will be automatically exited once all other non-dæmon threads exit. One can change a thread into a dæmon thread (and back) using a method of the `java.lang.Thread` class called `setDaemon(boolean on)` and query whether a thread is a dæmon with the method `isDaemon()`.

Finally, one can control the stack size of a thread by using the constructor (tested with JDK 1.4.1):

```
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

The `ThreadGroup` needed can be a new group of threads, or it can be the current one, that is, the `Thread.current_thread().getThreadGroup()`.

Listing 2 shows a small multi-threaded application. Another way to declare the threads would have been to define them as classes implementing the `Runnable` interface, as shown in Listing 3 (listings start at Appendix A which is on page 20). Then, we would have used that class as the `Runnable` target in the aforementioned thread constructor. That means that lines 38–43 of Listing 2 would now have been as shown In Listing 1. Note, however, that since Java lacks the `sizeof()` operator, it is difficult to derive a correct size for the stack. For this reason, the `stackSize` is *not necessarily used!* That is, Java can decrease it, augment it, or simply ignore it altogether! Indeed, since the `stackSize` we have declared is equal to 1 Byte, it is evident that our demand was silently ignored [1].

Listing 1: Using `Runnables` – *(see Listing 3 for the full code)*

```
/* 38 */  ThreadGroup myGroup = Thread.currentThread().getThreadGroup();
/* 39 */  long stackSize = 1L;     // Just 1 Byte...
/* 40 */  Thread tOne = new Thread(myGroup, (Runnable) threadOne,
/* 41 */                           "\tthreadOne:", stackSize);
/* 42 */  Thread tTwo = new Thread(myGroup, (Runnable) threadTwo,
/* 43 */                           "\t\tthreadTwo:", stackSize);
/* 44 */  tOne.setPriority(java.lang.Thread.MAX_PRIORITY - 0);
/* 45 */  tTwo.setPriority(java.lang.Thread.MAX_PRIORITY - 1);
/* 46 */  tOne.start();
/* 47 */  tTwo.start();
```

Other methods of the class `Thread` which are of interest are the `boolean holdsLock(Object)`, `interrupt()`, `join()`, `yield()`, `sleep(/* delay */)`, and `destroy()`. Methods `resume()`, `stop()` and `suspend()` have been deprecated, since they are unsafe [16]. Even though `destroy()` is roughly equivalent to a `suspend()`, it is not deprecated, but it is left for cases where a program is willing to risk a deadlock rather than exit outright. However, it is not currently implemented...

Coming back to thread scheduling, if you try to execute Listing 2 you will get a different output depending on your system. Table 2 shows the output I got when using a SUN workstation (was the same all the times I tried it) and the two different outputs I got when using a Linux machine.

3.1.1 Interrupts in Java

Java allows a thread to be interrupted through the `interrupt()` method. This method sets a *pollable* and *resettable* flag in the target thread and returns. The targeted thread then receives a synchronous exception if it is blocked at an invocation of `wait()`, `sleep()`, or `join()`. If, however, the targeted thread is not blocked in such a call, then all is done is to set its interrupted flag, without throwing an exception. The targeted thread can then check later on whether it has been interrupted through the `isInterrupted` method (see Listing 4).

3.2 Monitors in Java

A monitor for the shared object `obj` is declared in a Java program through the language construct:

[1] Unless Java can execute a thread with such a small stack...

Table 2: Thread scheduling

SUNSparc Solaris

```
main: Finished initialisation - my priority is 5
        threadOne: Started
        threadOne: my priority is 10
        threadOne: My parent is main:
                threadTwo: Started
                threadTwo: my priority is 9
                threadTwo: My parent is main:
```

Intel Linux

output (a)

```
        threadOne: Started
                threadTwo: Started
main: Finished initialisation - my priority is 5
        threadOne: my priority is 10
        threadOne: My parent is main:
                threadTwo: my priority is 9
                threadTwo: My parent is main:
```

output (b)

```
        threadOne: Started
main: Finished initialisation - my priority is 5
        threadOne: my priority is 10
        threadOne: My parent is main:
                threadTwo: Started
                threadTwo: my priority is 9
                threadTwo: My parent is main:
```

```
synchronized(obj) { /* critical region code */ }
```

The opening brace corresponds to the entering of the monitor and the closing brace to exiting it. Java ensures that even if we exit abnormally this synchronised block due to an exception, a return or a jump (*i.e.*, a `break`, a `continue`, or a `throw`) the monitor will be exited correctly, properly unlocking the object `obj`.

Cooperation of threads is also supported in Java through the method `wait()` & its duals `notify()` and `notifyAll()`, which are inherited from the `java.lang.Object` class by all Java objects. The `wait()` method causes the calling thread to block until it receives a notification. When blocking on a `wait()` method a thread exits the monitor, allowing thus other threads to access the shared object. When a thread is notified and returns from the call to `wait()`, then it automatically re-enters the monitor which it occupied at the moment when it called the `wait()`. A thread can perform a notification using the `notify()` and `notifyAll()` methods; the first in order to notify one among the waiting threads (selected in an unspecified manner), while the second in order to notify all of the waiting threads. If no thread is waiting when a notify occurs then the notification is lost and if there are more than one threads waiting for a notification then the choice of the waiting thread to be notified is performed in an *unspecified manner*, as it better suits the current implementation. In the case of a `notifyAll()`, even though all waiting threads will be notified, only one of them can return from the call to `wait()`, since for doing so it also needs to re-enter the monitor. There are also two versions of the `wait()` method which accept as parameter a period of time the thread is willing to wait for a notification, after which, the wait times-out. The first version accepts a `long` argument denoting the relative time-out in milliseconds, while the second version accepts (in addition to the milliseconds) a second `int` argument which denotes the additional nanoseconds that the thread is willing to wait.

We've already seen how the choice of the thread to be notified is *unspecified* and, thus, *unrelated to its priority*. Indeed, the standard does not mention anything about multiple wait queues or that these should be sorted according to the priorities of the threads which are waiting (see [5, section 17.14]). It does not mention anything concerning the *priority inversion* problem either and does not demand any specific locking

protocol.

Finally, it should be mentioned that locking an object is *not similar to using a mutex*. Indeed, in [5, section 17.5] it is noted:

> *moreover, a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of unlock operations have been performed.*

SUN's implementation of Java implements this using a mutex *and* a counter but another implementation could do it differently.

Listing 2 shows how one can use monitors, as well as, the `wait()` and `notifyAll()` methods.

3.3 Timers in Java

Java has a class (`java.util.Timer`) for defining timers, that is, it provides a facility for threads to schedule tasks (`java.util.TimerTask`) for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals. Timers *do not offer real-time guarantees*: they schedule tasks using the `Object.wait(long)` method. In fact, if a timer task takes excessive time to complete, it can delay the execution of subsequent tasks. These subsequent tasks may then end up executing in rapid succession when (and if) the offending task finally completes. One should also note that, by default, the task execution thread *does not run as a dæmon thread*, so it is capable of keeping an application from terminating. Listing 5 shows a small example using timers.

The periodic timer tasks can be scheduled in two different ways, either according to a *fixed-delay* execution, or according to a *fixed-rate* one. In the former case, each execution of a timer task starts at a time instance which is `period` milliseconds after the end of the previous execution of this timer task. That is, if a timer task gets delayed for any reason, then the subsequent executions of it will be delayed by the same duration. Thus, the timer may drift in time but all executions will be `period` milliseconds apart. In the latter case, the `period` is relative to the *first* execution, and therefore if a timer task execution gets delayed, then subsequent timer task executions may occur in rapid succession, in an attempt to keep the frequency of execution as close as possible to the reciprocal of the period.

Finally, we have the possibility to `cancel()` a timer task or a timer. If we cancel a timer, then all the timer tasks which were scheduled with it will be cancelled (a timer is reusable by different timer tasks).

3.4 Java Native Interface

Another option we have in Java is to use JNI [11] in order to call functions implemented in another language. To show how one can use JNI we will use an example which prints the thread's data and a welcoming message. Let us assume that we have the Java program shown in Listing 6 (on page 24). There, we can see that the class `HelloWorld` has a `static` block which loads at start-up the library "HelloWorld". Using JNI, we will provide this library ourselves by writing it in C (see Listing 7).

Finally, we put all the parts together, doing the following (on a Linux machine using the Bourne shell (`sh`), where Java has been installed at `/usr/local/soft/jdk/1.4.1`) [2]:

```
$ javac hello-jni.java
$ javah -jni HelloWorld
$ gcc -shared -o libHelloWorld.so hello-jni.c \
>     -I/usr/local/soft/jdk/1.4.1/include \
>     -I/usr/local/soft/jdk/1.4.1/include/linux
$ java -Djava.library.path=. HelloWorld
Starting Java_HelloWorld_print
Thread "Thread[The Main from Java,5,main]" says using JNI: Hello World!
$
```

Note: To find the type of a field or method of a class, *e.g.*, of `java.lang.Thread`, you can use:

```
javap -s java.lang.Thread
```

[2] Java version (`java -version`) is `1.4.1 HotSpot Client VM (build 1.4.1-b21, mixed mode)`, GCC version (`gcc --version`) is `3.2` and Debian kernel's release & machine (`uname -rm`) is `2.4.18-bf2.4 i686`.

In the notation used by JNI, basic types are denoted as *(i)* `V`, *(ii)* `Z`, *(iii)* `C`, *(iv)* `B`, *(v)* `S`, *(vi)* `I`, *(vii)* `J`, *(viii)* `F` and *(ix)* `D` for *(i)* `void`, *(ii)* `boolean`, *(iii)* `char`, *(iv)* `byte`, *(v)* `short`, *(vi)* `int`, *(vii)* `long`, *(viii)* `float` and *(ix)* `double` respectively. An object of a class X is denoted `L`X`;` and an array is denoted by a leading square bracket. Arguments of a method are placed inside parenthesis and the return value follows after the parenthesis. So, if we had the method

```
java.lang.String[] foo(java.lang.Thread[] array,
                       java.lang.String str,
                       int i,
                       long j)
```

then the type of this method would be:

```
([Ljava/lang/Thread;Ljava/lang/String;IJ)[Ljava/lang/String;
```

Through JNI we can enter or exit monitors, allocate memory using Java's `new`, *etc.* So, it might seem that this is a good way to bypass all of Java's shortcomings with respect to Real-Time programming, by implementing the missing mechanisms using JNI. However, this is not true. First of all, JNI is a rather heavy and slow mechanism, exactly because of its generality and wish for portability. Queries for methods, fields and classes are performed using strings as arguments and traversing the internal data structures of the JVM in order to locate the corresponding object. This makes the queries themselves portable, since no internal data structures need to be exported, but the execution time needed to perform such a query increases drastically. In addition, it becomes impossible for an optimising compiler to analyse the native code for Java classes and methods used, so as to be able to remove the unused ones. One might be better served by a less portable but faster native interface, such as the *Cygnus Native Interface* (CNI) [3] which is offered by GCJ [4], the GNU Java front-end to the GCC compiler. The GCJ compiler considers all Java classes as C++ classes (but not *vice versa*) and thus allows one to write code in C++ which is using directly the Java classes. Listing 8 shows the corresponding C++ implementation of the native call using CNI. The commands to use in this case are the following:

```
$ gcj -C hello-jni.java
$ gcjh HelloWorld
$ gcjh -stubs HelloWorld
$ gcc -shared -o libHelloWorld.so HelloWorld.cc -I`pwd`
$ gcj -o HelloWorld --main=HelloWorld hello-jni.java \
>     -L`pwd` -Wl,-R`pwd` -lHelloWorld
$ ./HelloWorld
Starting Java_HelloWorld_print
Thread "Thread[The Main from Java,5,main]" says using CNI: Hello World!
$
```

However, there is another more basic reason for which JNI is difficult to use for doing Real-Time thread manipulations. This is the fact that Java threads are not necessarily implemented using the native threads of your system. If they're indeed implemented using native threads, then it might be easier to manipulate them in a Real-Time system. If, however, they're not (*e.g.*, implemented using `setjmp`/`longjmp` as in the *Green* threads in SUN's Java implementation) then it's anything but obvious...

Finally, we must note that when using JNI and the native function we call allocates dynamic memory, then we may need to use the `finalize()` method. Its purpose is to allow us to manually deallocate this memory when the Java object will be garbage collected. Java cannot do so automatically, since this memory was not allocated through its mechanisms and is therefore unknown and unreachable by the GC. Take note of this part of the Java Language Specification [5, section 12.6]:

> *The Java programming language does not specify how soon a finalizer will be invoked, except to say that it will happen before the storage for the object is reused. Also, the language does not specify which thread will invoke the finalizer for any given object. It is guaranteed, however, that the thread that invokes the finalizer will not be holding any user-visible synchronization locks when the finalizer is invoked. If an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.*
>
> *The finalize method declared in class Object takes no action.*
>
> *The fact that class Object declares a finalize method means that the finalize method for any class can always invoke the finalize method for its superclass, which is usually good practice.*

(Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.)

More on JNI can be found at `http://java.sun.com/docs/books/tutorial/native1.1`.

4 Java for Real-Time Programming

Java has lots of merits but it is easy to see that it has certain problems with respect to Real-Time programming. These problems can be summarised as:

- Java is slow;
- Java's memory model [5, chapter 17] is especially convoluted (there's already a new draft specification on this, see [14]);
- The GC is unpredictable and can block you for an undetermined period of time;
- scheduling of threads is unpredictable;
- monitors may cause inversion of priorities and there is no support for multiple wait queues or for queues sorted according to thread priority;
- Java's design for portability implies that it cannot take advantage of your OS's capabilities — Java ignores these capabilities on purpose.

The rest of this section examines these problems in more detail.

4.1 Speeding-up Java

Java has got the reputation of being a slow language; indeed, certain benchmarks showed it to be 20 times slower than C++. However, we should note here that describing a language as slow is misleading. An implementation of a language can be slow but that does not mean that the language itself has this disadvantage. No one would ever accuse C of being a "slow" language, just because a particular implementation of it is slow. Therefore, we must examine the specification itself to see whether there is a specific characteristic/mechanism which is obligatory and can cause all possible implementations of the language to be slow. The specification of Java does not have any such demand *per se*. Indeed, it has been shown that execution speed becomes comparable to that of C++ when compiling Java directly to native code. For this reason, some JVMs use a technique called *Just In Time (JIT)* compiling, which compiles the Java bytecodes of an application to native code during the execution itself. Another approach is to use an *Ahead Of Time (AOT)* compiler which compiles the Java application into native code before its deployment.

Since the type-safety of Java is stronger than that of C++ and Java does not allow pointer arithmetic or casting to incompatible types (*e.g.*, casting a pointer towards an int, or a pointer of a class to a pointer of an unrelated class), it is in fact easier for a compiler to apply aggressive optimisations which are impossible in a C/C++ environment.

4.1.1 To JIT or to AOT? That is the question...

Let us now compare JIT with AOT. JIT allows us to be compatible with the standard specification, since it can do all that an interpreter-based JVM can do. Indeed, a JIT consists of an interpreter-based JVM which under specific conditions (*e.g.*, execution of a portion of code multiple times) decides to compile a part of the application on-the-fly. Its disadvantage is that the compilation itself is a rather heavy task which introduces memory and computation demands by itself. To decrease these demands JIT compilers do not perform any aggressive optimisations. In fact, since they only compile small portions of the code each time, it would be difficult to apply any aggressive optimisations, even if the time & memory needed for doing so could be afforded.

AOT compilers on the other hand can be highly aggressive with respect to optimisations and therefore produce highly efficient code, since they are used before the actual deployment of the application and they have a global view of all the code of the application. For example, this allows them to remove synchronisations when an object is not shared among different threads, which is the most usual case. Another possible optimisation is to remove the checks for out of bound accesses when it can be shown that these can never occur. A critique of the AOT approach is that the size of the produced native code is bigger than the size of the initial bytecodes, since bytecodes are usually more "expressive" than native instructions. However, there is also the opposite opinion, since bytecodes have to include a large set of data structures (*e.g.*, class, object, field, and method names, file and line numbers, *etc.*) whose size cannot be optimised through sharing among class files, but an AOT compiler can easily optimise.

An aspect of Java which could possibly be problematic is its demand for Java applications to be able to load new classes at run-time. Since these classes may be provided in the form of bytecodes, we may be forced to include in the executable a bytecode interpreter, if we desire this functionality and/or wish to abide by the standard.

Finally, there always exist the possibility of using a hardware-based solution through the use of a Java chip, or a co-processor to which the main processor delegates the task of executing the Java bytecodes, or finally as a single hybrid processor which contains hardware logic for processing Java bytecodes as well as native instructions.

4.2 Real-Time GC

Automatic GC as well has long been considered slow, expensive and unpredictable but these characteristics are not shared by all the different GC techniques. Indeed, there are GCs which are predictable, interruptible, fast, with small memory demands, *etc.* Some benchmarks even showed that using a GC is comparable in speed to manually deallocating memory in C (and with the additional safety).

Most of these techniques were developed for systems with a lot of memory, which renders their use on embedded systems questionable. However, there are other ways to obtain automatic GC for embedded systems without sacrificing speed and memory and above all safety. Such a technique consists of *statically* identifying the moment where each object can be garbage collected, so that the corresponding `free`s can be inserted *automatically* by the compiler itself at the appropriate places.

4.3 Real-Time Specification for Java

The *Real-Time Specification for Java (RTSJ)* [2] specifically examines seven areas of Java for Real-Time programming (from the introduction of [2]): *(i) Thread Scheduling and Dispatching*, *(ii) Memory Management*, *(iii) Synchronisation and Resource Sharing*, *(iv) Asynchronous Event Handling*, *(v) Asynchronous Transfer of Control*, *(vi) Asynchronous Thread Termination*, and *(vii) Physical Memory Access*.

The solutions it proposes for these areas were derived using the following guiding principles (again from the introduction of [2]): *(i) Applicability to Particular Java Environments*, *(ii) Backward Compatibility*, *(iii) Write Once (Carefully), Run Anywhere (Conditionally)*, *(iv) Current Practice* versus *Advanced Features*, *(v) Predictable Execution*, *(vi) No Syntactic Extension*, and *(vii) Allow Variation in Implementation Decisions*. Notice that there is a strong demand for reusing the Java application code and, even more, the Java development environments and tools which have been developed so far, as well as, for allowing the specification to be implemented differently, depending on the particular needs of a sub-area of Real-Time systems (large *versus* embedded, hard *versus* soft, *etc.*).

In the following text we examine these areas in more detail. Note that the presentation has been heavily based on Dr. Bollella's slides [1].

4.3.1 RTSJ: Thread Scheduling and Dispatching

Since a single scheduling policy does not suffice to cover all possible needs, the RTSJ has taken the option of demanding one particular policy (`PriorityScheduler`) as always available and at the same time allowing other scheduling policies (*e.g.*, RMA, EDF, *etc.*) to be declared and used. Thus, the RTSJ provides mechanisms for declaring methods to create, manage, admit and terminate Real-Time Java threads.

The required scheduling policy (`PriorityScheduler`) is a *fixed priority, preemptive scheduler*, where the priorities are *provided by the application*, and it offers *at least 28 unique priority levels*. The demand for only 28 unique priorities is a compromise, since a Real-Time Java application may execute along with other non-Java applications which may need to use priorities above, below, or both of those of the Real-Time Java application for system activities.

A scheduling policy affects an object which implements the `Schedulable` interface, that is, one whose base class is `RealtimeThread`, `NoHeapRealtimeThread`, or `AsyncEventHandler`. Objects belonging to classes which implement the `Schedulable` interface need have a reference to a `Scheduler` object, which governs how the `Schedulable` objects should be scheduled.

A `RealtimeThread extends` the `Thread` class and is managed by a scheduler. It may use the heap or other sections of the memory, participate in asynchronous transfer of control and thread termination and directly access the physical memory.

A `NoHeapRealtimeThread extends` the `RealtimeThread` class and it is not allowed to read or write on the heap, or even to manipulate references to objects on the heap. Thus, it must be created within a particular scoped memory area. Thanks to these constraints, a `NoHeapRealtimeThread` can preempt the GC, since it does not interfere with it.

Finally, a `AsyncEventHandler` is used for handling asynchronous events and at reception of such an event executes its `run()` method on a separate thread.

A `Scheduler` object is created with a number of parameters which define *(i)* the eligibility metric to use for deciding what should be the executing thread (this is the scheduling policy itself), *(ii)* the priority parameters, which can be either *traditional* priorities or *importance* priorities to be used in case of overload, *(iii)* the release parameters (`PeriodicParameters`, `AperiodicParameters`, `SporadicParameters`), *(iv)* the memory parameters for the memory demands of the `Schedulable` object, and, *(v)* the parameters concerning this processing group, which allows us to manage many aperiodic or sporadic threads as a (meta-level) periodic thread.

The *importance* priorities can be used in a setting where more than one threads share the same traditional priority. This can easily be the case if we are using RMA to schedule threads, since threads with equal periods will be assigned the same priority under such a scheduling policy. Importance priorities can therefore be used to define which among these threads must execute when the system is in an overload situation and cannot honour all the demands of this traditional priority group.

The release parameters contain a `RelativeTime cost` which identifies the maximum processing time per period/minimum inter-arrival interval that the schedulable object is allowed to use. They also contain an `AsyncEventHandler overrunHandler` which is invoked if an execution of the schedulable object exceeds `cost`. However, on implementations which cannot measure execution time, the `cost` value is used only as a hint to the feasibility algorithm and the `overrunHandler` is not invoked if the execution time exceeds `cost`. One also uses the release parameters to declare the `RelativeTime deadline` associated with the object and a `AsyncEventHandler missHandler` which should be invoked if the `deadline` is exceeded.

The constructors of these classes are shown in Table 3. Table 4 shows how one creates a new periodic Real-Time thread.

4.3.2 RTSJ: Memory Management

In order to allow for deterministic Real-Time GC, the RTSJ introduced *memory scopes*. These define a tree of heap memory regions which are used to give bounds to the lifetime of objects allocated within each region. Objects residing in a memory scope are not managed by the GC. Instead, when a syntactic scope is exited, then the memory scope associated with it *may* be immediately collected, freeing the objects within it *en masse* (it's like allocating an object on the execution stack). Note that we say *may* and not *must*, because the standard simply demands that the objects of a scope should be collected (and their finalizers executed) *before* the scope is to be re-entered, so this can be done anytime between the moment when a scope is exited and the moment we re-enter it (*if we ever do so...*). So, again, *avoid finalizers* and *do not depend on them ever being executed*! Instead, try using the `finally` clause of a `try {} catch () {} finally {}` construct to clean-up.

Objects which will survive a particular scope (*e.g.*, a return value) must be allocated at least as high as on the highest scope which contains another object that can refer to them. However, one may allocate

Table 3: Scheduling constructors

```
setScheduler(Scheduler             sched,
             SchedulingParameters  sched_params,
             ReleaseParameters     release_params,
             MemoryParameters      mem_params,
             ProcessingGroupParameters group)

PriorityScheduler()

PriorityParameters(int priority)

ImportanceParameters(int priority, int importance)

PeriodicParameters(HighResolutionTime start, /* null = start immediately */
                   RelativeTime     period,
                   RelativeTime     cost,
                   RelativeTime     deadline, /* if null, use period */
                   AsyncEventHandler overrunHandler,
                   AsyncEventHandler missHandler)

AperiodicParameters(RelativeTime  cost,
                    RelativeTime    deadline,
                    AsyncEventHandler overrunHandler,
                    AsyncEventHandler missHandler)

SporadicParameters(RelativeTime  minInterarrival,
                   RelativeTime    cost,
                   RelativeTime    deadline,
                   AsyncEventHandler overrunHandler,
                   AsyncEventHandler missHandler)

MemoryParameters(long maxMemoryArea, /* or MemoryParameters.NO_MAX (units=bytes) */
                 long maxImmortal,  /* or NO_MAX (units=bytes) */
                 long allocationRate) /* or NO_MAX (units=bytes/sec) */

ProcessingGroupParameters(HighResolutionTime start,
                          RelativeTime     period,
                          RelativeTime     cost,
                          RelativeTime     deadline,
                          AsyncEventHandler overrunHandler,
                          AsyncEventHandler missHandler)
```

them even higher, in order to optimise the use of the heap towards same-sized scopes, fewer scopes, *etc.* This shows that exiting a scope can be problematic if there are still objects in it which are referenced. For this reason, scopes cannot be exited *explicitly* (*i.e.*, through some `scope.exit()` method) but only *implicitly*. That is, each scope is associated with some `Runnable` object (or a `RealtimeThread`) and is exited only when its `Runnable` object exits. This ensures that there will be no more references to the objects allocated within the scope, with the disadvantage that for each scope we wish to create we have to construct a new `Runnable`.

The top-most memory scope called the *Immortal* memory area is preallocated when the JVM starts and is never deallocated. It is used for sharing objects among the Real-Time threads, as well as, among the Real-Time threads and the non-Real-Time threads.

Scopes are divided into `LTMemory` ones, where the execution time of a `new` is *linear* with respect to the size of the object allocated (and *NOT* with respect to its constructor!), and into `VTMemory` ones, where the execution time of a `new` is *variable*. Therefore, inside a `VTMemory` region it is possible to have a local GC collecting the unused objects of the region and a `VTMemory` can ask the underlying system for more memory when there is not enough memory to perform a `new`. A `LTMemory`, on the other side, does not offer such capabilities, since these cannot be guaranteed to run in linear time (with respect to object size).

Table 4: Creating a periodic Real-Time thread

```
Scheduler sched = javax.realtime.Scheduler.getDefaultScheduler();
ImportanceParameters prio = new ImportanceParameters(MAX_PRIORITY, 3);
PeriodicParameters pp = new PeriodicParameters(
  new RelativeTime(0,0), // start it when it is released
  new RelativeTime(100, 10),// period
  new RelativeTime(30, 0), // cost
  new RelativeTime(60, 0), // deadline
  null,                  // no Overrun Handler
  null);                 // no Miss Period Handler
MemoryParameters mp = new MemoryParameters(MemoryParameters.NO_MAX,
                                    MemoryParameters.NO_MAX,
                                    MemoryParameters.NO_MAX);
MemoryArea ma = new LTMemory(1024, 1024);
ProcessingGroupParameters gp = null;
RealtimeThread rt = new RealtimeThread(prio, pp, mp, ma, gp,
  new Runnable() {
   public void run() {
     RealtimeThread t;
     try {
       t = (RealtimeThread) Thread.currentThread();

       do {
         /* Thread logic */
       } while(t.waitForNextPeriod());
     } catch (ClassCastException e) {}
   }
 });
rt.setScheduler(sched);
if (!rt.getScheduler().isFeasible())
  throw new Exception("Not feasible");
rt.start();     // Release the thread
```

To summarise, there are three kinds of memory with respect to the GC- the GC'ed *heap*, the (unique) *Immortal* memory region (not GC'ed and never deallocated), and the *scoped memory* regions (not GC'ed either but deallocated automatically when its respective syntactic scope is exited). Currently, the following classes (and subclasses) are available: `HeapMemory`, `ImmortalMemory`, `VTMemory`, `LTMemory`, `VTPhysicalMemory`, `LTPhysicalMemory` and `ImmortalPhysicalMemory`. Since the immortal region is always available, objects in it can reference objects on the heap. Since the GC'ed heap appears as an infinite memory where objects

Table 5: Assignment rules

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes, if same or higher scope
Local Variable	Yes	Yes	Yes, if same or higher scope

are never deallocated (we cannot tell if an unreachable object has been collected) objects on the heap can reference objects on the immortal region as well. Objects allocated in a scoped region can also reference objects allocated in the heap or the immortal region. However, objects in the heap or in the immortal region *are not allowed* to reference objects allocated in a scoped region. This is because a scoped region can be exited and therefore deallocated at a moment which is unrelated to the lifetime of the objects in the heap/immortal region. Such accesses are either checked against using static analysis during the compilation of a Java file, or using run-time checks when the code has not been analysed or when we cannot analyse it. These assignment rules for avoiding dangling pointers are summarised in Table 5.

In order to find out the size of a scoped region, one can use the `SizeEstimator` class and its method `void reserve(java.lang.Class c, int no_of_instances_of_c)`. By using additional blocks "`{ ... }`" to break the code of a method into compartments where a certain object reference is used and setting it to `null` at the end of its block, one can give extra hints to an optimising compiler, so that the latter can better analyse the use of objects.

Finally, Real-Time threads can query the maximum preemption latency that the GC may cause them, through `RealtimeSystem.currentGC().getPreemptionLatency()`.

RTSJ: Physical Memory Access Physical memory access is possible through the classes `VTPhysicalMemory`, `LTPhysicalMemory` & `ImmortalPhysicalMemory`, and through one of the classes `RawMemoryAccess` & `RawMemoryFloatAcc`
For example, the `LTPhysicalMemory` class accepts in its constructor the base address of the physical memory block, its size, as well as its type (*i.e.*, `ALIGNED`, `BYTESWAP`, `DMA`, `SHARED`). Since Java lacks a `sizeof` operator, calculating a size for the physical memory (and mapping that to a Java object) is rather cumbersome. One could again use a `SizeEstimator` object to estimate the size needed (from the standpoint of the Java object which will be mapped to the physical memory region).

For this reason, one might be better off using the class `RawMemoryAccess`. Objects of this class model a range of physical memory as a fixed sequence of bytes and offer methods for accessing the contents of the region through offsets from the base address interpreted as `byte`, `short`, `int` or `long` data values or as arrays of these types. Note that a raw memory region cannot contain references to Java objects, since this could be used to defeat Java's type checking.

In addition to the `RawMemoryAccess` class, one can also use the class `RawMemoryFloatAccess` to access a raw memory area by `float` and `double` types or by arrays of these floating point types. However, note that according to the RTSJ an implementation is only required to implement this class if and only if the underlying JVM supports floating point data types.

4.3.3 RTSJ: Synchronisation and Resource Sharing

With respect to synchronisation, the RTSJ demands that conforming implementations enforce by default the *basic priority inheritance* protocol (while also asking for an implementation to provide the *priority ceiling emulation one*). One can set the priority inversion avoidance algorithm to be used by monitors either globally or for a particular monitor. It also demands that threads waiting to enter a synchronised block should be priority queue ordered and if more than one threads have the same priority then these must be queued in a FIFO order.

However, synchronisation among Real-Time threads and normal ones is problematic, since the normal threads may cause delays to the Real-Time ones due to the execution of the GC. In addition, the fact that threads belonging to the `NoHeapRealtimeThread` class have a higher priority than the GC by definition, makes the implementation of a priority inversion avoidance protocol impossible (since the other threads

must have a lower priority than the GC and a priority inversion avoidance protocol can change this). For this reason, the RTSJ provides three classes of *wait-free queues*, which can be used for synchronisation among Real-Time threads and normal ones [3]. The wait-free queues implement a unidirectional data flow, where the Real-Time part (either the `read` or the `write`) is non-blocking and simply returns `null` (resp. `false`) when the operation cannot be performed (*i.e.*, on an empty, resp. full, queue). So, a `WaitFreeWriteQueue` offers the possibility to a Real-Time thread to perform non-blocking `write` operations. If the operation is impossible, the thread itself decides what action must be taken (*e.g.*, overwrite a previous value, ignore the new one, *etc.*). Since no locking is used for synchronisation, there is no need for using a priority inversion avoidance protocol. If more than one Real-Time thread may write to a `WaitFreeWriteQueue` then they must provide their own synchronisation. The side of the `WaitFreeWriteQueue` which corresponds to the non-Real-Time part is synchronised as usually. That is, the `read()` method on a `WaitFreeWriteQueue` is declared as a `synchronized` method and it blocks when the queue is empty.

4.3.4 RTSJ: Asynchronous Event Handling

The RTSJ allows the use of asynchronous event handling mechanisms. To do so, one uses the classes `AsyncEvent`, which corresponds to the asynchronous event we want to handle, and `AsyncEventHandler`, which corresponds to its handler. The latter is semantically equivalent to a Real-Time thread and its method `handleAsyncEvent()` corresponds to the `run()` method of a thread (*i.e.*, it is called when the event happens). Handlers are bound to events using the `addHandler()` method of an `AsyncEvent` object. An `AsyncEvent` object itself is bound to an external event using the method `bindTo(String)`, where the meaning of the string parameter is *implementation dependent*. Using the `fire()` method of an event, we can force a handler to execute right-away. Events are *data-less*, that is, the `fire()` method does not pass any data to the handler(s). A single event may be handled by multiple handlers and a single handler may handle multiple events.

Handlers have scheduling parameters just like the Real-Time threads. In addition, one can use the `BoundAsyncEventHandler` class to obtain handlers which are statically bound to particular threads and thus avoid the extra latency of creating a new thread when an event occurs (at the extra cost of having an extra thread present on the system). Handlers which are created with memory parameters corresponding to a scoped region or to the immortal memory, act like a `NoHeapRealtimeThread` and thus can preempt the GC.

Finally, there's also the `POSIXSignalHandler` class if the underlying system supports Posix signals. One can use this class to bound an asynchronous event handler to a particular Posix signal.

4.3.5 RTSJ: Asynchronous Transfer of Control

Acknowledging the fact that there are cases where things can go really wrong, the RTSJ introduced a mechanism for *asynchronous transfer of control*. The idea behind it is similar to doing a `longjmp` in C/C++. The mechanism implementing it is similar to normal Java exception handling, with the difference that Java exceptions are synchronous. So, for a method to be interruptible by through an exception of type `AsynchronouslyInterruptedException` (AIE), the method must declare this kind of exception in its `throws` clause. If this is not the case, then the asynchronous exception is postponed (set to pending), until the execution enters a method which has declared the possibility of such an exception. In addition, if the execution is inside a `synchronized` block, then the asynchronous exception is again postponed until the synchronised block has been exited. These two rules were set in order to ensure that: *(i)* methods which were written without *a priori* knowledge of a possible interruption will not be interrupted, and *(ii)* that shared objects will not be left in an inconsistent state.

A method which catches an AIE must call its method `happened` to figure out whether the current AIE is the one it expected and propagate it if not. If it is indeed the AIE it expected, then it calls the `doInterruptible` method, passing it as argument an object of a class which implements the interface `Interruptible`, that will provide the handler for the exception.

One can also use the `Timed` class to program AIE's which will be fired at the expiration of a timer. Once the AIE from a timer is handled (*i.e.*, its `doInterruptible` method exits) the timer is *restarted* for the

[3] See the references given at `http://gee.cs.oswego.edu/dl/cpj/s2.4.html`.

amount of time given in the constructor of the `Timed` object, or the amount of time given to the most recent invocation of the method `resetTime`.

AIE's can be `disabled`, `enabled`, `fired`, `propagated`, handled through the `doInterruptible` and queried through the `isEnabled` and `happened` methods.

Another requirement that the RTSJ introduced is that blocking methods in `java.io.*` must be prevented to block indefinitely when they are invoked from a method with an AIE exception in its `throw` clause. In such a case, a conforming implementation can choose among three different possible actions: *(i)* unblock the blocked call, *(ii)* raise an `IOException`, or *(iii)* allow the call to complete normally if the implementation determines that the call would eventually unblock.

RTSJ: Asynchronous Thread Termination Just like the asynchronous transfer of control, sometimes it is needed to asynchronously terminate a thread altogether. However, being able to terminate a thread at any moment is inherently unsafe, so such an action is only allowed through the use of the asynchronous event handling and of the asynchronous transfer of control mechanisms. In other words, we can only terminate a thread which has declared that it can be terminated (*i.e.*, its method `run` declares AIE among the exceptions which can be thrown during its execution), and the thread is not currently executing inside a synchronised block.

Listing 9 shows how one can create periodic `RealtimeThread`, use an `AsyncEventHandler`, search for a particular scheduler and use scoped memory regions.

4.4 Other Real-Time Java Proposals

Here is a list of some other proposals for Real-Time Java. Not all of them are finished specifications, or have (at least) a reference implementation, but they are interesting enough to keep on your mind. The next move on Real-Time Java may well come from them, or something similar, especially with respect to *safety critical systems*.

- RTCE [15] & HIP [6] from the J-Consortium.

  From J-Consortium's web site:

  > *The "High Integrity Profile" [6] is a subset of the "Real-Time Core Extensions" [15] Draft Specification that is suitable for use in high integrity and safety critical applications. The subset provides:*
  >
  > – *Partitioning support for code of different criticality levels*
  > – *Determinism in memory usage, execution speed and functionality*
  > – *Very small footprint for the execution environment*
  > – *Constructs suitable for formal certification to the highest level*

  The RTCE and the HIP are well designed specifications but have a big disadvantage - they introduce new keywords (thus it is impossible to reuse existing tools with them) and demand for a complete, parallel class hierarchy of the base Java classes to be used by the Real-Time threads. The latter makes it difficult to implement a conforming specification, since for every base class one has to implement the base class itself plus its Real-Time counterpart. It also makes development of applications difficult, since one has to keep track of the class hierarchy he is using in his methods and transform arguments from one hierarchy to another (was that a `java.lang.String` or a `realtime.String`?...).

- Ravenscar-Java [9] inspired by the work of the Ada community on Ravenscar [4].

  The idea behind Ravenscar-Java is that by constraining what an application can do, we can construct tools which can automatically analyse and guarantee the safety of the applications and provide us with efficient implementations.

[4] Ravenscar stands for **R**eliable **A**da **V**erifiable **E**xecutive **N**eeded for **S**cheduling **C**ritical **A**pplications in **R**eal-Time.

- JTRON [8] from Japan.

  μITRON is a *de facto* industry standard specification in Japan for Real-Time kernels (eCos [5] provides a certain level of conformance to μITRON and it is open-source). JTRON is the Java API to μITRON and therefore can have a great impact in the future.

- Java 2 Platform, Micro Edition - J2ME.

  Even though J2ME was created for embedded devices and not for Real-Time *per se*, it has a lot of interesting characteristics. It is modular, permitting one to choose a particular profile among the ones possible, and is designed for obtaining small footprint applications which will be running on systems with limited resources.

For even more pointers to other Real-Time Java proposals, see Dr. Valérie Issarny's slides [7].

5 Conclusions

Why use Java?
Because it's:

- easier to write well structured programs in and easier to get them right, since it has built-in support for monitors and threads (therefore, it's easier to maintain your code and have it validated);
- rich on libraries;
- strongly-typed;
- impossible to corrupt the memory because of an out-of-bound array access;
- offered with automatic GC- no more memory leaks or corrupted heap due to using a `free`'ed heap object;
- easier to have a highly optimising compiler, and, *last but not least*,
- so *buzzword-compliant* your boss would probably *force* you to use it in the very near future...;-)

Yes, but Java for Real-Time programming?

- NASA used Lisp for the Deep-Space probe...;
- others are already using Java for Real-Time systems (see GCJ's page);
- new versions are out which make it easier to use with Real-Time systems;
- more and more tools are getting developed each minute;
- doing it with C/C++ is tedious, error-prone, difficult to maintain and to have it validated — Java might help diminish the pain & help automate most of what is needed to be done...

Don't forget: embedded Real-Time systems are becoming a huge market and the software in them is replacing mechanisms which, until now, were implemented in hardware. The resulting complexity (both of development and of maintainability) needs further support from tools to be tamed and this is the hope and driving force behind Real-Time Java.

[5] `http://sources.redhat.com/ecos`

Bibliography

[1] Gregory Bollella. The Real-Time Specification for Java™. Slides available at `http://www.opengroup.org/rtforum/info/oct2000/slides/rtsj.pdf`, October 2000.

[2] Gregory Bollella, Benjamin Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull, Rudy Belliardi, Doug Locke, Scott Robbins, Pratik Solanki, and Dionisio de Niz. *The Real-Time Specification for Java™*. The Java™ Series. Addison-Wesley, 2000. Also available on-line from `http://www.rtj.org/rtsj-V1.0.pdf`. The Reference Implementation (RI) and the Technology Compatibility Kit (TCK) are available from TimeSys `http://www.timesys.com`.

[3] The Cygnus Native Interface for C++/Java Integration. Web page: `http://gcc.gnu.org/java/papers/cni/t1.html`.

[4] GCJ: The GNU Compiler for the Java™ Programming Language. Web page: `http://gcc.gnu.org/java/`.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification Second Edition*. The Java™ Series. Addison-Wesley, 2000. Also available on-line from `http://java.sun.com/docs/books/jls`.

[6] High Integrity Profile Task Group within WG1, RTJWG. High Integrity Profile. Technical report, J-Consortium, 2002. The draft specification is available to members only at `http://www.j-consortium.org/hip`. However, there is a presentation there which is publicly available.

[7] Valérie Issarny. Java et les systèmes temps réel. Available from `http://info.in2p3.fr/page/formation/infoG/DOCUMENTS%20TR-2002/JavaTR.pdf`, October 2002.

[8] ITRON Committee. Yukikazu Nakamoto and Kazutoshi Usui (editors). JTRON2.0 SPECIFICATION. TRON Association, Katsuta Building 5F, 3-39, Mita 1-chome, Minato-ku, Tokyo 108-0073, Japan, September 1999. Available from `http://www.assoc.tron.org/spec/jtron/jtron-200e.pdf` [6]

[9] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A High Integrity Profile for Real-Time Java. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 131–140, November 2002. Also available from `ftp://ftp.cs.york.ac.uk/papers/rtspapers/R:J.:Kwon:2002.pdf`.

[10] Doug Lea. The JSR-133 Cookbook. Web page available at `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`, February 2003 (last visited). Discusses what the Java memory model means for compiler writers.

[11] Sheng Liang. *Java™ Native Interface: Programmer's Guide and Specification*. The Java™ Series. Addison-Wesley, 1999. Also available on-line from `http://java.sun.com/docs/books/jni`.

[12] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification Second Edition*. The Java™ Series. Addison-Wesley, 1999. Also available on-line from `http://java.sun.com/docs/books/vmspec`.

[13] The Open Group. *The Single UNIX Specification, Version 2: Threads*, 1997. Available online at `http://www.unix-systems.org/single_unix_specification_v2/xsh/threads.html`.

[14] Bill Pugh. The Java Memory Model. Web page available at `http://www.cs.umd.edu/~pugh/java/memoryModel`, February 2003 (last visited). The latest proposal is at `http://www.cs.umd.edu/~pugh/java/memoryModel/newest.pdf`. See also [10].

[6] TRON is an abbreviation of "The Real-time Operating System Nucleus."; BTRON is of "Business TRON."; CTRON is of "Communication and Central TRON."; ITRON is of "Industrial TRON."; μITRON is of "Micro ITRON."; JTRON is of "Java API for TRON." The μITRON Real-Time kernel specification is a *de facto* industry standard in Japan.

[15] Real-Time Java Working Group. Real-Time Core Extensions. Technical report, J-Consortium, September 2000. Available from `http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf`.

[16] Why are Thread.stop, Thread.suspend and Thread.resume Deprecated? Web page: `http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html`.

A Listings

Listing 2: Using threads in Java

```
/* 1 */ class SharedObject {
/* 2 */   boolean isInitialised;     // false by default
/* 3 */ }
/* 4 */
/* 5 */ class MyThread extends Thread {
/* 6 */   private SharedObject myObject;
/* 7 */   private Thread myParent;
/* 8 */
/* 9 */   public MyThread(SharedObject anObject) {
/*10 */     myObject = anObject;
/*11 */     myParent = Thread.currentThread();
/*12 */   }
/*13 */
/*14 */   public void run() {
/*15 */     // This is the main body of the thread
/*16 */     System.err.println(this.getName() + " Started");
/*17 */     synchronized(myObject) {
/*18 */       while (! myObject.isInitialised) {
/*19 */         try {
/*20 */           myObject.wait();      // Wait for initialisation to end.
/*21 */         } catch (InterruptedException e) { }
/*22 */       }
/*23 */     }
/*24 */     System.err.println(this.getName() + " my priority is " +
/*25 */                        this.getPriority());
/*26 */     System.err.println(this.getName() + " My parent is " +
/*27 */                        myParent.getName());
/*28 */     return;
/*29 */   }
/*30 */ }
/*31 */
/*32 */ class Initialise {
/*33 */   public static void main(String[] argv) {
/*34 */     Thread.currentThread().setName("main:");
/*35 */     SharedObject sharedObject = new SharedObject();
/*36 */     MyThread threadOne = new MyThread(sharedObject);
/*37 */     MyThread threadTwo = new MyThread(sharedObject);
/*38 */     threadOne.setName("\tthreadOne:");
/*39 */     threadOne.setPriority(java.lang.Thread.MAX_PRIORITY - 0);
/*40 */     threadTwo.setName("\t\tthreadTwo:");
/*41 */     threadTwo.setPriority(java.lang.Thread.MAX_PRIORITY - 1);
/*42 */     threadOne.start();
/*43 */     threadTwo.start();
/*44 */     System.err.println(Thread.currentThread().getName() +
/*45 */                        " Finished initialisation - my priority is " +
/*46 */                        Thread.currentThread().getPriority());
/*47 */     synchronized(sharedObject) {// End of initialisation.
/*48 */       sharedObject.isInitialised = true;
/*49 */       sharedObject.notifyAll();
/*50 */     }
/*51 */     try {
/*52 */       threadOne.join();
/*53 */     } catch (InterruptedException ie) {
/*54 */       System.err.println("Interrupted join on threadOne: " + ie);
```

```
/* 55 */     }
/* 56 */     try {
/* 57 */       threadTwo.join();
/* 58 */     } catch (InterruptedException ie) {
/* 59 */       System.err.println("Interrupted join on threadTwo: " + ie);
/* 60 */     }
/* 61 */   }
/* 62 */ }
```

Listing 3: Using threads in Java through the `Runnable` interface

```
/*  1 */ class SharedObject {
/*  2 */   boolean isInitialised;     // false by default
/*  3 */ }
/*  4 */
/*  5 */ class MyThread implements Runnable {
/*  6 */   private SharedObject myObject;
/*  7 */   private Thread myParent;
/*  8 */
/*  9 */   public MyThread(SharedObject anObject) {
/* 10 */     myObject = anObject;
/* 11 */     myParent = Thread.currentThread();
/* 12 */   }
/* 13 */
/* 14 */   public void run() {        // This is the main body of the thread
/* 15 */     Thread myself = Thread.currentThread();
/* 16 */     System.err.println(myself.getName() + " Started");
/* 17 */     synchronized(myObject) {
/* 18 */       while (! myObject.isInitialised) {
/* 19 */         try {
/* 20 */           myObject.wait();      // Wait for initialisation to end.
/* 21 */         } catch (InterruptedException e) { }
/* 22 */       }
/* 23 */     }
/* 24 */     System.err.println(myself.getName() + " my priority is " +
/* 25 */                        myself.getPriority());
/* 26 */     System.err.println(myself.getName() + " My parent is " +
/* 27 */                        myParent.getName());
/* 28 */     return;
/* 29 */   }
/* 30 */ }
/* 31 */
/* 32 */ class Initialise {
/* 33 */   public static void main(String[] argv) {
/* 34 */     Thread.currentThread().setName("main:");
/* 35 */     SharedObject sharedObject = new SharedObject();
/* 36 */     MyThread threadOne = new MyThread(sharedObject);
/* 37 */     MyThread threadTwo = new MyThread(sharedObject);
/* 38 */     ThreadGroup myGroup = Thread.currentThread().getThreadGroup();
/* 39 */     long stackSize = 1L;      // Just 1 Byte...
/* 40 */     Thread tOne = new Thread(myGroup, (Runnable) threadOne,
/* 41 */                              "\tthreadOne:", stackSize);
/* 42 */     Thread tTwo = new Thread(myGroup, (Runnable) threadTwo,
/* 43 */                              "\t\tthreadTwo:", stackSize);
/* 44 */     tOne.setPriority(java.lang.Thread.MAX_PRIORITY - 0);
/* 45 */     tTwo.setPriority(java.lang.Thread.MAX_PRIORITY - 1);
/* 46 */     tOne.start();
/* 47 */     tTwo.start();
/* 48 */     System.err.println(Thread.currentThread().getName() +
```



```
/* 49 */                        " Finished initialisation - my priority is " +
/* 50 */                        Thread.currentThread().getPriority());
/* 51 */     synchronized(sharedObject) {// End of initialisation.
/* 52 */       sharedObject.isInitialised = true;
/* 53 */       sharedObject.notifyAll();
/* 54 */     }
/* 55 */     try {
/* 56 */       tOne.join();
/* 57 */     } catch (InterruptedException ie) {
/* 58 */       System.err.println("Interrupted join on threadOne: " + ie);
/* 59 */     }
/* 60 */     try {
/* 61 */       tTwo.join();
/* 62 */     } catch (InterruptedException ie) {
/* 63 */       System.err.println("Interrupted join on threadTwo: " + ie);
/* 64 */     }
/* 65 */   }
/* 66 */ }
```

Listing 4: Interrupting threads in Java

```
/*  1 */ class ERR {
/*  2 */   static public void println(String msg){
/*  3 */     System.err.println(Thread.currentThread().toString() + ": " + msg);
/*  4 */   }
/*  5 */ }
/*  6 */ class MyThread extends Thread {
/*  7 */   Thread parent;
/*  8 */   public MyThread() {parent = Thread.currentThread();}
/*  9 */   public void run() {
/* 10 */     ERR.println("Hello");
/* 11 */     try {this.sleep(2);} catch (InterruptedException ie) {}
/* 12 */     parent.interrupt();
/* 13 */   }
/* 14 */ }
/* 15 */ class Main {
/* 16 */   public static void main(String[] argv) throws InterruptedException {
/* 17 */     MyThread aThread = new MyThread();
/* 18 */     ERR.println("this is main");
/* 19 */     aThread.start();          // calls run()
/* 20 */     try {
/* 21 */       aThread.join();
/* 22 */       if (Thread.currentThread().isInterrupted()) {
/* 23 */         ERR.println("Have been interrupted");
/* 24 */         throw new InterruptedException("Suicide...");
/* 25 */       }
/* 26 */     } catch (InterruptedException ie) {
/* 27 */       ERR.println("Un-joinable:" + ie);
/* 28 */       throw ie;
/* 29 */     } finally {
/* 30 */       ERR.println("Bye");
/* 31 */     }
/* 32 */     ERR.println("Never printed...");
/* 33 */   }
/* 34 */ }
```

Listing 5: Using timers in Java

```
/* 1 */
/* 2 */ import java.lang.*;
/* 3 */ import java.util.*;
/* 4 */
/* 5 */ class MyTimerTask extends TimerTask {
/* 6 */   private int MAX_TARDINESS = 15;
/* 7 */   private String myName = null;
/* 8 */
/* 9 */   public MyTimerTask(String aName) {myName=aName;}
/*10*/
/*11*/   public void run() {
/*12*/     // This is the main body of the thread
/*13*/     System.err.println("\tTimerTask"+myName+": Started");
/*14*/     if (System.currentTimeMillis() - scheduledExecutionTime() >= MAX_TARDINESS)
/*15*/       return;                // Too late; skip this execution.
/*16*/                              // Perform the task
/*17*/     System.err.println("\t\tTimerTask"+myName+": running @ " +
/*18*/                        System.currentTimeMillis());
/*19*/     return;
/*20*/   }
/*21*/ }
/*22*/
/*23*/ class Initialise {
/*24*/   public static void main(String[] argv) {
/*25*/     Thread.currentThread().setName("main:");
/*26*/     MyTimerTask aTimerTask = new MyTimerTask(" one ");
/*27*/     MyTimerTask bTimerTask = new MyTimerTask(" two ");
/*28*/     Timer      aTimer    = new Timer(false); // Not a daemon timer
/*29*/
/*30*/     // Start it after 10 milliseconds, with a period of 30 milliseconds
/*31*/     aTimer.scheduleAtFixedRate(aTimerTask, 10L, 20L);
/*32*/
/*33*/     // Start it after 30 milliseconds, with a period of 30 milliseconds
/*34*/     aTimer.scheduleAtFixedRate(bTimerTask, 30L, 30L);
/*35*/
/*36*/     try {
/*37*/       Thread.currentThread().sleep(100L);
/*38*/     } catch (InterruptedException ie) { }
/*39*/
/*40*/     bTimerTask.cancel();
/*41*/ // The following is wrong - we cannot re-schedule a canceled task.
/*42*/ //     aTimer.scheduleAtFixedRate(bTimerTask, 30L, 30L);
/*43*/
/*44*/     try {
/*45*/       Thread.currentThread().sleep(100L);
/*46*/     } catch (InterruptedException ie) { }
/*47*/
/*48*/     // Cancel all timer tasks
/*49*/     aTimer.cancel();
/*50*/
/*51*/
/*52*/
/*53*/     return;
/*54*/   }
/*55*/ }
```


Listing 6: Using JNI/CNI— the Java part

```
/* 1 */ class HelloWorld {
/* 2 */   private native void print();
/* 3 */   static {
/* 4 */     System.loadLibrary("HelloWorld");
/* 5 */   }
/* 6 */   public static void main(String[] args) {
/* 7 */     Thread.currentThread().setName("The Main from Java");
/* 8 */     new HelloWorld().print();
/* 9 */   }
/*10*/ }
```

Listing 7: Using JNI— the C part

```
/* 1 */ #include <jni.h>
/* 2 */ #include <stdio.h>
/* 3 */ #include "HelloWorld.h"
/* 4 */
/* 5 */ JNIEXPORT void JNICALL
/* 6 */ Java_HelloWorld_print(JNIEnv *env, jobject obj)
/* 7 */ {
/* 8 */   jstring Unicode_str;
/* 9 */   const jbyte *UTF_str;
/*10*/   jclass class;
/*11*/   jmethodID method_id;
/*12*/   jobject this_thread;
/*13*/
/*14*/   fprintf(stderr,"Starting Java_HelloWorld_print\n");
/*15*/
/*16*/   /*
/*17*/     In C++ the following call would have been written:
/*18*/
/*19*/     class = env->FindClass("java/lang/Thread");
/*20*/   */
/*21*/   class = (*env)->FindClass(env, "java/lang/Thread");
/*22*/   if (NULL == class) {
/*23*/     fprintf(stderr,"Could not find class java.lang.Thread\n");
/*24*/     return;
/*25*/   }
/*26*/
/*27*/   method_id = (*env)->GetStaticMethodID(env, class,
/*28*/     "currentThread", "()Ljava/lang/Thread;");
/*29*/   if (NULL == method_id) {
/*30*/     fprintf(stderr,
/*31*/       "Could not find static method currentThread of java.lang.Thread\n");
/*32*/     return;
/*33*/   }
/*34*/
/*35*/   this_thread = (*env)->CallStaticObjectMethod(env, class, method_id);
/*36*/   if (NULL == this_thread) {
/*37*/     fprintf(stderr,"Could not get the currentThread\n");
/*38*/     return;
/*39*/   }
/*40*/
/*41*/   class = (*env)->GetObjectClass(env, this_thread);
/*42*/   if (NULL == class) {
/*43*/     fprintf(stderr,"Could not find class java.lang.Thread\n");
/*44*/     return;
```

```
/* 45 */   }
/* 46 */
/* 47 */   method_id = (*env)->GetMethodID(env, class,
/* 48 */     "toString", "()Ljava/lang/String;");
/* 49 */   if (NULL == method_id) {
/* 50 */     fprintf(stderr,"Could not find method toString of java.lang.Thread\n");
/* 51 */     return;
/* 52 */   }
/* 53 */
/* 54 */   Unicode_str = (*env)->CallObjectMethod(env, this_thread, method_id);
/* 55 */   if (NULL == Unicode_str) {
/* 56 */     fprintf(stderr,"Did not get the name of the current thread\n");
/* 57 */     return;
/* 58 */   }
/* 59 */
/* 60 */   UTF_str = (*env)->GetStringUTFChars(env, Unicode_str, NULL);
/* 61 */   if (NULL == UTF_str) {
/* 62 */     return;                   /* OutOfMemoryError already thrown */
/* 63 */   }
/* 64 */
/* 65 */   /*
/* 66 */    * Here I *KNOW* that the UTF string returned contains only 7-bit ASCII
/* 67 */    * characters, that's why I'm passing it to printf.
/* 68 */    *
/* 69 */    * When this is *not* the case, see section 8.2 of the JNI Guide...
/* 70 */    */
/* 71 */   printf("Thread \"%s\" says using JNI: Hello World!\n", UTF_str);
/* 72 */
/* 73 */   /* Collecting our garbage to avoid a memory leak */
/* 74 */   (*env)->ReleaseStringUTFChars(env, Unicode_str, UTF_str);
/* 75 */   return;
/* 76 */ }
```


Listing 8: Using CNI— the C++ part

```
/*  1 */ // This file was created by 'gcjh -stubs'. -*- c++ -*-
/*  2 */ //
/*  3 */ // This file is intended to give you a head start on implementing native
/*  4 */ // methods using CNI.
/*  5 */ // Be aware: running 'gcjh -stubs ' once more for this class may
/*  6 */ // overwrite any edits you have made to this file.
/*  7 */
/*  8 */ #include <HelloWorld.h>
/*  9 */ #include <gcj/cni.h>
/* 10 */ #include <java/lang/UnsupportedOperationException.h>
/* 11 */
/* 12 */ // Includes I have added
/* 13 */ #include <java/lang/Thread.h>
/* 14 */ #include <java/lang/System.h>
/* 15 */ #include <java/io/PrintStream.h>
/* 16 */
/* 17 */ void
/* 18 */ HelloWorld::print ()
/* 19 */ {
/* 20 */   /* Original automatically created stub code was:
/* 21 */
/* 22 */   throw new ::java::lang::UnsupportedOperationException
/* 23 */     (JvNewStringLatin1 ("HelloWorld::print () not implemented"));
/* 24 */
/* 25 */   */
/* 26 */   java::lang::System::err->print
/* 27 */     (JvNewStringLatin1 ("Thread \""));
/* 28 */   java::lang::System::err->print
/* 29 */     ((java::lang::Thread::currentThread())->toString());
/* 30 */   java::lang::System::err->println
/* 31 */     (JvNewStringLatin1 ("\" says using CNI: Hello World!"));
/* 32 */ }
```

Listing 9: Using RTSJ

```
/*  1 */ /*
/*  2 */
/*  3 */   RTSJ_HOME=/import/linux/soft/src/rtsj-ri/refimp-1.0
/*  4 */   javac -bootclasspath ${RTSJ_HOME}/lib/foundation.jar rtsj-ex.java
/*  5 */
/*  6 */   (cd ${RTSJ_HOME}/pthreadrt ; test -f libpthreadrt.so  ln -s libpthreadrt.so.2.0 libpthreadrt.so)
/*  7 */
/*  8 */   LD_LIBRARY_PATH=${RTSJ_HOME}/pthreadrt:${LD_LIBRARY_PATH}
/*  9 */
/* 10 */   I don't have an EDF scheduler but I can *fake* one
/* 11 */
/* 12 */   ${RTSJ_HOME}/bin/tjvm \
/* 13 */     -Xbootclasspath=${RTSJ_HOME}/lib/foundation.jar \
/* 14 */     -Djava.class.path=`pwd` \
/* 15 */     -Djavax.realtime.scheduler.EDF=javax.realtime.PriorityScheduler \
/* 16 */     SchedExample
/* 17 */
/* 18 */   One would have normally run this as:
/* 19 */
/* 20 */   ${RTSJ_HOME}/bin/tjvm \
/* 21 */     -Xbootclasspath=${RTSJ_HOME}/lib/foundation.jar \
/* 22 */     -Djava.class.path=`pwd` \
```

```
/* 23 */    SchedExample
/* 24 */
/* 25 */ */
/* 26 */
/* 27 */ import java.lang.*;
/* 28 */ import java.util.*;
/* 29 */ import javax.realtime.*;
/* 30 */
/* 31 */ class MyRunnable implements Runnable {
/* 32 */   public static void remaining(String s, MemoryArea m) {
/* 33 */     System.err.println("\t"+s+":"+m
/* 34 */                        +" Size="+m.size()
/* 35 */                        +" Left="+m.memoryRemaining()
/* 36 */                        +" ");
/* 37 */   }
/* 38 */
/* 39 */   public void run() {
/* 40 */     RealtimeThread I = null;
/* 41 */     try {
/* 42 */       I = (RealtimeThread) Thread.currentThread();
/* 43 */     } catch (ClassCastException e) {
/* 44 */       System.err.println(e);
/* 45 */     }
/* 46 */
/* 47 */     System.err.println(I);
/* 48 */
/* 49 */     MemoryArea memArea = null;
/* 50 */
/* 51 */     int stackDepth = javax.realtime.RealtimeThread.getMemoryAreaStackDepth();
/* 52 */
/* 53 */     switch (stackDepth) {
/* 54 */      case 3:
/* 55 */       memArea = javax.realtime.RealtimeThread.getOuterMemoryArea(stackDepth-3);
/* 56 */       remaining("Heap ",memArea);
/* 57 */       /* fall through */
/* 58 */      case 2:
/* 59 */       memArea = javax.realtime.RealtimeThread.getOuterMemoryArea(stackDepth-2);
/* 60 */       remaining("oldMem ", memArea);
/* 61 */       /* fall through */
/* 62 */      case 1:
/* 63 */       memArea = javax.realtime.RealtimeThread.getOuterMemoryArea(stackDepth-1);
/* 64 */       remaining("newMem ", memArea);
/* 65 */
/* 66 */       System.err.print('\n');
/* 67 */     }
/* 68 */   }
/* 69 */ }
/* 70 */
/* 71 */ class SchedExample {
/* 72 */
/* 73 */   static protected LTMemory anewMem = null;
/* 74 */   static protected Runnable anewRun = null;
/* 75 */
/* 76 */   public static Scheduler findSched(String policy) {
/* 77 */    String className = System.getProperty("javax.realtime.scheduler."+policy);
/* 78 */    Class clazz;
/* 79 */    try {
/* 80 */      if (null != className && null != (clazz = Class.forName(className))) {
```



```
/* 81 */       System.err.println("findSched: Found " + policy);
/* 82 */       return (Scheduler) clazz.getMethod("instance",null).invoke(null,null);
/* 83 */     } else {
/* 84 */       System.err.println("findSched: Could not find " + policy);
/* 85 */     }
/* 86 */    } catch (Exception e) {
/* 87 */     System.err.println("findSched: " + e);
/* 88 */    }
/* 89 */    return null;
/* 90 */   }
/* 91 */
/* 92 */   public static void main(String[] args) throws Exception {
/* 93 */     Scheduler sched = findSched("EDF");
/* 94 */     if (null != sched) {
/* 95 */       System.err.println("main: Found EDF scheduling policy");
/* 96 */     } else {
/* 97 */       System.err.println("main: Could not find EDF scheduling policy"
/* 98 */                          + " - Using default");
/* 99 */       // What are the system properties anyway?
/*100*/       System.getProperties().list(System.err);
/*101*/       System.err.println("-- end of listing properties --");
/*102*/       sched = javax.realtime.Scheduler.getDefaultScheduler();
/*103*/     }
/*104*/     System.err.println("main: Your scheduling policy's *real* name is "
/*105*/                        + sched.getPolicyName());
/*106*/
/*107*/     AsyncEventHandler missHandler = new AsyncEventHandler(new
/*108*/                                                           Runnable ()
/*109*/       {
/*110*/         public void run() {
/*111*/           System.err.println("Missed a period - exiting");
/*112*/           System.exit(1);
/*113*/         }
/*114*/       }
/*115*/                                                           );
/*116*/
/*117*/     PeriodicParameters pp = new
/*118*/       PeriodicParameters(
/*119*/                          new RelativeTime(0,0),//when released
/*120*/                          new RelativeTime(400, 0), // period
/*121*/                          new RelativeTime(30, 0),   // cost
/*122*/                          new RelativeTime(60, 0), // deadline
/*123*/                          null,          // no Overrun Handler
/*124*/                          missHandler); // the Miss Period Handler
/*125*/     ImportanceParameters prio = new
/*126*/       ImportanceParameters(3, 3);
/*127*/     MemoryParameters mp = new
/*128*/       MemoryParameters(MemoryParameters.NO_MAX,MemoryParameters.NO_MAX);
/*129*/
/*130*/     final LTMemory oldMem = new LTMemory(6*1024, 6*1024);
/*131*/
/*132*/     ProcessingGroupParameters gp = null;
/*133*/
/*134*/     final Class argTypes[] = {long.class,// this is the type of a primitive long
/*135*/                               long.class};
/*136*/     // Find the constructor of the LTMemory class which takes two longs as
/*137*/     // arguments.
/*138*/     final java.lang.reflect.Constructor ltmemconstructor =
```

```
/*139*/        LTMemory.class.getConstructor(argTypes);
/*140*/    // You could also have used the Class.forName(String) to get the class:
/*141*/ //  final java.lang.reflect.Constructor ltmemconstructor =
/*142*/ //      Class.forName("javax.realtime.LTMemory").getConstructor(argTypes);
/*143*/
/*144*/    // I need an array of *Objects* to pass to the constructor invocation,
/*145*/    // so I must use Long, instead of long.
/*146*/    final Object minMax[] = { new Long(4*1024L), new Long(4*1024L) };
/*147*/
/*148*/    RealtimeThread rt = new
/*149*/      RealtimeThread(prio, pp, mp, oldMem, gp, new Runnable ()
/*150*/      {
/*151*/         public void comp(final String s1, final String s2) {
/*152*/           RealtimeThread I = null;
/*153*/           try {
/*154*/             I = (RealtimeThread) Thread.currentThread();
/*155*/           } catch (ClassCastException e) {
/*156*/             System.err.println(e);
/*157*/           }
/*158*/           if (I.getCurrentMemoryArea() != oldMem)
/*159*/             System.err.println("Should never happen");
/*160*/
/*161*/           try {
/*162*/             // Get a new instance using the constructor & args you have
/*163*/             // precalculated.
/*164*/             if (null == anewMem)
/*165*/               anewMem = (LTMemory)oldMem.newInstance(ltmemconstructor, minMax);
/*166*/           }
/*167*/           catch (java.lang.IllegalAccessException e){
/*168*/             System.err.println(e);
/*169*/             System.exit(20);
/*170*/           }
/*171*/           catch (java.lang.InstantiationException e){
/*172*/             System.err.println(e);
/*173*/             System.exit(21);
/*174*/           }
/*175*/           catch (Exception e) {
/*176*/             System.err.println(e);
/*177*/             System.exit(22);
/*178*/           }
/*179*/           final LTMemory newMem = anewMem;
/*180*/
/*181*/           // newRun *runs in* newMem, but is *allocated* on oldMem.
/*182*/           try {
/*183*/             // Get a new instance using the constructor & args you have
/*184*/             // precalculated.
/*185*/             if (null == anewRun)
/*186*/               anewRun = (MyRunnable) oldMem.newInstance(MyRunnable.class);
/*187*/           }
/*188*/           catch (java.lang.IllegalAccessException e){
/*189*/             System.err.println(e);
/*190*/             System.exit(30);
/*191*/           }
/*192*/           catch (java.lang.InstantiationException e){
/*193*/             System.err.println(e);
/*194*/             System.exit(31);
/*195*/           }
/*196*/           catch (Exception e) {
```



```
/*197*/             System.err.println(e);
/*198*/             System.exit(32);
/*199*/           }
/*200*/           final Runnable newRun = anewRun;
/*201*/
/*202*/           try {
/*203*/             newMem.joinAndEnter(newRun);
/*204*/           } catch (InterruptedException ie){
/*205*/             System.err.println(ie);
/*206*/           }
/*207*/         }
/*208*/
/*209*/         public void run() {
/*210*/           try {
/*211*/             RealtimeThread I =
/*212*/                  (RealtimeThread) Thread.currentThread();
/*213*/             MemoryArea myMem =
/*214*/                  I.getCurrentMemoryArea();
/*215*/             if (oldMem != myMem)
/*216*/               System.exit(10);
/*217*/
/*218*/             String first = new String(I + ": Started");
/*219*/             String next = new String(I + ": Running");
/*220*/
/*221*/             System.err.println(I+":");
/*222*/             MyRunnable.remaining("oldMem ", oldMem);
/*223*/             System.err.print('\n');
/*224*/
/*225*/             comp("1-Not using the right scope", first);
/*226*/
/*227*/             java.lang.Runtime.getRuntime().traceInstructions(true);
/*228*/             java.lang.Runtime.getRuntime().traceMethodCalls(true);
/*229*/
/*230*/             do {
/*231*/
/*232*/               comp("2-Not using the right scope", next);
/*233*/
/*234*/             } while(I.waitForNextPeriod());
/*235*/           } catch (ClassCastException e) {
/*236*/             System.err.println(e);
/*237*/           }
/*238*/         }
/*239*/       }
/*240*/                    );
/*241*/
/*242*/    rt.setScheduler(sched);
/*243*/    if (!rt.getScheduler().isFeasible())
/*244*/      throw new Exception("Not feasible");
/*245*/
/*246*/    System.err.println("\n\t\t\t\t\tmain is : "+Thread.currentThread()+'\n');
/*247*/
/*248*/    rt.start();              // Release the thread
/*249*/  }
/*250*/ }
```