8

Writing Optimized Code for Constrained Devices

Jonathan Allin

This chapter looks at how the Java developer can get the most from constrained devices such as Personal Digital Assistants (PDAs) and Wireless Information Devices (WIDs). Optimization is always important, but especially so on these constrained devices.

Although the chapter focuses on WIDs and PDAs, much of what we will discuss is more widely applicable. We will be concerned not just with improving performance but also with minimizing memory requirements.

The chapter's goal is to help you develop high quality Java applications for the Symbian OS. The intention is to achieve this by encouraging you to think about the issues involved and so make rational decisions, rather than by providing hard and fast rules for optimization.

We start with a number of general issues, including benchmarking, required performance, and principles of optimization. We then move on to discuss individual areas of optimization.

The next few sections discuss several specific areas for optimization: object creation, method and variable modifiers, the use of Strings, using containers sensibly. These ideas are brought together in an example (Section 8.9). We then look at some more advanced techniques: using JNI, blocking techniques to avoid polling, issues with graphics.

Section 8.13 provides a case study which explores optimization issues in depth. This leads us into the next section, which looks at the use of profiling tools in the context of the case study.

Subsequent sections discuss memory issues on constrained devices and the need to cope with out of memory situations, JIT and adaptive compilation technologies, and a couple of design patterns relevant to optimization. We end with a look at Symbian's plans to further enhance the performance of its Java implementation.

EX-1012
U.S. Patent 9,619,578

8.1 Introduction

8.1.1 What are we Starting With?

We have said that Symbian devices are by their nature constrained. In comparison to a desktop computer we will have a smaller screen, a keypad or pointer for input rather than a keyboard or mouse, restricted memory, restricted network and IO performance, and restricted processing power.

Of concern to us are memory, IO, and performance. Although the current range of Symbian devices typically have between 16 and 64 MB of RAM (e.g. Psion's Series 5MX and netBook respectively), the next generation of Symbian devices may well have no more than 8 MB of RAM. My current desktop computer has 256 MB of RAM!

Serial IO on a Symbian device is reasonable: both the IR and serial ports operate at 115.2 kBaud, but this is still far short of my office network which runs at 100 Mb/s or my wireless LAN which operates at 10 Mb/s. Wireless networking is currently more constrained. GSM provides 9.6 Kb/s, GPRS 2.5G technology will increase this to over 100 kb/s, while 3G UMTS will provide a maximum of 2 Mb/s (though typical data rates will be much less than this).

Benchmarking PDA devices is problematic. There are no good benchmarks for PDA devices and in the end the only real test is running representative applications on representative hardware. However, the Caffeine benchmark has the benefit of being easy to apply and the results are instructive (Table 8.1)

Table 8.1 Benchmark results for a number of devices using CaffeineMark 3

Device	Processor	CaffeineMark 3 result (higher is better)
Psion Series 5MX	36 MHz ARM710T	26
Psion netBook	190 MHz StrongARM SA1100	141
Jornada 820	190 MHz StrongARM SA1100	27–28
Desktop PC, NT4 SP4, 128MB Without JIT	366 MHz Pentium II	500
Desktop PC, NT4 SP4, 128MB With JIT	366 MHz Pentium II	5,000

Comparing the Psion netBook with the Jornada 820 (second and third lines) illustrates the benefit of running Java on the Symbian OS: a fivefold improvement in performance. In fact the Jornada, with a 190 MHz ARM processor, performs only marginally better than the Psion Series 5MX with a 36 MHz ARM 710T.

Comparison with desktops is really an apples to oranges comparison, but does make two points:

1. The dubiousness of benchmarks. In general, the desktop PC with JIT does not feel 35 times faster than the netBook.
2. The realities associated with running Java on small machines. There is, and always will be, a large gulf between the performance of PDA-type devices and desktop machines.

The JIT performance is particularly misleading. Comparing the underlying figures shows that the UI performance (the graphics, image, and dialog scores) is hardly affected by the JIT, perhaps improving performance by 10% . However, the logic performance improves by a factor of 140! This is simply because the JIT optimizer has noticed that nothing is really happening in the logic tests and has therefore optimized all the code out. (We'll talk about JIT and adaptive compilation technologies in Section 8.17).

The ARM implementation of the byte code interpreter loop has been written in assembler and optimized so that we should expect the performance on an ARM-based device to be relatively better than the Emulator's, which uses Sun's original C byte code interpreter loop. Table 8.2 suggests that this is indeed the case. The embedded performance is doubled while the overall performance (limited by the graphics capability) is about 25% better.

Table 8.2 Comparing the performance of the Emulator with ARM performance

Benchmark	Emulator on 500MHz PC	190MHz netBook	Emulator on 1MHz PC	1MHz netBook
Embedded	324	248	0.65	1.31
Overall	263	128	0.53	0.67

What performance do we need? Table 8.3 gives a very rough indication of the type of application we might expect to run for a given CaffeineMark.

Table 8.3 The range of benchmarks needed to run different types of applications

Benchmark	Application type
<50	Smartcard / Deeply Embedded - No UI
50–150	Simple E-commerce / Web applets
150–300	Simple 2D Games / E-commerce
300–1000	Complex 2D Games / Notepad / Diary
1000–2000	Simple 3D Games / Word processing
2000–3500	Complex 3D Games / Spreadsheet
3500+	Native C Level performance

The figures are for pure Java applications, whereas a typical application on the Symbian OS will make extensive use of the underlying Symbian OS functionality through, for instance, the JavaPhone APIs. This allows more to be

accomplished for a given benchmark. And, of course, much of this chapter is about how to write code that will help you get the most from a constrained device.

On the next generation of Symbian devices we can expect a CaffeineMark of around 70 on Smartphones and perhaps 200 on Communicators. Performance should roughly double MHz per MHz.

However, Symbian's fast Java implementation is only half the story. The rest is down to you, the developer. And that's what this chapter is about.

8.2 General Guidelines for Optimization

This section outlines some general principles for optimizing code. These don't attempt to say anything new; however, restating the obvious is not always a bad thing:

Get the design right: The biggest gains generally come from getting the overall architecture and design right. Deciding how operations should be split between server and client, what technologies to use (messaging, RMI, object database, relational database?), what hardware to use, even what languages are used. It is important to design to interfaces, not implementations. This makes it easier to slot in a different or improved algorithm; for example depending on your data size and how it might already be sorted, there are times when a pigeon sorting algorithm will be the best choice, and times when a bubble sort will be appropriate.

Optimize late: Optimizing too early means that you will be producing intricate code that will get in the way of good design.

Find out where you really need to optimize: Find out where the bottlenecks are and concentrate on sorting these out. This in turn requires access to suitable profiling and heap analysis tools.

Don't overoptimize: The more you optimize your code, the more highly tuned it becomes to the particular environment. If the environment changes or you want to use the code in a different application, you may find it runs more slowly. Compiler technology in particular can have a profound effect on the benefits or otherwise of a particular optimization.

Optimization is often in conflict with code clarity and maintainability: Improving performance at the code level generally (though not always) means writing more code, and quite often quite obscure code. We'll see an example of this later in our case study.

Improving performance is often in conflict with reliability: The corollary of the previous point is that you run the risk of introducing bugs when you optimize!

Improving start-up time and execution speed are often in conflict: I can frequently improve start-up time by deferring a task until it's required during execution. This is worthwhile if the task may not always be required, and even

then may still be worthwhile, especially if the task can be carried out by a background thread.

Improving performance is often in conflict with reducing memory: Many of the optimizations discussed in this chapter require extra code. Caching is a vital weapon in improving performance but requires extra memory.

Finally, the behavior of an optimization will vary with the platform. As a Java developer for Symbian devices you are likely to be working with three platforms: Java under Windows, the emulator, and a target device. The first two platforms may give a rough idea of the benefits or otherwise of an optimization; however, they cannot be used for a reliable analysis. In particular, the performance behavior of the emulator is very different to target hardware for reasons we will discuss in this chapter.

[Haggar] and [Larman et al] are good general references for optimizing Java applications.

8.3 Feedback and Responsiveness

Performance is in the eye of the beholder, so as well as ensuring that our application is fast as measured by a stopwatch it also needs to be responsive to the user and to provide them with feedback: the user should never be confronted with an unresponsive screen with no indication that something is happening. Large applications in particular can take a long time to initialize, so rather than leave the user with a blank screen, pop up a splash screen.

There's generally no wait icon on pen-based systems for the same reason there's no mouse pointer: the system has no knowledge of the pen's location (and hence where to display a wait icon) unless the pen is actually pressed on the screen. Therefore it's necessary to have a status area, animated icon, or some other way of conveying progress to the user. The Symbian OS's PrintNotify is available to Java developers (see ***http://www.symbiandevnet.com/***) and provides a convenient way of popping up a brief message.

Threads are an expensive resource and should therefore be used judiciously (this is why native Symbian OS applications tend to be single threaded and to rely on cooperative multitasking). However, you might want to consider loading or saving data in a separate thread, which will allow the user to carry on with other work. PowerPoint, Word, and other Windows applications lock the user out while a file is being saved: this is frustrating and unnecessary. Even if I'm saving a file I should still be able to read it or edit another file of the same type.

8.4 JAR Files and Class Path

Using JAR files will improve Applet download time. The browser has to create an HTTP request for each file transferred, so Applet download time will be

improved by putting all the required classes and resources into a single JAR file. The fact that JAR files are compressed means that there is less to transfer, further improving download time; however, this is generally of less importance.

JAR files should be used for applications precisely because they are compressed. Psion devices to date have shared RAM between processes and storage, therefore the less disk space required by your application means more dynamic memory is available to it, as well as other applications. Future Symbian devices will generally have persistent media for storage (e.g. flash memory for the internal disk) so that decreasing the size of the application's code will not free up dynamic memory. JAR files will nonetheless improve an application's performance a little because the class loader has less searching to do and uses less IO to load a class.

It's also worth noting that the class path is searched in order. Given the following class path:

```
-cp main.jar;sometimes.jar;almostnever.jar MyApp
```

Put the most commonly used classes in `main.jar` and those that are rarely used in `almostnever.jar`. Also remember that the system class paths will be searched after this, i.e. the class loader will look for a class in **classes.zip** and **javaphone.jar** after it has looked in the application's JAR files or class paths.

Class loading under Symbian OS can be slow. Before the VM loads a class it checks if it is already loaded. This requires checking against the several hundred preloaded classes.

If class loading becomes a bottleneck, consider using an explicit class path, e.g.:

```
-classpath main.jar;z:\system\java\lib\classes.zip
```

On one application, running on a netBook using an explicit class path reduced the start-up time from 25 s to 12 s!

Increasing the initial heap size might help a large program. The default heap is 128 KB. Adding the switch `-ms1mb` to the command line will increase this to 1 MB. Increasing the heap size in this way improves the String CaffeineMark by 60%, though this is a bit pathological!

As well as putting our application's classes into JAR files we should also include the application's resources. However, JAR files are accessed using a URL syntax and we need to be aware that: (a) the URL is relative to the directory from which the main class is loaded, and (b) for security reasons the resources we load must be in the same directory as this file or in a sub directory.

The DiceMachine is shown in Figure 8.2, running in the emulator. It rolls a number of dice in a similar way to a fruit machine rolling fruit.

In our example the URL is relative to **DiceMachine.class** and we have put our resources, an image and a sound file, into the **resources** subdirectory.

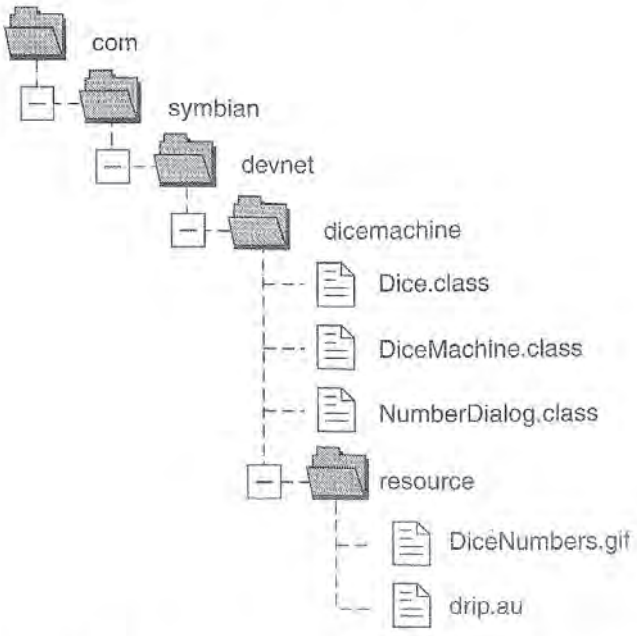


Figure 8.1 The organization of the DiceMachine's JAR file.

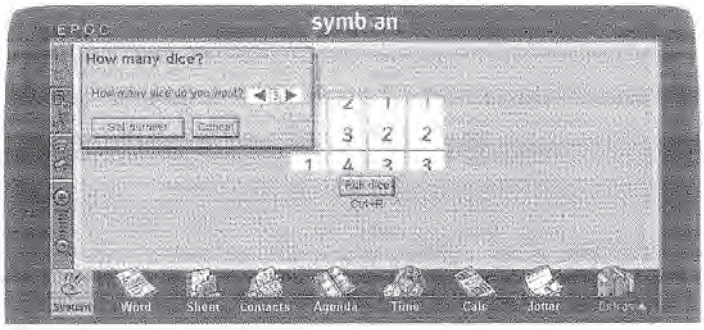


Figure 8.2 The DiceMachine running under the emulator, showing the dialog used to set the number of dice.

The following code, taken from the DiceMachine's constructor, is used to load these resources:

```
try
{
      URL urlSound = getClass().getResource("resources/drip.au");
      soundClip = (java.applet.AudioClip)urlSound.getContent();

      URL urlImage
           = getClass().getResource("resources/DiceNumbers.gif");
      diceImage = getToolkit().getImage(urlImage);
}
catch(IOException ie)
{
      // Handle any exception
}
```

The first two lines load the sound file and the next two load the image (we'll be coming back to the DiceMachine later). "resources/drip.au" and "resources/DiceNumbers.gif" follow the usual URL rules.

8.5 Object Creation

Object creation is an expensive process so it's worth examining your design to ensure you are not creating large numbers of objects, particularly short-lived objects, and to consider reusing objects. The AWT, for instance, is notorious for creating lots of short-lived objects: we'll look at this later.

Reusing objects means we don't waste time re-creating objects and is less work for the garbage collector when they are no longer needed.

The following is an extract from the DiceMachine's constructor. We create our dialog to change the number of dice in the constructor rather than re-create it every time it's displayed. `setNumberDice(4)` displays the first four of the six dice. We create a pool of dice, six in this case, rather than create new dice every time we change their number.

```
public DiceMachine()
{
     :
     :
     numberDialog = new NumberDialog(this, "How many dice?");

     for(int i = 0; i < MAX_DICE_NUMBER; i++)
     {
          dice[i] = new Dice(diceImage, soundClip);
     }
     setNumberDice(4);
     :
     :
     aboutDialog.setTitle("About" + title);
     aboutDialog.text = dialogText;
     show();
}
```

A valid alternative would be to create just the first four dice for our dice pool and then create additional dice only when we increased the number of dice.

It needs to be emphasized that we've used the Dice Machine only to illustrate a point. In a program as small as this, such decisions make little practical difference and worrying about them too much should definitely be regarded as over-optimization!

Consider object pools for such things as database connections and server connections. For instance, a file server program waits for a request from a client and on receipt of a request returns the appropriate file. The program might use a class called "FileRequestHandle" to listen for, and respond to,

file requests from the client. It creates a FileRequestHandler for each client it is serving, presumably on the port returned by `ServerSocket.accept()`. Alternatively it can create a pool of FileRequestHandlers at start-up and reinitialize an instance with the appropriate port number as needed.

The benefits of using a pool of FileRequestHandlers will be a faster connection time and an implicit limit on the number of clients, so that either a client will have an adequate bandwidth or will have to wait for a free FileRequestHandler. The downside could be a slower start-up time.

Object creation and pooling is discussed in detail in [Sosnoski].

8.6 Method Modifiers and In-Lining

Java provides a number of modifiers used to control the scope of methods and variables, i.e. private, protected, public, final, and static, plus volatile.

Methods or variables with no modifier have package scope, are non-static (i.e. belong to the instance of a class rather than the class itself), and are non-final, i.e. can be overridden in a derived class. We tend to use the default without thinking too much about it: it is a reasonably safe compromise. However, we shouldn't get lazy, and, in particular, as good designers we should keep things as private as possible and expose only what we absolutely have to. Invariant data (i.e. "constants") should in any case be marked as static and final. Such an approach reduces the risk of being stuck with an unsatisfactory public interface: we can always open up our design later but it's very hard to go back once we make something public.

Performance will also be affected by the scope we give our objects. Table 8.4 shows the results of tests with different modifiers. The test looped 10 million times and made 32 calls in each pass. Seven runs were made: the first five runs were accessor method calls with the method modified as indicated (S is static, P private, F final); the last two runs accessed a field directly, the field being either a static field or an instance field.

The first two columns used a 500 MHz Pentium III PC running NT 4 service pack 4.

Table 8.4 How different method modifiers affect performance

Run	Windows, Sun JVM, no JIT	EPOC Emulator JVM	netBook (190MHz StrongARM)
1. SPF method	12s	55s	344s
2. S method	12s	55s	344s
3. P method	12s	93s	281s
4. F method	12s	93s	281s
5. default method	52s	144s	281s
6. static field	12s	55s	70s
7. instance field	6s	36s	30s

The table illustrates a number of points:

- under Windows and the emulator, accessing a static method is faster than accessing a method with the default modifiers (i.e. a non-final method).
- under Windows and the emulator, accessing a static method is no slower than accessing a static field.
- on an ARM platform field, access is relatively fast while static method access is relatively slow.

The above also illustrates something we said earlier: that optimization behavior is platform dependent.

What is happening?

All method calls are "quickened" by the VM after their first execution, which replaces the by-name lookup with a more efficient lookup: virtual method dispatch is done using an index value into the class's method table, non-virtual methods are dispatched using a direct link to the method-block for the method. Both approaches offer a similar improvement, however, non-virtual methods can also be in-lined.

Public instance methods are virtual. Final methods may be virtual, but can never be overridden, so depending on the type of the object reference to make the call, in-lining may still be allowed. Private methods are non-virtual. Static methods are also non-virtual: they cannot be overriden by a derived class but only hidden. In addition, static methods don't have a "this" parameter, which saves a stack push.

The VM will attempt the actual in-lining at run-time after the first execution. It will replace the method call with an inline version of the method if the method body can be expressed in byte codes that fit into the method invocation byte codes. In practice this means that the simpler getter and setter functions can be in-lined by the VM.

However, this optimization was not implemented in the ARM version of the byte code interpreter in EPOC Release 5 (though it will be in releases beyond 6.0), which explains the poor performance of static methods on the netBook. This means you have a difficult decision to make if you are writing to Symbian OS v6.0 or EPOC Release 5. You should ideally use accessor methods when possible. However, if an accessor method becomes a performance bottleneck, you will need to access the variable directly.

You may have come across the -O option provided by the Java compiler, javac. This will attempt to in-line methods at compile time, even if the method is neither final, static, nor private, provided that no other class that is visible to the compiler attempts to override the method. This means that the -O option must be used with a group of classes, and that by using it you risk breaking compiler compatibility. *Caveat emptor*!

8.7 Strings

Java is very careful in how it handles Strings in order to minimize storage requirements and increase performance. This section outlines a number of issues that are a consequence of this approach.

8.7.1 Comparing Strings

In general we use `equals()` to compare two Strings, e.g.

```
if(stringA.equals(stringB)) {/* do something */}
```

However, you may recall that Java attempts to ensure that there is only one copy of a given String within its String literal pool. So given:

```
String stringA = "Now is the time";
String stringB = "Now is the time";
```

the expression `(stringA == stringB)` will generally return true. We have to say generally because Java JDK1.1 does not guarantee that it will maintain a single copy of identical Strings. However, we can force the issue by using `String.intern()`. This method returns a String which is guaranteed to be unique within the pool.

We can therefore do String comparisons using the much faster equality operator:

```
string1 = string1.intern();
:
string2 = string2.intern();
:
if(string1 == string2)) {/* do something */}
```

If your application is spending a lot of time comparing Strings (something particularly common in database applications), this approach can be of significant benefit.

8.7.2 Concatenating Strings

Another consequence of Java's String literal pool is that Strings are not mutable, i.e. a String cannot be modified once it has been created. So, although concatenating Strings is easy, it is also slow. It may be better to use StringBuffers instead.

The following code reads in characters one at a time from an InputStream, `inStream`, and appends each character to a String, `text`:

```
String text ="";
while(true)
{
    int value = inStream.read();
```

```
        if(value == -1) break;
        text = text + (char)value;
    }
```

The highlighted line is doing a lot more work than is apparent. `text` and `value` are both converted to StringBuffers, concatenated, then converted back to a String. This can be quite a performance hit in a tight loop.

The following is a better approach:

```
StringBuffer textBuffer = new StringBuffer(256);
while(true)
{
    int value = inStream.read();
    if(value == -1) break;
    textBuffer.append((char)value);
}
String text = textBuffer.toString();
```

By default a StringBuffer is created with an initial length of 16 characters; however, I know I will be reading at least 256 characters so I've used this as the initial capacity. The StringBuffer will automatically grow if more characters than this are appended.

An alternative to using the + operator to concatenate Strings is the `String.concat()` method. Given Strings s1, s2, and s3:

```
s3 = s1.concat(s2)
```

is over twice as fast as

```
s3 = s1 + s2
```

8.7.3 Using Strings as Keys in Hash Tables

Strings are often used as keys in hash tables. Hash table lookups make use of the key's hash code. Every Object, including Strings, implements `hashCode()`, which returns the Object's hash code. However, `String.hashCode()` recalculates the hash code each time it is called. To get around the problem [Larman et al] suggest creating a wrapper class around a String, called KeyString, which looks like this:

```
public final class KeyString
{
    private String key;
    private int hashCode;

    public KeyString(String key)
    {
        setKey(key);
    }
```

```
    public void setKey(String key)
    {
        this.key = key;
        hashCode = key.hashCode();
    }

    public int hashCode()
    {
        return hashCode;
    }

    public boolean equals(Object obj)
    {
        // See later
    }
}
```

The class caches the hash code rather than recalculating it each time. The use of `setKey()` allows a KeyString instance to be reused, potentially avoiding unnecessary Object creation.

If we reimplement `hashCode()` we are also required to reimplement `equals()`, and this suggests a further refinement that takes advantage of the `String.intern()` method we discussed in Section 8.7.1.

First we modify `setKey()`:

```
public void setKey(String key)
{
    this.key = key.intern();
    hashCode = key.hashCode();
}
```

Then we need to implement `equals()`

```
public boolean equals(Object obj)
{
    if((obj instanceof KeyString)
            && (key == ((KeyString)(obj)).key)) return true;
    else return false;
}
```

The highlighted `if` statement first checks that we are comparing two KeyString instances. The `if` statement's second clause, because we have interned our Strings, can very quickly check if the two KeyStrings are equivalent by comparing the identities of the Strings used as keys.

8.8 Containers

JDK1.1 provides four basic containers: Vector, Stack (which subclasses Vector), Hashtable, and Array. There are a few issues to be aware of.

Avoid using the default constructors for Hashtables and Vectors. Java does not specify the default initial capacity and therefore this could be bigger than we need. If we only want to store a few items we don't want a 1K container. This would be of little concern on a desktop computer but is a serious issue on a constrained device. Both Vector and Hashtable provide constructors that can be used to specify their initial size. Respectively these are `Vector(int initialCapacity)` and `Hashtable(int initialCapacity)`.

The initial capacity for a Hashtable should be thought about with a little care. Firstly, the JDK1.1 documentation states that "If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table." Secondly, [Larman et al] point out that to minimize clustering the size of a Hashtable should be a prime number and that powers of two (32, 64, 128) should definitely be avoided. Given that Hashtables grow by doubling and adding 1, if we believe that our Hashtable needs to grow we should choose an initial capacity that can grow and still be prime. A good candidate is 89, though it may be too large for smaller applications (keep doubling and adding 1 and see how long it takes before you get a non-prime number).

These containers grow by copying their internal data into larger arrays. This can be an expensive operation. Further, they can grow indefinitely. We might therefore be using far more memory than we expect and, perhaps worse, using it unpredictably. And although the container shrinks as we remove elements from it or if the container is garbage collected, the recovered memory is only available to our application. It is *not* available to other applications, Java or native.

Adding or removing items from Vectors and Hashtables can be slow. Removing an item from the middle of a Vector using `removeElementAt(n)` is achieved by copying all elements from `n` to the end of the Array to their preceding slot. As we will see in our case study (Section 8.13), this can take a long time.

Most of the comments we have made for Vectors apply to Stacks; however, there is no Stack constructor that sets its initial size so you will need to use the `setSize()` method inherited from Vector.

Strings are frequently used as keys in Hashtables; however, as we saw earlier (Section 8.7.3), they are not without problems and we suggested that a wrapper class that stores the String's hash value could provide faster lookup. Integers also make useful keys because `Integer.hashCode()` simply returns the Integer's int value.

Finally, think about whether an Array might provide a faster and more memory-efficient container.

8.9 Optimization: How not to do it

We've discussed a number of ideas for improving the performance of our code. This section brings several of these together in an example.

The following code is a slightly extreme example of how not to do it. The code goes through a company database extracting information about each employee:

```
float IMP_TO_MET = 1/2.2;
String nameList = "";
Vector employees = new Vector();
for(int id = 0; id < company.getNumberEmployees(); id = id+1)
{
    nameList += company.getCompanyName() + "\t";
    nameList += company.getEmployee(id).getName() + "\t";
    nameList += company.getEmployee(id).getWeight()*IMP_TO_MET
        + "\n";
    // do something with nameList
    nameList = "";
    employees.addElement(company.getEmployee(id));
}
```

Here are the problems:

Line 1: `IMP_TO_MET` should be declared static and final.

Line 3: It isn't necessary to use a Vector, particularly as we can find out the number of employees. An Array would be more appropriate. If we do need to use a Vector we should at least specify its initial size.

Line 4: We should dereference `company.getNumberEmployees` outside of the loop and use a local variable.

Line 4: `id++` may optimize better, and as we see in the next section, counting down rather than up can be more efficient.

Line 5: `getCompanyName` should be dereferenced outside the loop.

Line 7ff: `company.getEmployee(id)` should be obtained only once inside the loop.

Line 6ff: Should use a StringBuffer.

And this is how we might resolve these problems. We have defined `tab` and `nl` as `char` rather than as StringBuffers: appending one StringBuffer to another is no better than appending a String to a StringBuffer and significantly slower than appending a `char`.

```
public final static float IMP_TO_MET = 1/2.2;
final char tab = '\t';
final char nl = '\n';
int numberEmployees = company.getNumberEmployees();
Employee[] employees = new Employee[numberEmployees];
StringBuffer nameList = new StringBuffer(128);
```

```
StringBUffer companyName
     = new StringBuffer(company.getCompanyName()).append(tab);
for(int id = numberEmployees; --id >= 0;)
{
     nameList.append(companyName);
     Employee employee = company.getEmployee(id);
     nameList.append(employee.getName()).append(tab);
     nameList.append(employee.getWeight()*IMP_TO_MET).append(nl);
     // Do something with nameList
```

Reuse the StringBuffer, though note that this does not physically resize the StringBuffer's internal store:

```
     nameList.setLength(0);
     employees[id] = employee;
}
```

8.10 Going Native

The Java Native Interface (JNI) allows native code to be accessed in a standard way from Java. Given that C++ is generally faster than Java, an obvious way of speeding up an algorithm is to implement it in C++. However, before we go further it's worth listing a few potential problems:

- the native binary code will not be portable. The source code may not be portable.
- the native code will be bulkier than the equivalent Java code.
- working in a mixed C++ and Java environment is harder than working in a pure C++ environment, and of course a lot harder than working in a pure Java environment. Particular care must be given to avoiding memory leaks and exception handling.

There is quite a high overhead in making a native call, so there is little benefit in coding a simple algorithm in C++.

We can obtain an idea of the performance benefit of JNI programming by using `System.arraycopy()` as a benchmark. This is a standard Java method which copies the contents of one array into a second array, and is generally implemented as a native method. However, it doesn't use JNI but instead uses the VM's own native interface which provides direct access to the array contents and type data. To be strict in our comparison we should therefore have written a JNI equivalent of `arrayCopy()`: this could be slower because even if it just copies an array of Object; JNI calls are still required to access the array data.

The following is part of a test application used to compare native performance with Java performance:

```
:
private static Object[] sourceArray = new Object[100000];
```

```
private static Object[] destinationArray = new Object[100000];
private static int size;

private static void nativeCopy()
{
     System.arraycopy(sourceArray, 0, destinationArray, 0, size);
}

private static void javaCopy()
{
     for(int i = size; --i >=0; )
     {
          destinationArray[i] = sourceArray[i];
     }
}
:
```

The number of elements copied is determined by `size`, which is set by the user interface. The test was run on a Series 5MX. `javaCopy()` and `nativeCopy()` were called 320,000 times and the total durations compared:

size (i.e. number of elements copied)	0	1	2	3	10
Native copying using `nativeCopy()`	6.735s	7.203s	7.406s	7.594s	7.641s
Java copying using `javaCopy()`	3.266s	5.406s	7.531s	9.641s	24.500s

As an aside, the loop control statement in `javaCopy()` was originally written like this:

```
for(int i = 0; i < size; i++)
```

However, the loop termination comparison (`>=0`) used in the final code is built into the assembler's branch opcode and so is quicker. The difference is significant: the final `javaCopy()` ran about 16% faster than the original.

For zero length or single length arrays, Java copying was faster. For arrays larger than this, JNI copying was faster.

The results under the emulator were qualitatively different. Native copying was faster except for a zero length array, emphasizing once again that an optimization must be evaluated on target hardware.

It's not clear how meaningful the results are for zero length arrays, though dividing the total time of 6.7353s for native copying by the number of calls (320,000) suggests a native overhead of about 21μs per call on a Series 5MX.

8.11 Avoid Polling

Avoid using loops that poll. The following code polls the `keepRunning` flag:

```
Label status = new Label("Waiting");
boolean keepRunning = false;
:
public void run()
{
     status.setText("Started");
     while(keepRunning)
     {
          status.setText("In loop");
     }
     status.setText("Stopped");
}
```

Running the loop increased battery consumption from 66 mA to an unacceptable 163 mA on my Series 5MX. The battery consumption returned to 66 mA when `keepRunning` was set to false. Furthermore, the loop will hog the CPU and deprive other applications of CPU cycles.

Alternatives are to use an event model or a wait-notify model. Serial port support is provided by javax.comm (included with the JavaPhone APIs) and can provide an example of the former. If I were writing a low-level comms program and wanted access to the CTS line, I could poll it using `SerialPort.isCTS()`. However, it would be far more efficient to create a SerialPortEventListener and listen for a CTS event. We will look at using wait-notify in the Queue case study.

Both an event model and a wait-notify model require an extra Thread but in general this is worthwhile.

8.12 Graphics

Graphics coders face two problems: speed and flicker.

There are perhaps three causes of flicker:

- before drawing a new frame we need to clear the old frame by painting in the background. We can see flicker because the background shows though between frames.
- there is insufficient time to paint frames between paint requests so that we start drawing the new frame before the old frame has been completely drawn.
- the underlying OS buffers paint requests, so not every frame is drawn.

There are two main approaches we can take to reducing flicker: (1) use a clipping region so that we only repaint the area that needs to be updated, or

(2) create the new image in an off-screen buffer and copy this into our graphics context as required.

Using a clipping region will reduce but not eliminate flicker and should also be faster. An off-screen buffer should eliminate flicker and improve performance, but also requires extra memory for the buffer.

8.12.1 The Popcorn Drawing Demonstrator

Popcorn is a test program used to investigate the different approaches to painting.

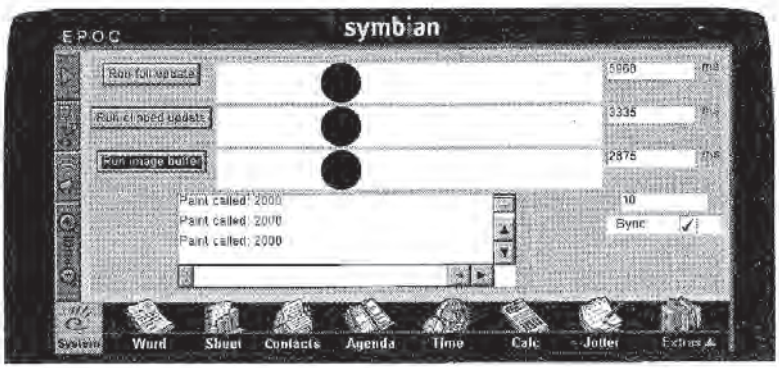


Figure 8.3 The Popcorn test running under the emulator.

The PlayingField class displays a puck (a filled circle) in a rectangular playing field. It is responsible for painting in the playing field and the puck. Every time `PlayingField.move()` is called the puck is redrawn `xDelta` pixels to the left or to the right. The horizontal size of the playing field is `xMax` pixels plus the size of the puck (`PUCK_SIZE`):

```
class PlayingField extends Canvas
{
    protected static final int PUCK_SIZE = 40;
    protected int xPos = 0;
    protected int xDelta = 1;
    protected int xMax = 350;

    PlayingField()
    {
        setBackground(Color.white);
    }


    public Dimension getPreferredSize()
    {
        return new Dimension(xMax + PUCK_SIZE, PUCK_SIZE);
    }

    public Dimension getMinimumSize()
```

```
    {
        return getPreferredSize();
    }
```

`move()` calls `repaint()`. However, remember that `repaint()` calls are buffered so that not every `repaint()` results in a call to `paint()`:

```
public void move()
{
    xPos += xDelta;
    if((xPos <= 0) || (xPos > xMax)) xDelta = -xDelta;
    repaint();
}
```

`paint()` draws our puck on the background. `paintCount` is used to keep track of how many times `paint()` is called:

```
public void paint(Graphics g)
{
    g.setColor(Color.black);
    g.fillOval(xPos, 0, PUCK_SIZE, PUCK_SIZE);
    paintCount++;
}
```

This implementation will flicker.

Using a Clipping Region

To define a clipping region we simply need to modify `move()` so that we call `repaint()` with a clipping rectangle:

```
public void move()
{
    xPos += xDelta;
    if((xPos <= 0) || (xPos > xMax)) xDelta = -xDelta;
    int absXDelta = Math.abs(xDelta);
    repaint(xPos - absXDelta, 0, PUCK_SIZE + 2*absXDelta,
    PUCK_SIZE);
}
```

We need to ensure the clipping region encompasses the old position and the new position of the puck irrespective of the puck's direction, which means we are repainting a bit more than we strictly need to.

Paint requests may be buffered so that the actual clipping rectangle will be the union of the clipping rectangles requested by each `repaint()`. We don't need to override `super.update()`: it will paint in just the amount of the background we need because it uses the same clipping rectangle as `paint()`.

Using an Image Buffer

There are various ways we could use an image buffer. In this case we've utilized an image buffer that looks like Figure 8.4.

`paint()` will copy the part of the image buffer within the clipping rectangle to the canvas. If lots of repaint requests are buffered up, the clipping region can grow to the width of the playing field, i.e. xMax + PUCK_SIZE. However, the puck can be at either the left-hand end or the right-hand end of the playing field, so the image buffer needs to be 2xMax + PUCK_SIZE in width.

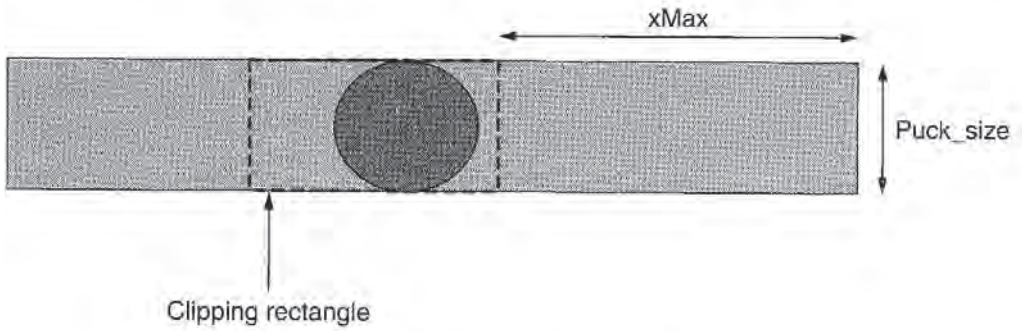


Figure 8.4 Popcorn's image buffer.

We need to declare our image buffer and the image buffer's graphics context:

```
Image puckImage;
Graphics gpi;
```

`paint()` now looks like this:

```
public void paint(Graphics g)
{
```

Create the image buffer if it doesn't exist:

```
    if(puckImage == null)
    {
        puckImage = createImage(PUCK_SIZE + 2*xMax, PUCK_SIZE);
    }
```

Once we have an image buffer, create the image (a black filled circle on a white background):

```
    if((puckImage != null) && (gpi == null))
    {
        gpi = puckImage.getGraphics();
        if(gpi != null)
        {
            gpi.setColor(Color.black);
```

```
                gpi.fillOval(xMax, 0, PUCK_SIZE, PUCK_SIZE);
            }
        }
```

Copy the image buffer to the correct position on the canvas. Only the part of the image within the clipping rectangle will be copied:

```
        g.drawImage(puckImage, xPos - xMax, 0, this);
        paintCount++;
}
```

We should override `update()` because we don't need to repaint the background:

```
public void update(Graphics g)
{
    paint(g);
}
```

`move()` is unchanged.

8.12.2 Comparing the Three Painting Implementations

The Popcorn program can be used to compare the actual painting times for the three implementations.

The following table gives the results of running the application on a Psion MXPro. Times are for 2,000 calls to `paint()`, PUCK_SIZE = 40, xMax = 350:

	Time	Comment
Unmodified paint	30s	Flicker
Clipping region	28s	Flicker
Image buffer	24s	Flicker-free

In all three implementations the puck's movement wasn't smooth but jerked every few seconds, presumably whenever the garbage collector was run.

The image buffer provided flicker-free animation and provided the fastest animation, though at the expense of extra memory. Under different circumstances an image buffer might be slower: copying a large image buffer is likely to take longer than a small amount of foreground drawing.

It is worth noting that flicker will not be apparent on the EPOC Release 5 emulator because all screen repaints use an image buffer. Furthermore, the screens used on WIDs and PDAs can be quite slow and blurry so that flicker may not be immediately obvious on an actual device.

Behaviour under Symbian OS v6.0 is different. All graphics operations are faster, however painting with an image buffer is relatively slower than using

a clipping region. The emulator has also changed, so that flicker is apparent under the emulator unless an image buffer is used.

8.13 Optimizing a Queue: A Semi-Fictitious Case Study

8.13.1 Background

I wanted a Queue to synchronize messages between two threads (a queue is a first in, first out container). My messages in this case were simple ints. As JDK1.1 doesn't supply a Queue class I had to create one. I started with an example from the *Java Fundamental Classes Reference* by [Grand & Knudsen]:

```
class Queue
{
     private Vector queue = new Vector();

     public final synchronized void push(int value)
     {
          queue.addElement(new Integer(value));
          notify();
     }

     public final synchronized int getNext()
          throws InterruptedException
     {
          while(queue.size() == 0) wait();
          Object obj = queue.elementAt(0);
          queue.removeElementAt(0);
          return ((Integer)(obj)).intValue();
     }
}
```

The Queue is implemented using a Vector. `push()` is used to add an int to the Queue while `getNext()` returns the next item in the Queue. The methods are synchronized so that should the Queue be empty `getNext()` will wait until it is notified by `push()` that data has been added to the Queue.

`push()` wraps the int to be pushed in an Integer and uses `Vector.addElement()` to add the Integer to the end of the Vector. `getNext()` reads the Integer at the start of the Vector using `Vector.elementAt(0)`, and having read the element removes it with `removeElementAt(0)`.

What happens if getting ints from the Queue takes a little longer than pushing them onto the Queue? Clearly the Queue will grow. The bottleneck is then with `queue.removeElementAt(0)`: A Vector is implemented using an Array, so to remove the element at the start of the Array the remaining elements have to be copied down one. As the Queue get bigger, the longer this operation takes, which means that getting ints from the Queue takes longer, which means the Queue grows more quickly...

8.13.2 The QueueTest Programme

I wrote a test programme, QueueTest, because I wasn't comfortable with using a Vector to implement a queue. QueueTest implements a Queue in three different ways: as the original Vector of Integers, as an Array of Integers, and as an Array of ints. The last two used the Array as a recirculating buffer with read pointers and write pointers to keep track of where I was. The third method allowed me to remove the cost of creating Integer objects: if my messages are plain ints, converting them to Integers is not required. The QueueTest class diagram is shown in Figure 8.5.

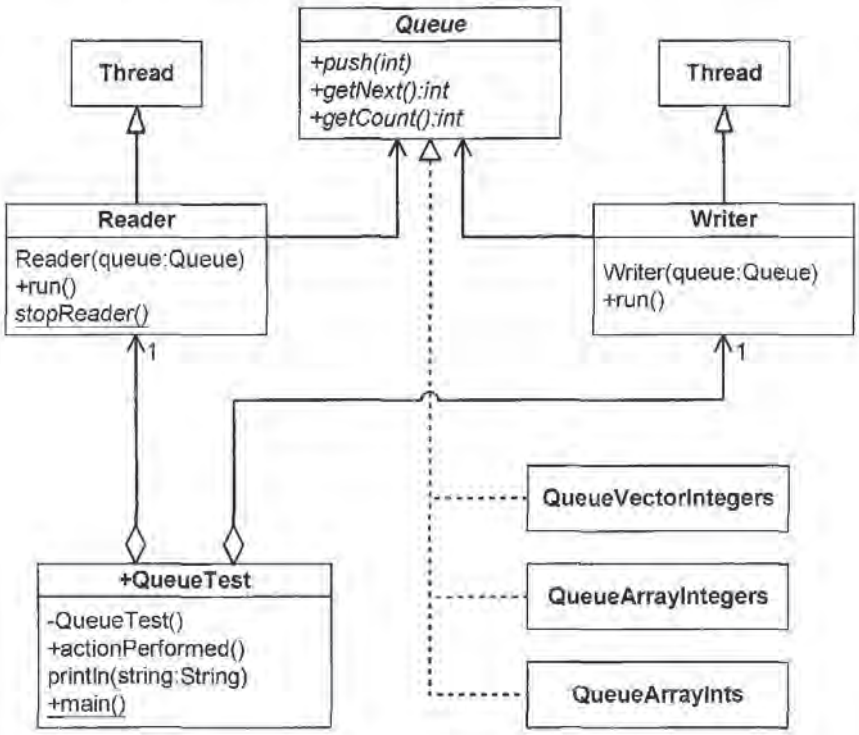


Figure 8.5 The QueueTest class diagram.

The Queue class has become an interface with three different implementations. As well as `push()` and `getNext()`, the Queue interface defines `getCount()`, a utility method to return the number of items currently in the Queue.

QueueTest creates two separate Threads, one that continuously reads from the Queue and one to write into the Queue.

8.13.3 First Results

Figure 8.6 shows the test application. The loop count is used to set the number of ints to be written, in this case 100,000. The "Report every" field says how often progress should be reported: in this case, after every 5,000 reads

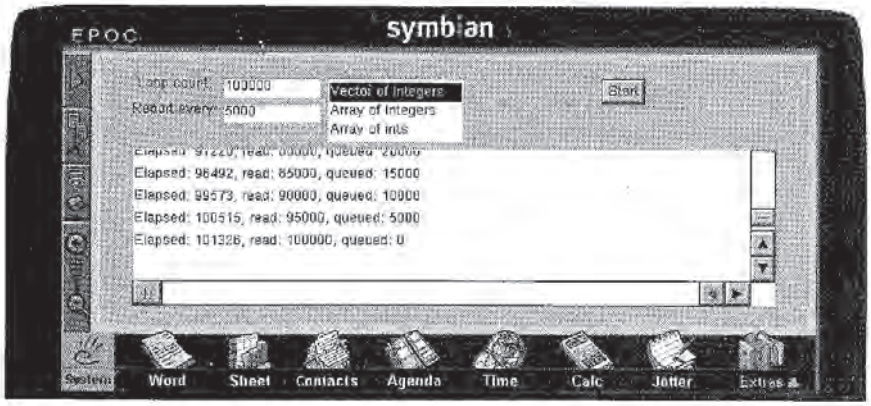


Figure 8.6 QueueTest screen shot.

the application reports the total time elapsed, the number of items read, and the number of items currently in the Queue. The time is in milliseconds, so this particular run, which used the QueueVectorIntegers implementation, took 101s to complete.

The results below were obtained using the EPOC Release 5 emulator, running on a 500 MHz Pentium III PC. `System.gc()` is called before each run to invoke the garbage collector, but even so, results can vary from run to run by half a second. The results are shown to the millisecond; however, timer resolution under Windows is only 100 ms!

Vector of Integers	101.326s
Array of Integers	21.781s
Array of ints	21.441s

Clearly, using an Array with re-circulating pointers is of considerable benefit, improving performance by nearly a factor of 5.

This is the QueueArrayInteger's `push()` method:

```
public final synchronized void push(int value)
{
    try
    {
        while(freeSpace == 0) wait();
        queue[writePointer] = new Integer(value);
        writePointer = (++writePointer) % queue.length;
        freeSpace--;
        notify();
    }
    catch(InterruptedException ie)
    {
        ie.printStackTrace();
    }
}
```

We are writing into a fixed size array of Integers, queue, of 100 elements. We first check if space is available and, if not, we wait (we could sit in a loop polling the `freeSpace` field, but, as we've discussed earlier, polling should be avoided). As soon as there's space in the queue Array we convert our int to an Integer and insert it into the Array at index `writePointer`. There's a very small optimization we might consider here: convert the int value to an Integer before we check for free space. We then have to increment our writePointer, wrapping if necessary, and decrement the freeSpace field. Finally we call `notify()` which will let `getNext()` know that there's some data available.

Here's the code for `getNext()`. The first line will wait if necessary for data to be written to the Queue (i.e. until `push()` calls `notify()`). We read from the queue Array using the `readPointer` index. This is then incremented (wrapping if necessary) together with the `freeSpace` count.

```
public final synchronized int getNext() throws InterruptedException
{
     while(freeSpace == queue.length) wait();
     Integer obj = queue[readPointer];
     readPointer = (++readPointer) % queue.length;
     freeSpace++;
     notify();
     return obj.intValue();
}
```

Notice that `getNext()` calls `wait()` inside a `while` loop. Between the time that `wait()` is notified that it should try to regain its lock and the time it actually does regain the lock, another thread might have called `getNext()` and emptied the Queue. The `while` loop guards against this condition. A similar argument applies to `push()`.

8.13.4 Avoiding the AWT

This is the `run()` method from the Reader Thread. The same method is used to read from all three Queue implementations:

```
public void run()
{
     lastValue = -1;
     readerThread = Thread.currentThread();
     long timeStart = System.currentTimeMillis();
     try
     {
          while(true)
          {
               int value = queue.getNext();
               if(++count %  QueueTest.getIntervalCount() == 0)
               {
```

```
                long timeElapsed = System.currentTimeMillis()-
                    timeStart;
                QueueTest.println("Elapsed:" + timeElapsed + ",
                    read:" + count + ", queued:" + queue.
                    getCount());
            }
            // Check we haven't missed a read or write
            if(lastValue + 1 ! = value) QueueTest.println ("Error
                in Queue. Last:" + lastValue + ", new:" + value);
            lastValue = value;
        }
    }
    catch(InterruptedException ie)
    {
        QueueTest.println(''Reader interrupte'');
    }
}
```

Successive values written by the Writer Thread are incremented: `lastValue` is compared with the current value to ensure we don't miss a read or write.

In the first line of the `while` loop `queue.getNext()` reads the next value from the Queue. The `if` block prints out the current status (i.e. elapsed time, the number of ints read, and the number of ints currently in the Queue) *if* the number of ints read modulo the "Report every" value is zero. This value is returned by the static method `QueueTest.getIntervalCount()`, which in turn reads the value directly from the TextField UI component, `reportIntervalF`:

```
static int getIntervalCount()
{
    int count = 5000;
    try
    {
        count = Integer.valueOf(reportIntervalF.getText()).
            intValue();
    }
    catch(NumberFormatException nfe)
    {
        QueueTest.loopCountF.setText("" + count);
    }
    return count;
}
```

In other words, every time an int is read from the Queue the reporting interval is read from the `loopCountF` TextField. We said before that the AWT is slow, so the next step was to read the value once at the start of the run into a local variable. The first part of the `run()` method now looks like this:

```
public void run()
{
```

```
:
int interval = QueueTest.getIntervalCount();
try
{
    while(true)
    {
        int value = queue.getNext();
        if(++count % interval == 0)
        {
            long timeElapsed = System.currentTimeMillis()
                - timeStart;
            QueueTest.println("Elapsed:" + timeElapsed + ",
                read:" + count + ", queued:" + queue.
                getCount());
        }
:
```

The improvement is dramatic (we've given up on our Vector implementation):

	Reading the report interval from the UI	Reading the report interval from a local variable
Array of Integers	21.781s	6.920s
Array of ints	21.441s	6.839s

We have trebled our performance and gained some idea of how slow the AWT can be. The difference between the two Array implementations was generally between 0.1s and 0.5s, which is hardly significant.

Perhaps the original reason for reading from the UI TextField was so that the reporting interval could be changed during a run. However, this can be achieved more elegantly by attaching an EventListener to the TextField.

8.13.5 Reducing Calls to Notify()

Returning to the `push()` and `getNext()` methods for both Array implementations we notice that `notify()` is being called by the former every time an item is added to the Queue and by the latter every time an item is removed from the Queue. However, `push()` only needs to call `notify()` when it adds an item to an empty Queue and `getNext()` should only call `notify()` when it removes an item from a full Queue.

`push()` and `getNext()` now look like this for the int Array implementation (the Array of Integers implementation is equivalent):

```
public final synchronized void push(int obj)
{
    try
    {
```

```
            while(freeSpace == 0) wait();
            queue[writePointer] = obj;
            writePointer = (++writePointer) % size;
            freeSpace--;
            if(freeSpace == size - 1) notify();
        }
        catch(InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }

    public final synchronized int getNext() throws InterruptedException
    {
        while(freeSpace == size) wait();
        int obj = queue[readPointer];
        readPointer = (++readPointer) % size;
        freeSpace++;
        if(freeSpace == 1) notify();
        return obj;
    }
```

This change gives the following results:

	Calling `notify()` on every read and write	Calling `notify()` only when required
Array of Integers	6.920s	2.374s
Array of ints	6.839s	1.832s

A significant improvement that underlies how much work `notify()` has to do. We can also start to see a consistent, if not necessarily significant, gain in using ints rather than Integer objects.

8.13.6 Back to the AWT

We've seen that the AWT can be expensive. Currently we are reporting our results into a TextArea. Is this slowing us down? To check this out, we can set the reporting interval to be the same as our loop count, i.e. 100,000:

	Report every 5,000 reads	Report once
Array of Integers	2.374s	2.013s
Array of ints	1.832s	1.453s

Reducing the amount of reporting has made a small improvement and we might now begin to argue that the difference between our two Arrays, i.e. the

cost of creating Integer objects, is significant. It's also worth noting that the results are more consistent, with runs not varying by more than 10 ms.

8.13.7 Increasing the Size of the Arrays

The size of the Arrays used by our Array implementations is 100. Increasing this to 512 might have two benefits: (a) using a power of two could make the modulo arithmetic a little faster, (b) using a bigger Array means that `notify()` should be called less often:

	Array of 100 elements	Array of 512 elements
Array of Integers	2.013s	1.923s
Array of ints	1.453s	1.352s

i.e. a small improvement that is probably not worth the cost of the larger Array.

8.13.8 Arithmetic Operations

Currently there is no hardware assistance available for division and modulo arithmetic on Symbian devices. So, if possible, arrange your divisions so that they are a power of two: you can then use the shift right operator for division. Similarly, for modulo arithmetic you can use a masking operation if the modulus is a power of two.

The `push()` and `getNext()` methods for the Array implementations respectively wrap the write and read pointers using the following lines of code:

```
writePointer = (++writePointer) % size;

readPointer = (++readPointer) % size;
```

Because we have changed the size of the Array to 512, a power of two, we can instead use the following for `push()`:

```
writePointer = (++writePointer) & 0x1ff;
```

and for `getNext()`:

```
readPointer = (++readPointer) & 0x1ff;
```

The benefit for our Queue implementations was not dramatic, perhaps about 30 ms. However, in an arithmetically intensive application the benefit could be significant.

8.14 Tools for Optimization

We took rather a guess and test approach to improving the performance of our Queue in the previous section. What we need are tools and techniques that will help us identify the bottlenecks: the ability to carry out on-target profiling is almost certainly more important than on-target debugging for the wireless application developer. We will discuss two tools in this section: profiling and heap analysis.

8.14.1 Profiling

Profiling tools allow us to see how much time was spent in a method and in a line of code in a method, to understand the calling tree, how much time a callee method spent servicing a calling method, etc.

There is a Sun technical note at ***http://developer.java.sun.com/developer/TechTips/2000/tt0124.html#tip2*** which covers the basics of profiling and Sun provide a tutorial at ***http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/perf3.html*** which concentrates on heap analysis (discussed below).

Both JDK1.1 and JDK1.2 support profiling. The JDK1.1 interpreter provides the `-prof` option to create profiling information. However, we will look at profiling our application with JDK1.2 for two reasons: (a) Symbian's next Java implementation will support the 1.2 model, and (b) tools and documentation are more readily available. The downside is that this won't be as representative as we'd like, made worse because for the moment we will have to profile under Windows rather than the Emulator or an actual device.

The following therefore assumes that you have installed JDK1.2. Profiling generates huge amounts of data, so to keep the profiling files to a reasonable size runs have to be fairly short. Here's the command line used to start up QueueTest:

```
java -Djava.compiler= -Xrunhprof:cpu=samples,depth=6,
    thread=y com. symbian.devnet.queue.QueueTest
```

`-Djava.compiler=` disables the JIT compiler

`-Xrunhprof` runs the profiler with a number of options:

- `cpu=samples` enables profiling with sampling: the Java VM regularly pauses execution and checks to see which method call is active. With enough samples you can pinpoint where your code spends its time.
- the other options are `times` and `old.times` checks the time whenever the code makes or returns from a subroutine call
- `depth` controls the depth of the stack traces saved during profiling. You may need to experiment with this value: large values produce more accurate information but increase the amount of data generated

- `thread=y` allows you to analyze the results by thread
- `file=<filename>` can be used to specify the name of the output file. By default, hprof generates an output file called **java.hprof.txt**

To generate a suitable run I set both the "Loop count" and the "Report every" field to 100,000, ran the test with the int Array, and quit. The output file is created when the application quits. This can be analyzed by hand but it's extremely tedious work.

The Performance Analysis Tool

A freeware analysis tool with the perhaps regrettable name of PerfAnal can be obtained at ***http://developer.java.sun.com/developer/technicalArticles/GUI/perfanal/***. Unzip this and run it with a command line of the following form:

```
java -jar <jar_dir>PerfAnal.jar <output_dir>java.hprof.txt
```

where `<jar_dir>` and `<output_dir>` locate respectively the PerfAnal jar file and the output file generated by our application.

This should bring up a window similar to that shown in Figure 8.7.

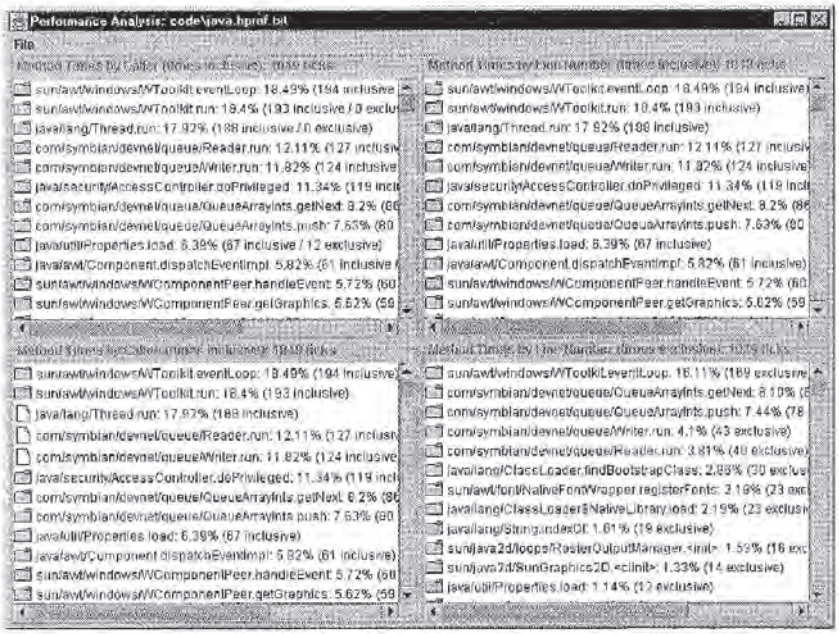


Figure 8.7 The Performance Analysis tool.

The four quadrants contain four different analyses of the data. Let's take a closer look at each quadrant.

Analyzing CPU Usage by Caller

Figure 8.8 shows the upper left quadrant, which ranks and analyses CPU usage by caller.

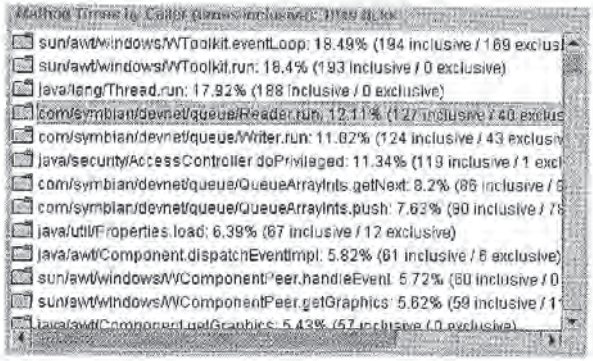


Figure 8.8 CPU usage by caller.

The total number of samples in our analysis was 1,049. Looking at the first line, we see that 18.49% of the time the application was running was spent in the event loop, i.e. waiting for user input. Looking at the 4th, highlighted, line we can see that 12.11% of the time was spent in the `run()` method of the Reader class. Including subroutine calls, this corresponded to 127 of our 1,049 samples, while the number of samples spent in the routine exclusive of subroutine calls was 40.

Analyzing Times by Callee

The lower left quadrant (Figure 8.9) allows us to analyze methods by callee. We can look at a method, find out how much time was spent in the method, who called the method, and how much time (or how many samples) were devoted by the method to each of its callees.

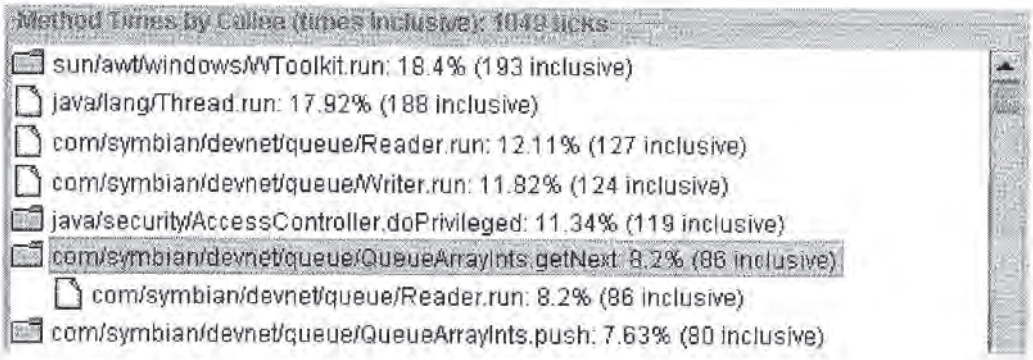


Figure 8.9 CPU usage by callee.

In the figure we see that 86 samples were spent in `QueueArrayInts.getNext()` and that this method was only called from `Reader.run()`.

Analyzing Times by Line Number

The top right and bottom right quadrants (Figure 8.10) allow times to be analyzed by line number, i.e. how much time, or how many samples, were



Figure 8.10 CPU usage by line number.

spent on a particular line of code. The top right quadrant times include subroutine calls while the bottom right times exclude subroutine times.

Inclusive of subroutine calls, 127 samples occurred in `Reader.run()`, while 40 samples occurred exclusive of subroutine calls.

Line 282 is from `Reader.run()`:

```
int value = queue.getNext();
```

This line gets the next item from our Queue: the complete code for `Reader.run()` was given earlier; 126 samples occurred in this line, including those samples occurring in the `getNext()` subroutine, and 40 samples occurred on this line excluding the subroutine.

Looking at the lower of these two quadrants, the second line shows that 8.1% of the time was spent in `getNext()`. In fact this appears to be the most expensive method we have written and is therefore a target for further optimization.

Synchronization and Improving the Speed of `getNext()`

We saw the code for `getNext()` in Section 8.13.5. `getNext()` is a synchronized method and we know that synchronization under JDK1.1 is expensive. We are synchronizing every time we read and write an item: how we might avoid this depends very much on how our Queue is being used, but the general guide is to do more in each call. One way might be to process all the items that are currently available in the Queue in one call to `getNext()`.

The following table shows that this strategy was beneficial:

	`getNext()` reads one item	`getNext()` reads all available items
Array of ints	1.352s	0.971s

I haven't shown the source code for two reasons: (a) my Queue is no longer a generic container: `getNext()` makes a callback to the Reader thread to process each item it reads from the Array, and (b) the code is not particularly elegant. However, the reader can obtain a copy of the code from ***http://www.symbian.com/books/***.

Synchronization is expensive under JDK1.1 and hence under the Java implementation provided with EPOC Release 5. However, this area has been optimized so that synchronization on Symbian Smartphones will be much faster. Nonetheless, synchronized code is generally slower than code that isn't synchronized and so should not be used unnecessarily.

8.14.2 Heap Analysis

The heap is the main block of memory used by running Java applications. The Heap Analysis Tool (HAT) lets you analyze profile reports of the heap and generate information on the objects used to run your application. This also means that the HAT can be used to track down memory leaks as well as providing a fascinating insight into where memory is going.

A copy of the HAT can be found at ***http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/hat_bin.zip***.

To analyze the data using the HAT we need to generate a binary format file. I started QueueTest with the following command line:

```
java -Djava.compiler = -Xrunhprof:file = QueueProfile.hprof,
    format = b com.symbian.devnet.queue.QueueTest
```

I set the reporting interval to be the same as the loop count, i.e. 100,000, and this time ran the Vector of Integers implementation, then quit. This created the binary report file **QueueProfile.hprof**.

Once you have installed the Heap Analysis Tool it can be started using a command of the form:

```
java hat.Main <output_dir>QueueProfile.hprof
```

where `<output_dir>` locates the output file generated by our application. **hat.zip** needs to be on the class path. This generates the output:

```
Started HTTP server on port 7000
Reading from code\QueueProfile.hprof...
Dump file created Sun Sep 03 07:24:18 GMT 2000
Snapshot read, resolving...
Resolving 8815 objects...
Chasing references, expect 17 dots.....
Eliminating duplicate references.....
Snapshot resolved.
Server is ready.
```

The above output tells you an HTTP server is started on port 7000 by default. To view this report, enter the URL ***http://localhost:7000*** in your web browser.

The first page displayed is the default report which shows a list of all the classes sorted by package. If we go down to our application package, `com.symbian.devnet.queue`, we can see a link to the Reader class, "class com.symbian.devnet.queue.Reader." Following this link brings up details of the class, including its superclass and static members. Under the "Instances" heading are two links which will bring up details of instances of this class: following the "Exclude subclasses" link brings up a summary of the instances of the Reader class. In this case there is only one: com.symbian.devnet.queue.Reader@0x008e9b00 (52 bytes). Clicking on this link brings up details of this instance, including details of where it was allocated from, which, in this case, is QueueTest's `actionPerformed()` method, found in QueueTest.java at line 465.

Returning to our initial page and going to the bottom we can see two links under the heading "Other Queries":

- Show all members of the rootset
- Show instance counts for all classes

Following the second of these two links brings up a list of all classes sorted by instance count. Heading the list are Strings and character arrays, with, in our case, 2369 and 2361 instances respectively. A summary at the bottom of the page states that there were a total of 8840 instances occupying 301,413 bytes of heap memory. Of this, 126,210 bytes were used for character arrays!

Not far from the top, however, we see that there are 543 instances of the Integer class. This number of Integers shouldn't be surprising as we ran our Array of Integers implementation with an Array of 512 elements.

Clicking on the "instance" link for the Integer record brings up a list of these instances. The bottom of this page confirms that there were indeed 543 instances and that they occupied 2172 bytes (which fortunately works out at 4 bytes per Integer). Following the link for one of these instances brings up the instance's details, which includes the fact that it was allocated from `push()` in the QueueArrayIntegers class and that this was in turn allocated from `run()` in the Writer class. We can also see the value of the Integer: in my case 99980, and where it is referenced from: element 140 of an Integer Array. Following the link back to the details for this Integer Array brings up a page listing all of its 512 elements and the fact that it was allocated from our QueueArrayIntegers class.

8.14.3 Debugging: A Useful Optimization

What will the compiler do with this code?

```
boolean debug = false;
if(debug)
```

```
{
    System.out.println(''Debug informatio'');
    // other statements
    System.out.println(''Status:'' + myClass);
}
```

In fact, the compiler will not compile this obviously "dead" code. Therefore you shouldn't be afraid of putting in debug statements in this manner as, provided the debug flag is false, the code will not add to the size of your class files. There is one "gotcha": if the debug flag is in a separate file, ensure that you recompile both files when you change the state of the debug flag.

8.14.4 Summarizing the Queue Case Study

We have made dramatic improvements to the performance of our Queue and this is a Good Thing. Our first implementation required over 100 s to process 100,000 items. Our final, practical, implementation required a little over 1 s. However, there is a downside: increased development time, bulkier code, more obscure code. But perhaps most importantly there's a very real risk we've introduced bugs into what started as a very simple implementation.

8.15 Design Patterns

In Section 8.2 we stated that one of the most important rules for optimization was getting the design right. For instance, it should be possible to defer the choice of sorting algorithm until the trade-offs between bubble sort, quick sort, or some other algorithm can be made intelligently on the basis of performance, memory requirements, and the size and distribution of the data set to be sorted. However, this requires designing your code such that substituting one sorting algorithm for another is painless.

We'll look at a couple of patterns in this section that can help achieve a better design.

8.15.1 Caching

Caching can produce very significant improvements in performance. The World Wide Web would probably be unusable if your desktop machine didn't cache pages. Disk performance relies on read-ahead caching. Virtual memory is a form of caching. Just about all modern processors use data caches because these can be accessed far more quickly than main memory. Sun's HotSpot compiler technology caches byte code as optimized native code.

The Queue Test case study discussed a very simple example of caching (Section 8.13.4). Rather than read the reporting interval count from the UI

every time we read an element from the Queue, we chose to cache the value in a local variable.

There are a number of issues to consider when architecting a cache (some of these are discussed in [Buschmann et al]):

What to cache: Cache objects which are likely to be reused, aren't too big, and that are slow to create or to access.

How much to cache: The 80:20 rule probably applies. Look for the 20% that is used 80% of the time. In practice, even a small cache can significantly improve performance. On the other hand, a cache that is bigger than the data set is wasting memory.

How to update the cache: When the cache becomes full you will have to throw away old items. Strategies include first in, first out, least frequently used, least recently used, and random. A random policy works surprisingly well because it is immume to pathological patterns.

How to maintain integrity between cached data and the data source: This will take some thought. For our trivial Queue example we could have used the UI component's event handler (the `actionPerformed()` method) to update the local variable when the user entered a new value. For more realistic situations you will be writing data into the cache as well as reading data from the cache. You can maintain cache integrity using an observer-observable model: read integrity is maintained by making the cache an observer of changes made to the primary data (the cache can also be an observable that is observed by the application); write integrity is maintained either by using a write-through policy such that data written by the application to the cache is simultaneously written to the primary data source or by making the primary data an observer of the cache.

The figure below shows how the optimum cache size depends on the speed of the cache versus the speed of the primary data source, and the size of the primary data set.

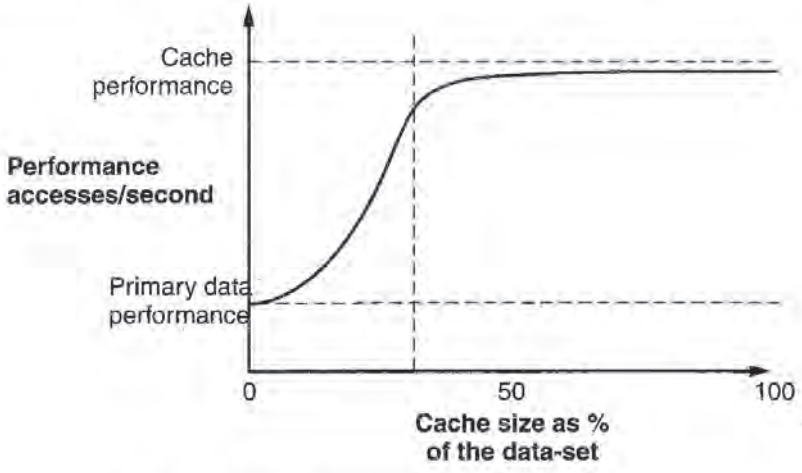


Figure 8.11 Choosing a suitable cache size.

The reason for having a cache is because it's faster to access objects in the cache: in this case the cache is about 5 times faster than the primary data set. Our actual performance (in accesses per second) won't quite reach the cache performance because we still have to decide whether or not to read data from the cache or the primary data set. The object usage pattern suggested by this curve indicates a cache size that is 30% of our primary data set would be about right.

8.15.2 Caching Results from a Database

We often want to scroll through records obtained from a database. This might be a remote or local database or we might be using the JavaPhone PIM APIs to access our address book.

It is impractical on a constrained device to hold all the data in memory from even a moderately sized database. Therefore consider (a) using a cache to hold data already read from the database and (b) predictively loading data: the latter can be carried out in a background Thread.

For instance, JavaPhone accesses the address book or calendar databases using an Iterator. Caching ahead will allow the user to look at an entry and then quickly iterate through the next few entries. Keeping a small cache of entries that have already been scrolled through will allow the user to scroll back. If the user scrolls back to the beginning of your cache then you have little choice but to reset the Iterator and read through the database again (which can also be performed in a background Thread).

8.15.3 Early or Lazy Instantiation

The Dice Machine created its dice at start-up time: this is early instantiation. Alternatively I could have created the dice as I needed them and added them to a pool, i.e. just in time or lazy instantiation. This would reduce my start-up time at the cost of increasing the time taken to add more dice the first time round. A third alternative would be to create new dice every time we change their number, but being good programmers we don't give this option too much consideration!

We talked earlier about creating object pools for things like database connections or server connections: we can either create a pool at start-up, i.e. early instantiation, or build up the pool to some maximum as needed, i.e. lazy instantiation.

8.15.4 Larger-Grained Operations

Setting up and tearing down an operation can take a long time compared to the time the operation spends doing real work. It's therefore worth seeing if we can do more in a given operation.

We saw an example of this in the Queue case study when we read all the available data in the Queue in one synchronized method.

Jar files are used to transfer multiple objects in one HTTP request.

Using buffered streams means that we transfer multiple items of data at one time, rather than item by item or byte by byte. It's rare that unbuffered IO has to be used, and buffered IO should always be the default.

8.16 Memory Management

It's not often that a Java application will run out of memory on a desktop computer. However, this is not the case for PDAs and WIDs. We should regard out of memory as the rule, not the exception, and handle out of memory gracefully.

The JDK1.1 garbage collector does not return memory to the system. Freed memory will only be available to your application: it will not be available to other Java or native applications. Therefore don't grab memory just in case you need it; you'll deprive other programs of this much required resource. Remember that Vectors and recursive routines have unconstrained memory requirements.

Of course, once your application quits and the JVM exits, the memory it used will become available to other applications.

8.16.1 Memory Leaks

Java has a different notion of a memory leak to C++. In C++ a memory leak occurs when a reference to an allocated object is lost before delete is called on the object. If an object is no longer referenced in Java it will be garbage collected, so C++-style memory leaks cannot occur.

However, a similar effect is created by a Java object that is no longer used but is still referenced by another Java object. Care should therefore be taken to de-reference such objects.

8.16.2 Mimicking Small Memory Machines in the Emulator

By default the emulator allocates 64 MB of memory, which is far greater than is generally available on a WID or PDA. However, the emulator can be configured to allocate a smaller amount of memory by modifying the relevant **.ini** file located in **\epoc32\data**.

A typical **.ini** file contains the following:

```
ScreenOffsetX 90
ScreenOffsetY 51
LedOffsetX 36
LedSize 14
_epoc_drive_j H:\erjp
```

Append a line of the form:

```
MegabytesOfFreeMemory 3
```

A real device will require around 1 MB for system use, so the above line will emulate a 4 MB EPOC device.

8.16.3 Invoking Heap Failure

The debug version of the emulator provides a heap allocation failure tool that can be used to invoke heap failures. Its use is documented in the C++ SDK and in Professional Symbian Programming [Tasker et al], p. 158ff.

We've seen that the AWT can be quite slow, partly because the AWT generates many temporary objects, particularly for event handling. The other difficulty we face is that because the heap allocation failure can occur in the VM the error message reported back will reflect the higher-level symptom and not the underlying cause, i.e. out of memory. We will frequently get a "class not found" error or "create from component" error.

We can demonstrate this using the Dice Machine example (see Section 8.5). If you have access to it, launch it from the debug emulator with the following command line:

```
\epoc32\release\wins\deb\java_g -cd c:\system\apps\DiceMachine -cp
        Code com.symbian.epoc.dicemachine.DiceMachine
```

Invoke the heap allocation tool by entering Ctrl+Alt+Shift+P and set the failure type to deterministic and the rate to 500 (Figure 8.12).

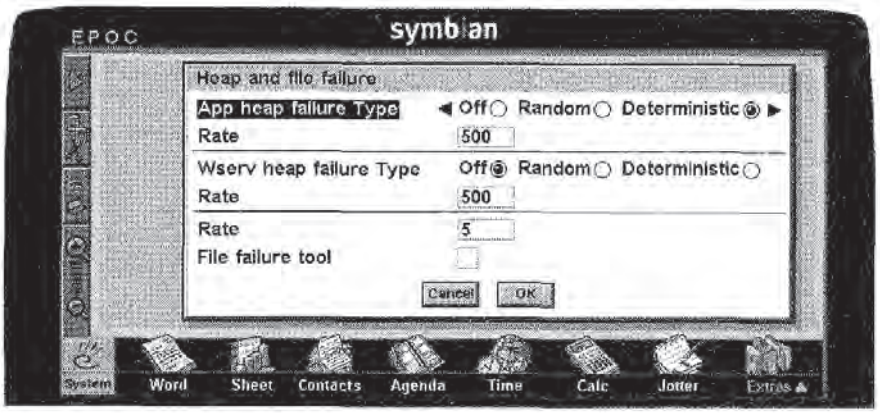


Figure 8.12 Using the heap failure tool under the Emulator.

Click on the OK button and roll the dice. You'll see an error similar to the following reported in the command window:

```
java.lang.NoClassDefFoundError: sun/awt/ScreenUpdaterEntry
    at sun.awt.ScreenUpdater.notify(ScreenUpdater.java:154)
    at sun.awt.ScreenUpdater.notify(ScreenUpdater.java:128)
    at
sun.awt.Epoc32ComponentPeer.repaint(Epoc32ComponentPeer.java:103)
    at java.awt.Component.repaint(Component.java:1355)
```

```
    at java.awt.Component.repaint(Component.java:1296)
    at com.symbian.epoc.dicemachine.Dice.bump(Dice.java:49)
    at
com.symbian.epoc.dicemachine.DiceMachine.run(DiceMachine.java:191)
    at java.lang.Thread.run(Thread.java:474)
```

Not until the failure rate is set to about 1300 will you be able to roll the dice without incurring a failure. However, the Dice Machine code does not explicitly generate any new classes, i.e. the AWT makes about 1300 heap allocations when the dice are rolled! Some of these may be related to loading classes; many are likely to be related to short-lived C++ objects and strings used to display the Dice Machine's UI.

8.16.4 Defensive Coding to Handle Out-of-Memory

How can we protect users of our applications from out of memory? The previous section has highlighted the problem in picking up heap allocation failures. Fortunately, under Java out of memory errors are unlikely to be caused by short-lived objects: the garbage collector should kick in before this happens.

Here are a couple of strategies:

- once your application has started, check how much free memory is available (you can use `freeMemory()` from java.lang.Runtime). If there is insufficient (and only you can judge what that means!) then give the user the opportunity to take appropriate action such as closing down an application
- create any large objects or arrays in `try-catch` blocks and catch any OutOfMemoryError exception that might be thrown. Do your best in the `catch` clause either to shut down gracefully or to take some action that will allow the user to carry on.

8.17 JIT and Adaptive Compilers

Most applications will benefit from improved compiler technology. However, this shouldn't be seen as a panacea because Java applications spend a lot of their time executing native code.

8.17.1 Just in Time Compilers

A JIT is a code generator that converts Java byte code into native machine code. Java programs invoked with a JIT generally run much faster than when the byte code is executed by an interpreter.

When a method is first called the JIT compiler compiles the method block into native code which is then stored. If code is only called once you will not see a significant performance gain: most of the gain is achieved the second

time the JIT calls a method. The JIT compiler also ignores class constructors so it makes sense to keep constructor code to a minimum.

8.17.2 Java HotSpot and Adaptive Compilation

The Java HotSpot virtual machine uses adaptive optimization and improved garbage collection to improve performance.

HotSpot compiles and in-lines methods that it has determined are used the most by the application. This means that on the first pass Java byte codes are interpreted as if there were no enhanced compiler present. If the code is determined to be a hot spot, the compiler will compile the byte codes into native code, cache the native code, then in-line the method.

An advantage of selective compilation over a JIT compiler is that the byte compiler can spend more time generating highly optimized code for the areas that would benefit most from optimization and by the same token avoid compiling code that would run best in interpreted mode.

Java HotSpot's garbage collector introduces several improvements over existing garbage collectors:

- the garbage collector is termed a fully accurate collector, which means that it knows exactly what is an object reference and what is just data.
- the garbage collector uses direct references to objects on the heap rather than object handles. This increased knowledge means that memory fragmentation can be reduced, resulting in a more compact memory footprint.
- the garbage collector uses generational copying. Java creates a large number of objects on the heap and often these objects are short-lived. By placing newly created objects in a memory "nursery" waiting for the nursery to fill, and then only copying the remaining live objects to a new area, the block of memory that the nursery used can be freed in one block. This means that the VM does not have to search for a hole in the heap for each new object and that smaller sections of memory are being manipulated. For older objects the garbage collector makes a sweep through the heap and compacts holes from dead objects directly, removing the need for a free list used in earlier garbage collection algorithms.

The perception of garbage collection pauses is removed by staggering the compaction of large free object spaces into smaller groups and compacting them incrementally

The Java HotSpot VM improves existing synchronized code. Synchronized methods and code blocks have always had a performance overhead when run in a Java VM. HotSpot implements the monitor entry and exit synchronization points itself rather than depending on the local OS to provide this synchronization. This results in a large improvement in speed, especially for often heavily synchronized GUI applications.

8.18 Obfuscators

Class files carry a lot of information from the original source file needed for dynamic linking. This makes it fairly straightforward to take a class file and reverse-compile it to a source file that can bare an uncanny resemblance to the original, including names of classes, methods, and variables.

Obfuscators are intended to make this reverse compilation process less useful. They also can reduce code size by removing unused data and symbolic names from compiled Java classes and by replacing longer identifiers with shorter ones.

A number of Java obfuscators are described in JavaWorld's resources section at ***http://www.javaworld.com/javaworld/tools/jw-tools-complier.html***.

8.19 What's Next—Enhancing Java Performance

We showed earlier that Symbian's Java implementation was relatively fast, but clearly the faster it is the more we can do. Symbian is therefore investing significant effort into improving all aspects of its Java performance. For instance:

Optimizations in the Java implementation include:

- reducing the cost of synchronization
- reducing the method call overhead
- speeding up class loading

Various accelerators are under investigation:

- hardware accelerators
- hotspot technologies
- JIT compilers
- replacing Java library classes with native implementations

Graphics:

- improving redrawing algorithms
- reducing the overhead of making graphics calls

8.20 Conclusion

We've looked at a various ideas for improving the performance of our code and in Section 8.2 we listed a number of guiding principles. Perhaps the two most important are these:

- optimization is always important, but especially so on constrained devices.
- get the design right.

It is possible on a desktop machine to get away with a poorly designed Java application; however, this is not true on a WID or PDA. The corollary is also true: a well-designed Java application on a constrained device can outperform a badly designed application on a desktop machine.

By thinking carefully about design and optimization we can create surprisingly complex Java applications for our WID that will perform just as effectively as an equivalent C++ application.

8.21 References

Buschmann et al *A system of patterns* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. Wiley, ISBN 0-471-95869-7

Grand & Knudsen *Java Fundamental Classes Reference* by Mark Grand, Jonathan Knudsen. O'Reilly, ISBN 1-5692-241-7

Haggar *Practical Java—Programming Language Guide* by Peter Hagar. Addison-Wesley, ISBN 0-201-61646-7

Larman et al *Java 2 Performance and idiom guide* by Craig Larman & Rhett Guthrie. Prentice Hall, ISBN 0-13-014260-3

Sosnoski "Smart object-management saves the day" by Dennis M Sosnoski ***http://www.javaworld.com/javaworld/jw-11-1999/f_jw-11-performance.html***

Tasker et al *Professional Symbian Programming* by Martin Tasker, Jonathan Allin, Jonathan Dixon, John Forrest, Mark Heath, Tim Richardson, Mark Shackman. Wrox, ISBN 1-861003-03-X

Section 3: Creating Wireless Java Services

The purpose of this section is to give operators and service providers in particular an understanding of how Java can meet their needs and the importance of Java to the wireless market. It covers security and provisioning of wireless Java applications and looks at two case studies from the Symbian community

9.	Creating Wireless Services	page 295–301
10.	Security	page 303–321
11.	Provisioning	page 323–337
12.	Case Studies	page 339–365

9

Creating Wireless Java Services

Jonathan Allin

Symbian devices are being developed to meet the requirements of operators and content providers with the goal of providing an open platform capable of safely and securely delivering highly functional, added value services. A rich Java environment is an important, probably vital, link in achieving this goal.

The purpose of this section of the book, "Creating wireless Java services," is to provide operators and service providers in particular with an understanding of how Java can meet their needs and its importance to the wireless market.

9.1 Introduction to Services

9.1.1 The Network Operator's Perspective

Operators are facing falling average revenues per user while facing an ever-greater need for forward investment. Many operators have invested heavily in 3G licenses and their shareholders are looking for a return on this investment. For first generation analog networks and second generation digital networks (GSM/TDMA/CDMA) revenue has generally come from air time. With GPRS and third generation networks, operators will need to offset lower airtime margins with far greater revenue from services.

Current average revenues per user are around $30 per month. This figure needs to rise to $60 if operators are to recover their 3G license costs. There will therefore be a drive for high value operator-based services, which will provide value both when connected and when disconnected from a network. These services will make use of information held by a variety of organizations (theatres, travel companies, television and radio companies), including the operators themselves who will want to utilize the information they hold on user usage profiles, location, buying patterns etc. However, not all of the revenue

will come directly from customers; a proportion will come through advertising and from service providers such as airlines.

In contrast to Internet-based e-commerce, billing systems are already in place, giving wireless commerce a head start.

9.1.2 The Service Provider's View

Service providers face a number of significant issues:

- security: in order to prevent malicious code we need to know who has provided an application and that the provider can be trusted, that the application hasn't been tampered with, and that its actions are limited to match the degree of trust we have in the provider. We must also protect transmitted data, such as PINs, confidential documents, or patient records, and we need a secure platform to enable wireless commerce. We also need a security model that can support Digital Rights Management and generally protect IPR
- provisioning: how applications arrive at the device, which might be by a Web or WAP browser, SMS, or Bluetooth. This also covers mechanisms for updating and removing applications, and how an operator can push an application or operating system upgrade onto a device
- device variability: how to target a wide range of devices such as features phones, Smartphones, and Communicators, with wide variations in each of these three categories. How to establish a device's features and capabilities
- rapid development: developing new services quickly and reliably, and having confidence that faults in the software won't show up weeks or even months after the service has been deployed.

Java offers a solution to security, is likely to succeed with provisioning, and provides the most promising (or least painful!) solution for multi-client deployment. Finally, Java provides a better vehicle than C++ for developing services quickly and reliably.

Chapter 10 looks at the general requirements for a secure system and the components that are needed to deliver services and applications safely and securely.

Chapter 11 discusses application and service delivery. It looks at the available technology and provides some ideas of how we can manage the wide variation in target devices.

Chapter 12 describes two case studies. The first, "Simple Conference," is a whiteboard application that provides a text message service and a simple shared drawing board. The second, "Handheld Travel Assistant," is a location-based service that provides a map, transport information, client or vehicle tracking, and information on points of interest.

9.2 Wireless Services

9.2.1 Provisioning Wireless Services

Wireless Information Devices enable far richer provisioning models that go beyond the presentation of essentially static information by Web or WAP browsers. Advanced client applications will be delivered by application providers through channels that include SMS, WAP, and Web browsers, which will then have access to data and information that may just come from the application provider, but could also come from a variety of other sources. Application providers will use information about the client's capability to configure an application for the client, and data will need formatting for both the client and the provisioning channel.

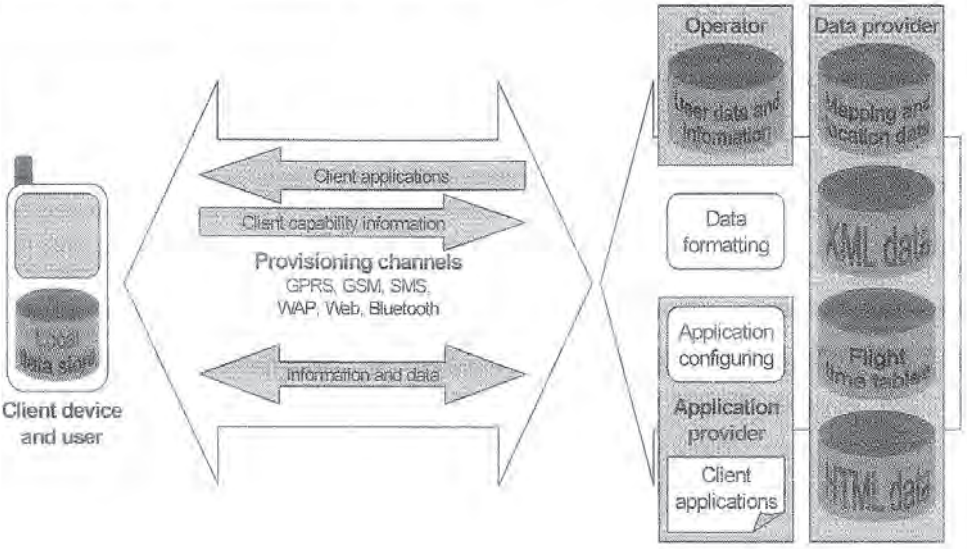


Figure 9.1 Wireless service provisioning model.

Operators will have their own data stores and may also control the services provided to the client. Applications and information will be both pulled by the client and pushed by the operator or service provider.

Importantly, client devices will cache data locally, only connecting to the data source to update and synchronize their information.

9.2.2 What Makes a Good Wireless Service?

For services to be successful they must meet a number of (more or less) obvious criteria:

- they must meet a user's need or solve a user's problem
- they must at least create a new opportunity for a user

- to be financially viable to the operator or service provider they need to make use of the added value that can be provided by server-based services or information held in external databases
- they need to be of use to the user even when disconnected from the network. This is firstly because users will not have permanent access to a network, and secondly because bandwidth will be limited.

In the introduction, we briefly discussed the characteristics of, and opportunities for, Wireless Information Devices. Let's walk through some of these opportunities in a bit more detail.

9.2.3 Location-Based Services

Location-based services seem to be the 'Next Big Thing' after games. They can be divided into positional information services that provide coordinates (this might be by GPS, signal strength triangulation, or proximity beacons), and push services that provide location-related information, such as local restaurants or special offers, that depends on the user's location.

Coordinate services answer "where am I?", " where is?", "how do I get there?". If you are walking through Florence the answer to "where am I?" might be Giotto's bell tower and your destination could be the Palazzo Pitti. Options to get from one to the other should cover foot, gondola, and other available modes of transport.

Push-type location services will enable you to receive information about local items that are of interest to you, such as restaurants, museums and shops.

Standards are needed to define high-level APIs for establishing the user's coordinates and to combine location information from different service suppliers, e.g. map information from one vendor, shop information from another, and transport information from a third.

Organizations Working with Location-Based Services

LIF (***http://www.locationforum.org/***) is the Location Interoperability Forum, formed by Motorola, Nokia, and Ericsson. LIF's purpose is to define, develop, and promote a common location services solution for wireless devices, irrespective of their underlying air interface technologies and positioning methods.

Companies developing location-based applications include MapInfo (***http://www.mapinfo.com/***), who are a member of LIF. Their technology provides route planning, dynamic map generation, and local area information services. Also Maporama (***http://www.maporama.com/***) who are large in Europe, and Webraska (***http://www.webraska.com***) who provide direction, location, and traffic information for wireless services.

The RoutePlanner Example

The RoutePlanner demonstration was written specifically for the Symbian OS. It was created by Cloudscape ***http://www.cloudscape.com/***), who are now part of Informix, and is based on Cloudscape's pure Java object relational database.

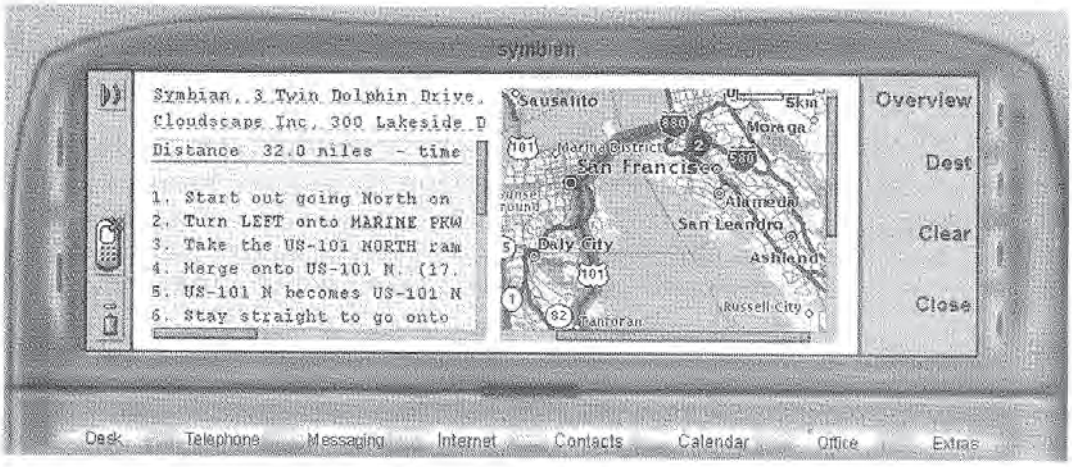


Figure 9.2 RoutePlanner demonstration from Cloudscape (running under the Crystal emulator)

RoutePlanner is intended to help you navigate around Silicon Valley. Enter your starting point in the top TextField (it doesn't have to be an exact match) and destination in the second TextField. The application will then pull the route out of the local database, together with a map showing the overall route and a map showing the area around your destination.

However, the interesting aspect is what happens if the route isn't in the local database. In this case the application will open a network connection to the appropriate website and download the information, so that over a period of time the database will contain details of your favorite routes. The application also allows you to enter new locations.

RoutePlanner typifies important features of wireless applications. It makes good use of local resources (the database of information and available processing power) and so minimizes the use of airtime and reliance on a network. It presents the information simply, making the application straightforward to use.

Other suppliers of databases for the Symbian OS include Oracle with Oracle 8i Light, and PointBase (***http://www.pointbase.com/***) who also provide a Java object-relational database.

9.2.4 Whiteboarding

Whiteboard services allow geographically separated users to cooperate on a task. In general, each user sees the same information (perhaps a diagram or document) and can update the information. Updates made by one user are reflected to the other users.

Benefits added by the server include extra power, maintaining the whiteboard model and maintaining whiteboard chat rooms. Users will be using a variety of client devices, and therefore the whiteboard model must be client independent. Either the server or the client will render the model to match the client's capabilities. More generally, clients will be able to download "shareable" objects from a whiteboard server, such as graphics primitives or more complex objects, perhaps an annotated map. They key is that a user can modify the object and that the modifications are shared with the other users.

9.2.5 Tailored News Services

Receiving information about the topics that interest us at a time that suits us. A topic might be quite specialized and require accessing local news services. For instance, I might want to track a famine relief program in Nepal.

9.2.6 Wireless Commerce

The opportunity to purchase goods or services securely and easily using a mobile phone. Hard goods such as the weekly shopping, soft goods that can be instantly downloaded such as stock information or a piece of music, or services such as banking: the ability to transfer money between accounts, control standing orders, or transfer money to someone else's account.

9.2.7 Mobile and Paperless Ticketing

A ticket is a way of confirming that the holder has paid for a service, such as a train journey or an airline flight, or a concert performance or a film, but it doesn't have to be in the form of a piece of paper.

It will be possible to buy tickets directly using your wireless phone. The ticket will be held in a secure format on the phone and then checked and canceled as you enter the cinema or theatre, or board the plane or train.

9.2.8 Travel and Traffic Congestion Information

We have all been on a car journey and suddenly hit the back of queue, when a few hundred meters down the road was a turn-off that might have provided an alternative route. We would benefit from traffic and congestion information that was pushed onto our WIDs in a timely manner, so that we could plan our journeys better and avoid delays.

9.3 The Simple Conference and Handheld Travel Assistant Case Studies

The last two chapters in this section describe two services, Simple Conference and the Handheld Travel Assistant, and show how these services work within the service provisioning model we saw earlier.

Common architectural themes start to emerge from these examples:

- strategies for coping with the wide variations in clients
- the benefit of separating client from service code
- separating the data source from data presentation (e.g. XML and WAP)
- using bearer-independent communications protocols
- local caching and local processing for off-network functionality
- adapting to the available data rate.

10

Java Security on Symbian OS

Jonathan Allin

This chapter looks at the issues of security in relation to service and application delivery. It discusses the general requirements for a secure system and the components that are used to fulfil these requirements. It describes Java's security features, particularly as they relate to the current Symbian OS v6.0, and also summarizes the native security capabilities. The Java security APIs are illustrated with the "Thumbprint" example.

The chapter ends with a short discussion on how platform security might evolve in the future.

10.1 Developing a Secure Environment

All but the simplest wireless services depend on the secure delivery of trusted applications and the secure exchange of information.

How then do we create a suitably secure environment? There are five key ideas we need to address: authentication, integrity, confidentiality, authorization, and non-repudiation.

10.1.1 Authentication

Authentication is the process of ensuring that the person or organization we are talking to is who they claim to be. For instance, we need to ensure that an application or document really was downloaded from www.symbian.com and not from www.cowboys.us pretending to be Symbian. Similarly, I want to be certain that I am indeed paying my loan company rather than Grabit & Lamb.

10.1.2 Integrity

Integrity is about ensuring that a document or application has not been tampered with. A modified application could make unauthorized calls or remove funds

from my bank account. Therefore I need to know that an application, even though it has come from a reputable source, has not since been modified, either prior to installation or once it was on my device.

10.1.3 Confidentiality

Confidentiality ensures that only authorized parties can read my data. I may have confidential documents I want to send to a client, and I don't want anyone else reading my health records or details of my bank account. It's also essential that passwords and PINs can be sent confidentially.

10.1.4 Authorization

Authorization is about safety. The application or requesting party should be able to undertake only certain, "safe" operations. For instance, I might allow an application from an unknown source to access a printer or read files in a particular directory, but I wouldn't want to grant it access to the phone or to my complete set of files. On the other hand, I might grant an application from a totally trusted source complete access to the system; for example, a phone operator from time to time will want to download patches or upgrades to the operating system.

10.1.5 Non-repudiation

Non-repudiation is also known as auditing or accountability. It's about ensuring that a party cannot later deny having sent us a document or receiving a cash transfer, or a customer cannot deny placing an order nor a merchant deny obtaining payment securely.

More precisely, non-repudiation means that both the sender(s) and recipient(s) can prove to a third party that:

- the sender(s) did indeed send the transaction(s)
- the recipient(s) received the identical transaction(s).

Non-repudiation requires that each transaction is signed with a digital signature. A trusted third party (who might be a certificate authority or the PKI) is needed to store these proofs of transaction and must also securely time-stamp them. In the case of a dispute the trusted third party must be able to provide these proofs.

A useful summary is provided by ***http://www.it.isy.liu.se/~asa/publications/NonRepudiation.html***.

10.1.6 End-to-end Security

Security must be implemented seamlessly from the originating server right through to the device. Having to decrypt and re-encrypt at a gateway introduces

a security risk. This is often necessary when moving between protocols such as WAP and HTTP. In particular, concerns with WAP security originate from the requirement to translate between wireless (WTLS) and Internet (TLS) encryption. There have been reports of malicious attacks entering the WAP gateway for packet sniffing and spoofing.

10.2 The Components Needed to Create a Secure Environment

Implementing our five key ingredients for a secure environment (authentication, integrity, confidentiality, authorization, and non-repudiation) requires a number of building blocks. This section describes some of these building blocks and at the same time provides a glossary of security-related terms.

Encryption: Taking clear text and encoding it with a known and well-understood algorithm to produce a cipher text. The algorithm is seeded with a key.

Decryption: Decoding cipher text into clear text, generally using the same algorithm that was used to encrypt the text. The key may be the key used for encryption (symmetric encryption) or a complementary key (asymmetric encryption).

Key: Keys range in size from tens of bits for symmetric encryption (e.g. DES) to thousands of bits for asymmetric encryption (e.g. RSA).

Key pair: Key pairs consist of a public key and a private key. They are used for asymmetric encryption, which means that if data is encrypted using a private key it must be decrypted using the matching public key (symmetric encryption uses the same key to encrypt and decrypt data). The private key should be known only to the owner, while the public key can be known to anyone.

DES: Data Encryption Standard. Symmetric encryption, which means that the same key is used for encryption and decryption. DES encodes 64-bit blocks with a 56-bit key. It's fast and, provided the key is kept confidential, it's very secure. The problem is transporting the key securely between parties.

Message digest: A digital "fingerprint"of some data. A message digest is relatively small (tens of bytes) and of a fixed size. It is generated from the original data using a one-way algorithm, which means that the message digest cannot be used to create the original data. It is also important that the amount by which the original data is changed is randomly related to the size of the change in the message digest. In other words, a small change in the original data can produce a large change in the digest. Equally, it should be very unlikely that two different sets of data can produce the same digest. Examples of message digest algorithms are MD5 and SHA-1.

Digital signature: A message digest signed with a private key and which is therefore not forgeable (assuming the private key is protected). A document's

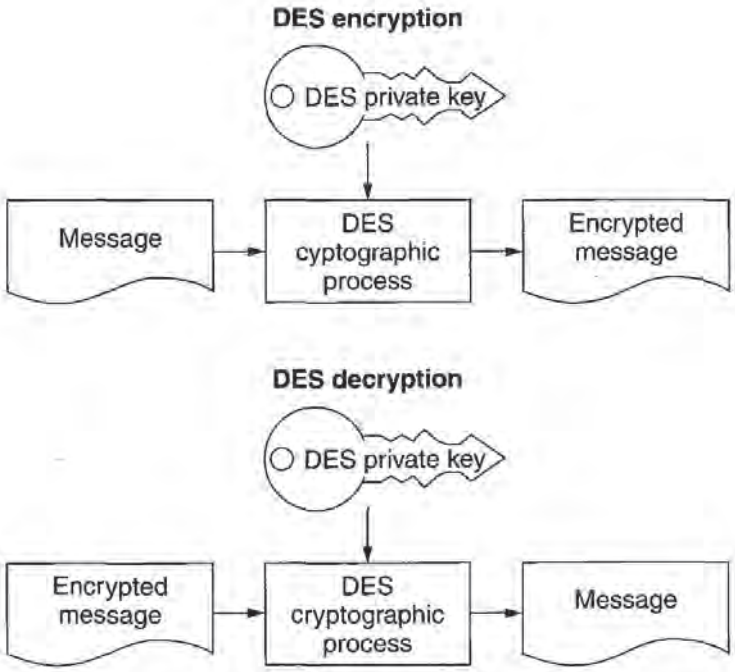


Figure 10.1 Using DES to encrypt and decrypt a message (the cipher text).

digital signature can be used to authenticate the document, i.e. ensure that it was sent by the person or organization who claims to have sent it, and that it has not been tampered with.

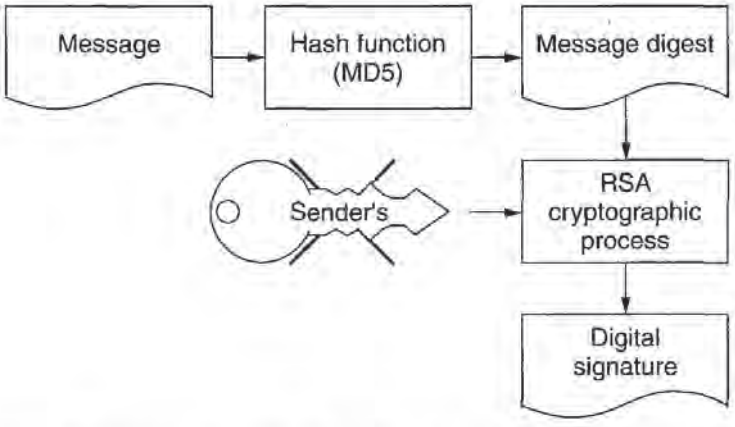


Figure 10.2 Creating a digital signature. The key with an "X" behind it represents the sender's private key.

Certificates (or Digital Certificates): A way to guarantee that a public key belongs to the person or organization it represents. The owner of the public key requests a certificate from a Certificate Authority (CA) and must satisfy the CA that they are the person or body they claim to be. The CA then issues the owner with a certificate, signed with the CA's private key. X.509 is the common standard for certificates.

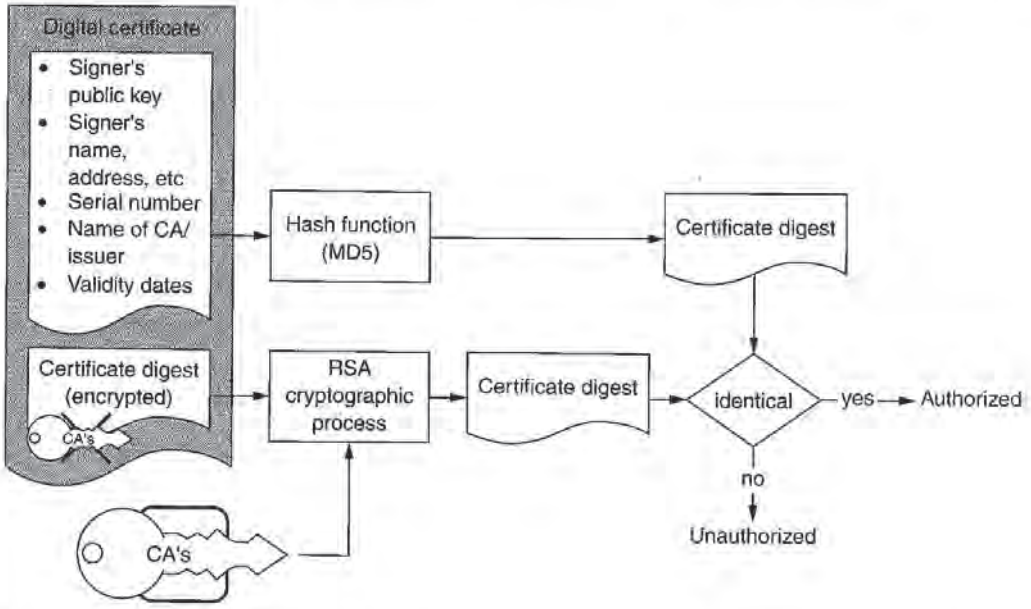


Figure 10.3 How a recipient uses the CA's public key to verify the owner's public key. The key with an "O"behind it represents the certificate authority's public key. The certificate contains the owner's unencrypted public key plus a digest signed with the CA's private key.

Key management: Manages a database of principals (entities such as individual users or groups), their public and private keys, and their certificates.

Identities are real-world entities such as people, companies, or organizations who have a public key. An identity may also have associated one or more certificates authenticating their public key.

Signers are entities who have private keys in addition to corresponding public keys. Private keys can be used for signing. Prior to signing any file, a signer must have a public and private key pair, and at least one certificate authenticating their public key.

Key management provides facilities to:

- import and export certificates
- display information about a certificate
- possibly generate certificates
- sign and verify JAR files.

The key manager must be tamper proof to protect the private keys and certificates it contains. JDK 1.1 provides a key management tool for Solaris and Windows platforms, named javakey; however, no key management facilities are provided with the Symbian OS v6.0 implementation.

PKI: Public Key Infrastructure. The set of hardware, software, people, and procedures needed to create, manage, store, distribute, and revoke certificates based on public-key cryptography (see ***http://dev.baltimore.com/pki/pki.html*** for further details).

Fine-grained security: Or fine-grained access control. By default Java provides two security domains, one for Applets and the other for applications. A Java Applet has very restricted access to a device's resources, for instance it cannot read or write local files, and on Symbian OS v6.0 it cannot access any JavaPhone APIs. A Java application, on the other hand, has complete system access within the constraints imposed by the Java specification, so arbitrary files can be read and written. This approach to security is too coarse for many applications. Fine-grained security allows more control over what an application can and cannot do. For instance, I can allow a certain application to create a file in a specified directory. More specifically, the rights that are granted to an application depend on where it was loaded from and its associated certificates.

10.2.1 Putting it Together

Table 10.1 shows how our building blocks are used to achieve the five key requirements for a secure system.

Table 10.1 How the building blocks are used to achieve the five key requirements for a secure system

	Authentication	Integrity	Confidentiality	Authorization	Non-repudiation
Encryption			•		
DES			•		
Keys	•				
Message digest		•			
Digital signature	•	•			
Certificates	•				•
Key management	•				
PKI	•				•
Fine grained security				•	

10.3 Java as a Secure Platform

Security was, and is, one of Java's critical design goals, and it's built into Java from the ground up. Features such as the byte code verifier, prevention of random memory accesses, and disallowing inappropriate casting ensure that Java can be used to develop secure systems. Achieving the same watertight security with other languages is difficult, if not impossible.

While the Java language was predicated on security, C and Visual Basic were driven by different criteria, in particular flexibility and freedom. C or C++ do not provide the basic security foundations provided by Java, so there will always be a lower level of confidence in solutions based on these languages. Visual Basic provides features such as VarPtr and ObjPtr which can be used to

access random memory locations, and variant data types can be used to make illegal casts.

10.4 Symbian OS v6.0 Security Capabilities

10.4.1 Java Security

Symbian OS v6.0 implements PersonalJava 3.0 which in turn inherits JDK1.1's security capabilities. However, there are a couple of differences: (i) the access control list package (java.security.acl) is not supported in PersonalJava, simply because PersonalJava is intended for single-user rather than multi-user systems, (ii) in PersonalJava the security classes are in the sun.pjava.security package while in JDK1.1 they are implemented in java.security. With these two caveats, the Symbian OS v6.0 and EPOS Release 5 Java security models are thus identical.

JDK 1.1 includes APIs to create key pairs, message digests, and digital signatures. It also includes abstract interfaces for key management and access control; however, specific APIs to support X.509 v3 certificates and other certificate formats are not included, and access control is fairly limited. DSA is the only built-in digital signature algorithm in JDK1.1, though the framework can easily accommodate other algorithms such as RSA. MD5 and SHA message digest algorithms are provided.

The JDK1.1 security architecture allows different cryptography providers to provide their own algorithms (SHA and MD5/RSA are examples of a hash algorithm and a signature generation algorithm respectively), and also to provide implementations for a particular algorithm. So a signature generated with one provider's MD5/RSA implementation can be verified with a second provider's MD5/RSA implementation.

Java security on Symbian OS v6.0 can only be partial because the underlying platform is itself insecure. All Java applications are trusted and can thus access any data file (as can any C++ application). For instance a Java or C++ application could replace the Java VM with its own. The class path is also unrestricted so that system classes, except for those that are preloaded, can be replaced. Nonetheless, Java Applets are secure, though at the price of limiting their functionality.

10.4.2 Extending Security Capabilities

Although PersonalJava 3.0 does not implement key management it does provide abstract classes that a cryptography provider can use to create their own key store (Figure 10.4).

Identity

The Identity class represents identities: as we saw earlier these are real-world objects such as people, companies, or organizations whose identities can be

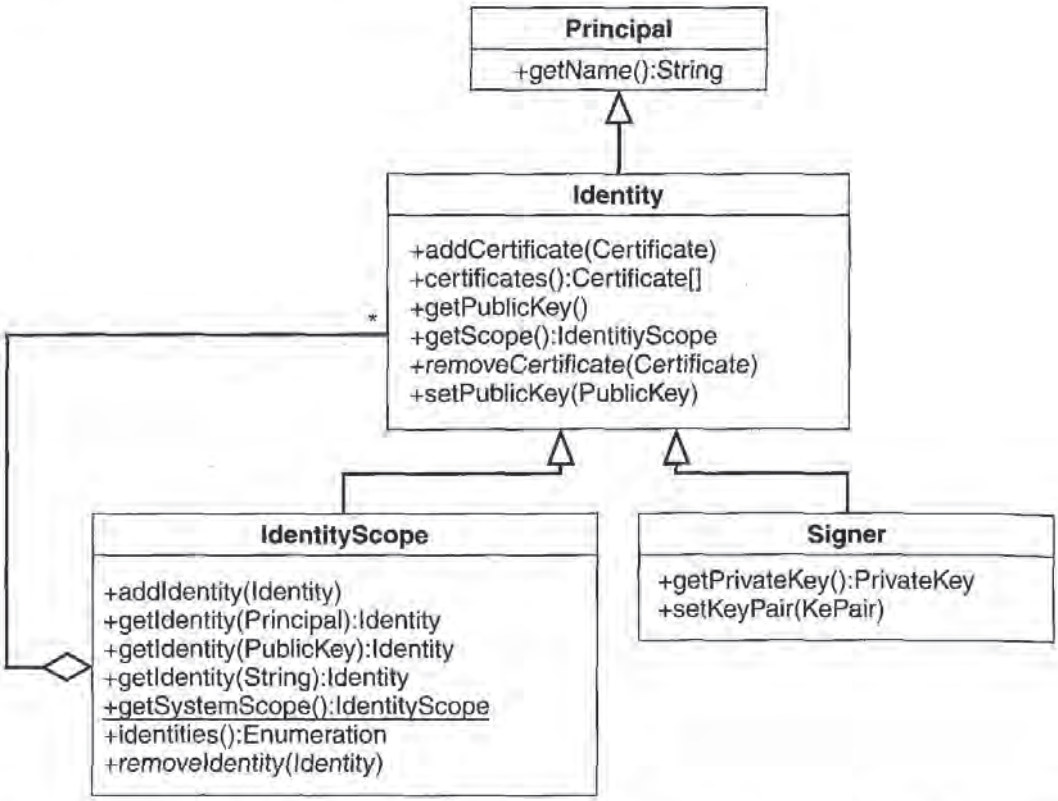


Figure 10.4 The Java key management classes. All four classes are abstract. Only the principal methods are listed.

authenticated using their public keys. Identities may also be more abstract (or concrete) constructs, such as demon threads or smart cards.

All Identity objects have a name and a public key. Names are immutable.

An Identity also has a set of certificates (all certifying its own public key). The Principal names specified in these certificates need not be the same, only the key.

Given an Identity we can obtain its name, its public certificate, and its IdentityScope, i.e. a reference to the object in which the Identity is stored.

Signer

The Signer class extends the Identity class.

A Signer represents identities that can be used to digitally sign data. As such, they contain private keys and must therefore be stored securely.

A Signer has methods to set its KeyPair and obtain its private key.

IdentityScope

The IdentityScope class is a subclass of the Identity class.

An IdentityScope serves as a general abstraction for repositories of Identity objects. The actual repository might be a database, SIM card, server, PGP key

ring, or a file. These repositories are used by various mechanisms, such as secure class loaders, to verify classes and assign permissions, or by signing tools to retrieve private keys and generate digital signatures. It is possible, and in fact expected, that different types of IdentityScope will apply different policies for operations on the various types of Identity objects.

No two Identity objects in the same IdentityScope can have the same public key or the same name. Similarly, since there is only one private key per public key, private keys must also be unique.

An IdentityScope can contain Identity objects of all kinds, including other IdentityScopes. The IdentityScope class provides methods to retrieve, add, and remove all types of Identity objects and their subclasses.

A default IdentityScope is supplied by the SUN cryptography provider. This manages a repository of keys, certificates, and trust levels and is available to applications for authentication or signing purposes. An instance of this class is created every time a Java program is run or an Applet viewer is started. This can be retrieved using `IdentityScope IdentityScope.getSystemScope()`.

In general, public key repositories should be kept separate from private key repositories. On a WID this might mean that private keys should be stored on the SIM card.

10.4.3 Using HTTPS to Access a Secure Site

The HTTPS protocol allows information to be transferred securely between a client and a secure website using the SSL protocol. Java on Symbian OS v6.0 provides HTTPS support via an HTTPS URL. This is implemented by the underlying EPOC SSL functionality and is independent of the security APIs in Java.

The following code segment shows how a secure site can be accessed. It copies the contents of the site specified by the String address into the file **temp.txt**:

```
String address = "https://<...>";
URL url = new URL(address);
BufferedInputStream inStream = new
BufferedInputStream(url.openStream());
FileOutputStream outStream = new FileOutputStream ("temp.txt");
while(true)
{
     int data = inStream.read();
     if(data == -1) break;
     outStream.write(data);
}
inStream.close();
outStream.flush();
outStream.close();
```

If you are using this code from the Emulator with the HTTPS protocol you need to ensure that you are using the Emulator's HTTP stack and not the Window's stack. The latter is the default; however, this will generate an IOException with the error "SimpleHttpsSocket is not open" if you attempt an HTTPS transfer.

The Emulator therefore needs to be configured to use a dial-in connection. This can be achieved either by using a serial line modem and RAS, or by using the IR port and a mobile phone. A few things to remember:

- change the name of **winnet.dll** in **\epoc32\release\wins\urel** or **winnet_g.dll** in **\epoc32\release\wins\udeb**. This will ensure that the EPOC stack is used
- ensure the emulator is configured with an Internet provider
- the code should be run as part of a windowed application, rather than as a console application

10.4.4 Native Security

Symbian OS v6.0 has added significant native security features to the platform. These are outlined here for interest; however, they are not accessible to Java developers. The new security functions include standard cryptography algorithms, hash key generation, random number generation, and a certificate store. Symbian is supporting PKI for certificate management, verification, and revoking.

The two principal modules are the cryptography module and the certificate management module.

The cryptography module includes the following components:

- raw cryptography algorithms allowing data to be encrypted and decrypted. The supported symmetric ciphers are DES, 3DES, RC2, RC4, RC5. The supported asymmetric ciphers are RSA, DSA, DH
- hash functions to support message digests: MD5, SHA (SHA1), HMAC
- a random number generator, the basis for the cryptographic key generation.

The certificate management module provides the following services:

- storing and retrieving certificates
- assigning trust status to a certificate on an application-by-application basis
- certificate chain construction and validation
- verifying a certificate.

The certificate management module includes an API for use by any Symbian or licensee client requiring its services; however, it is not a public API. One important client is the certificate management control panel application. This provides a user interface for configuring data used by the certificate

management component, including trusted root certificates and trust status of each certificate on an application-by-application basis.

Support is initially limited to X.509 certificates along with a PKIX certificate usage profile. The architecture allows for other certificate formats and profiles to be added.

10.5 The Thumbprint Example

The Thumbprint example illustrates the Personal Java security APIs. The application allows a key pair to be generated, a signature for a file to be created using a private key, and a file to be verified using a public key and the file's signature. The example is written for the Crystal reference design though it can easily be modified for other platforms.

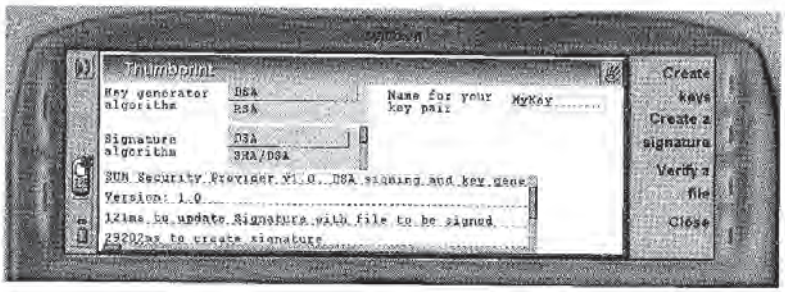


Figure 10.5 The Thumbprint example running under the Crystal emulator.

Thumbprint is intended to give a flavor of what can be achieved on Symbian OS v6.0. An online tutorial covering the JDK1.1 security APIs can be found at ***http://web2.java.sun.com/docs/books/tutorial/security1.1/index.html*** and further details of the architecture and APIs are provided with the JDK1.1 installation (***docs/guide/security/CryptoSpec.html***).

10.5.1 Setup

The application is created in the com.symbian.devnet.crystal.thumbprint package. The implementation for the security APIs is in sun.pjava.security, so we need to import this package (the APIs are defined in java.security). As we are creating an application for the Crystal reference design, our Thumbprint class extends CFrame and implements the CBAListener.

```
package com.symbian.devnet.crystal.thumbprint;
import com.symbian.devnet.crystal.awt.*;
import com.symbian.epoc.awt.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import sun.pjava.security.*;
```

```
class Thumbprint extends CFrame implements CBAListener
{
    private static CBAHandler cba; // Handles the CBA buttons
    static final int CREATE_KEYS
        = EikCommandButtonGroup.BUTTON1;
    static final int CREATE_SIG
        = EikCommandButtonGroup.BUTTON2;
    static final int VERIFY = EikCommandButtonGroup.BUTTON3;
    static final int CLOSE = EikCommandButtonGroup.BUTTON4;
}
```

`keyAlgorithmC`, `keynameF`, and `signatureAlgorithmC` are components used to select a key generation algorithm, choose a name for the key pair, and to select an algorithm for creating a signature.

`publicDir` and `privateDir` define directories used to store public and private keys respectively:

```
private static List keyAlgorithmC = new List(2);
private static TextField keyNameF = new TextField("MyKey", 12);
private static List signatureAlgorithmC = new List(2);
private static GridPanel mainPanel = new GridPanel();
private static TextArea statusF = new TextArea(6, 60);
private static File publicDir = new File("PublicKeys\\ ");
private static File privateDir = new File("PrivateKeys\\ ");
```

10.5.2 The Constructor

The constructor sets up the UI. The Security class is used to obtain information about the available cryptography providers, which is then displayed using `println()`. The only provider available to us in Symbian OS v6.0 is SUN:

```
Thumbprint()
{
    setTitle(Version.TITLE);
    aboutDialog.setTitle("About " + Version.TITLE);
    aboutDialog.text = Version.DIALOG_TEXT;

    GridPanel mainPanel = new GridPanel();
    mainPanel.constraints.anchor = GridBagConstraints.WEST;
    mainPanel.constraints.weightx = 1.0;

    Provider[] providers = Security.getProviders();
    for(int i=0; i < providers.length; i++)
    {
        println("Provider's name: " + providers[i].getName());
        println(providers[i].getInfo());
        println("Version: " + providers[i].getVersion());
    }
}
```

The security APIs don't provide methods for obtaining the available algorithms. Standard names are defined for the algorithms:

- key pair generation algorithms: RSA, DSA
- message digest algorithms: SHA-1 (or SHA), MD2, MD5
- signature algorithms, where the key-pair is DSA: DSA (also known as SHA-1/DSA)
- signature algorithms, where the key-pair is RSA) MD2/RSA, MD5/RSA, SHA-1/RSA.

We use these names to populate the UI components used to choose an algorithm; however, the SUN cryptography provider supports only DSA:

```
mainPanel.add(new Label("Key generator\nalgorithm"), 0, 0);
keyAlgorithmC.add("DSA");
keyAlgorithmC.add("RSA");
keyAlgorithmC.select(0);
mainPanel.add(keyAlgorithmC, 1, 0);

mainPanel.add(new Label("Name for your\nkey pair"), 2, 0);
mainPanel.add(keyNameF, 3, 0);

mainPanel.add(new Label("Signature\nalgorithm"), 0, 1);
signatureAlgorithmC.add("DSA");
signatureAlgorithmC.add("MD2/RSA");
signatureAlgorithmC.add("MD5/RSA");
signatureAlgorithmC.add("SHA-1/RSA");
signatureAlgorithmC.select(0);
mainPanel.add(signatureAlgorithmC, 1, 1);
```

Create the directories used to store the key pairs we generate, then set up the remaining UI components including the CBA buttons:

```
if(!publicDir.exists())
{
     if(!publicDir.mkdir()) println("Failed to create "
          + publicDir.getAbsolutePath());
}
if(!privateDir.exists())
{
     if(!privateDir.mkdir()) println("Failed to create "
          + privateDir.getAbsolutePath());
}

mainPanel.constraints.gridwidth
   = GridBagConstraints.REMAINDER;
mainPanel.add(statusF, 0, 2);
```

```
        cba = new CBAHandler(this);
        cba.setText(CREATE_KEYS, "Create\nkeys");
        cba.setText(CREATE_SIG, "Create a\nsignature");
        cba.setText(VERIFY, "Verify a\nfile");
        cba.setText(CLOSE, "Close");
        cba.activate();

        add(mainPanel);
        show();
    }
```

10.5.3 openFile()

`openFile()` is a utility method. It creates a file dialog with the given title and displaying the given directory. The method returns a buffered input stream or null if the user didn't select a file:

```
    private BufferedInputStream openFile(String title, String
        directory)
    {
        try
        {
            FileDialog fileDialog = new FileDialog(this, title);
            fileDialog.setDirectory(directory);
            fileDialog.show();
            return new BufferedInputStream(new FileInputStream(new
               File(fileDialog.getDirectory(),
               fileDialog.getFile())));
        }
        catch(FileNotFoundException fnfe)
        {
            println("Couldn't open " + directory);
            return null;
        }
        catch(NullPointerException npe)
        {
            return null;
        }
    }
```

10.5.4 updateSignature()

`updateSignature()` updates the specified Signature from the specified InputStream. The InputStream refers to the file to be signed or verified:

```
    private void updateSignature(Signature dsa, InputStream file)
        throws IOException, SignatureException
    {
        byte[] data = new byte[128];
        int count = 0;
        while(true)
```

```
        {
            count = file.read(data);
            if(count <= 0) break;
            dsa.update(data, 0, count);
        }
        file.close();
    }
```

10.5.5 Handling User Input

`cbaActionPerformed()` handles user input from the four CBA buttons, and as such does most of the work. It creates key pairs, creates the signature for a file, and verifies a previously signed file. Most of the activities are timed using `System.currentTimeMillis()`:

```
public void cbaActionPerformed(CBAEvent cbae)
{
    try
    {
        switch(cbae.getID())
        {
```

The user has pressed the Create keys button. `KeyPairGenerator.getInstance(String algorithm)` returns a KeyPairGenerator, `keyGen`. The user selects the name of the algorithm using the `keyAlgorithmC` List component. The KeyPairGenerator is initialized with the desired strength of the key (1024 in this case) and a random number generator. A KeyPair instance, `pair`, is obtained using `KeyPairGenerator.generateKeyPair()`. The private and public keys are then obtained from the KeyPair:

```
case CREATE_KEYS:
    long startTime = System.currentTimeMillis();
    KeyPairGenerator keyGen = KeyPairGenerator.
        getInstance(keyAlgorithmC.getSelectedItem());
    keyGen.initialize(1024, new SecureRandom());
    KeyPair pair = keyGen.generateKeyPair();
    PrivateKey privateKey = pair.getPrivate();
    PublicKey publicKey = pair.getPublic();
    println("" + (System.currentTimeMillis() -
       startTime) + "ms to create key pair");
```

The public and private keys are written to the relevant directories. Using an ObjectOutputStream serializes the keys:

```
ObjectOutputStream oos = new ObjectOutputStream(new
   FileOutputStream(new File(publicDir,
     keyNameF.getText())));
oos.writeObject(publicKey);
oos.close();
```

```
            oos = new ObjectOutputStream(new FileOutputStream(new
               File(privateDir, keyNameF.getText())));
            oos.writeObject(privateKey);
            oos.close();
        break;
```

The user has pressed the Create signature button. We get an InputStream for the file the user wants to sign and an ObjectInputStream for the private key they want to use to sign it. The PrivateKey is deserialized from the ObjectInputStream:

```
        case CREATE_SIG:
            InputStream file = openFile("Select the file to be
               signed", "j:/examples/thumbprint/");

            ObjectInputStream ois = new ObjectInputStream(openFile
               ("Select a private key",
                privateDir.getAbsolutePath()));
            privateKey = (PrivateKey)ois.readObject();
```

`Signature.getInstance(String algorithm)` returns a Signature, `dsa`. The user selects the name of the algorithm that will sign the requested file using the `signatureAlgorithmC` List component. We then use our utility method `updateSignature()` to update the Signature with the file to be signed:

```
            startTime = System.currentTimeMillis();
            Signature dsa = Signature.getInstance
                (signatureAlgorithmC.getSelectedItem());
            dsa.initSign(privateKey);
            updateSignature(dsa, file);
            println("" + (System.currentTimeMillis() - startTime)
                + "ms to update Signature with file to be signed");
```

Create the signature, `sig`, as a byte array:

```
            startTime = System.currentTimeMillis();
            byte[] sig=dsa.sign();
            println("" + (System.currentTimeMillis() - startTime)
               + "ms to create signature");
```

Save the file's signature to the specified location:

```
            FileDialog fileDialog = new FileDialog(this,
                   "Save the file's signature as...",
                   FileDialog.SAVE);

            fileDialog.setDirectory("j:/examples/thumbprint/
                   signature.sig");
                   fileDialog.setFile("filename.sig");
                   fileDialog.show();
```

```
            FileOutputStream signatureFile = new
               FileOutputStream(new File(fileDialog.
               getDirectory(), fileDialog.getFile()));
            signatureFile.write(sig);
            signatureFile.flush();
            signatureFile.close();
        break;
```

The user has pressed the Verify button, so we select the file to be verified, select and read in the correct public key, create a suitable Signature, read the file into the Signature, read in the file's signature, and finally verify the file against the signature:

```
        case VERIFY:
```

Select the file to be verified and read in the public key:

```
            file = openFile("Select the file to be verified",
               "j:/examples/thumbprint/");
            ois = new ObjectInputStream(openFile("Select the
               public key",
               publicDir.getAbsolutePath()));
            publicKey = (PublicKey)ois.readObject();
```

Initialize the Signature and update it with the data to be verified:

```
            startTime = System.currentTimeMillis();
            dsa = Signature.getInstance(signatureAlgorithmC.
                getSelectedItem());
            dsa.initVerify(publicKey);
            updateSignature(dsa, file);
            println("" + (System.currentTimeMillis() -
                startTime) + "ms to
                update Signature with file to be verified");
```

Read in the file's signature (we assume it is 512 bytes or less) and copy it to an array of the correct size:

```
            InputStream sigFile = openFile("Select the file's
                signature",
               "j:/examples/thumbprint/signature.sig");
            byte[] tempSig = new byte[512];
            int size = sigFile.read(tempSig);
            sig = new byte[size];
            System.arraycopy(tempSig, 0, sig, 0, size);
```

Verify the signature and display the result:

```
            boolean verifies = dsa.verify(sig);
            println("Signature verifies: " + verifies);
        break;
```

The user has pressed the Close button, so shut down the application:

```
                case CLOSE: // Close the program
                    super.shutDown();
                break;
                default:
            }
        }
        catch(Exception e)
        {
            println("Exception: " + e);
            e.printStackTrace();
        }
    }
```

10.5.6 Utilities

`println()` is a utility for displaying text in a TextField, statusF:

```
    private static void println(String text)
    {
        statusF.append(text + "\n");
    }
```

`main()` simply creates an instance of our Thumbprint class:

```
        public static void main(String[] args)
        {
            new Thumbprint();
        }
    }
```

10.5.7 Measuring Performance

The application gives an idea of how long the various cryptographic functions take. Under the emulator (which is faster than actual hardware), creating the signature for a 2 KB file took just under 30 seconds. Verification, however, was much quicker, requiring only a few milliseconds.

10.6 Security Management in the Future

Future versions of Symbian's Java implementation will either be based on the Connected Device Configuration which brings with it fine-grain security and improved key management functionality, including support for X.509 v3 certificates, or MIDP-NG if this provides a suitable security model.

We can't provide a truly secure Java environment if the native platform is itself insecure; however, we don't necessarily need native fine-grained security.

The following outlines how a secure execution environment might be provided in the future. This is to illustrate the concepts behind a secure system;

it is not meant to indicate how a secure environment will be implemented in practice:

- Java code will be run using the Java security manager and subject to FGS
- the native loader is modified to check the code source of all other code that is run (dlls, .exe, .app, etc)
- non-Java code that comes from Symbian or the relevant licensee will be allowed to run. Non-Java code from other sources will not be allowed to run. The executable's signature will be used to make the decision
- JNI won't be supported as it would require full native FGS (this could be a significant restriction)
- it might be possible to run ROM code without a signature; however, we need to distinguish original ROM code from patched code. Furthermore, in many cases code is running from flash, not ROM
- as far as possible, applications are also checked during installation. However, this cannot be relied on as there are too many ways in which executable code can find its way onto a device: copying, email attachments, beaming, etc.

11

Provisioning Java Services

Jonathan Allin

Provisioning, along with security, represents perhaps one of the biggest challenges in enabling the widespread uptake of wireless services.

11.1 Introduction to Provisioning

Provisioning covers a number of concepts that relate to an application's life cycle:

- service discovery: how does the user discover what services or applications are available? This might be through a web or WAP portal, or through a Bluetooth discovery service
- service delivery: applications might be downloaded by a web or WAP browser, via Bluetooth, or an application might be beamed across from another device or copied down from a desktop computer
- service installation: having delivered a service, it must be installed on the device so that it can be run by the user, preferably in the same way as a native application. The service may need to be registered with the device and perhaps also with the service provider
- service update: establishing the current version of a service that is already on a device and delivering upgrades as required
- service removal: it must be possible to remove an unwanted service
- push capability: operators and some providers may require the ability to push applications onto a device. This might be to deliver an upgrade to the operating system or so that an operator can ensure that all their devices have the same set of applications
- device variability: how to target a wide range of devices such as feature-phones, Smartphones, and Communicators, with wide variations in each of these three categories. Devices vary in screen size, input methods,

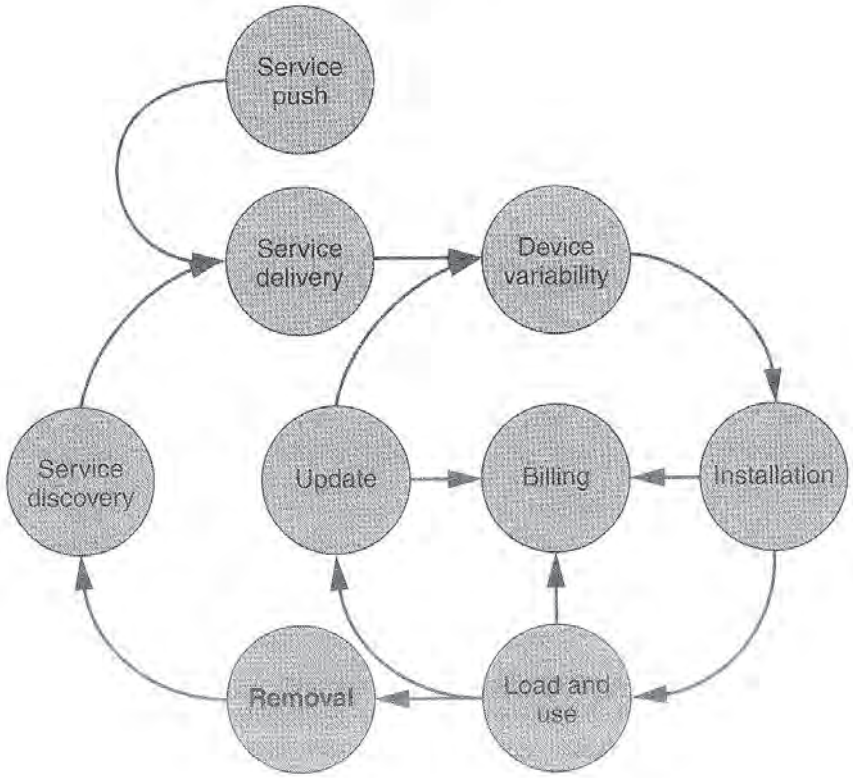


Figure 11.1 The provisioning "life cycle".

processing power, memory, and the APIs that they support. The application server needs to establish a device's features and capabilities and so deliver an appropriate application variant. We may want to make only suitable applications visible to a device

- billing: how the user pays for a service. This might be just once when the application is installed, or it might be as the service is used, for instance every time the user requests their location or reaches the next level in a game.

As we will see in this chapter, a complete solution does not yet exist, but we're getting close. We'll look at a number of technologies that address various parts of the over- the-air (OTA) provisioning lifecycle.

11.2 Applets

Applets provide the simplest way to deliver a Java application to a device: the user simply clicks or activates a link in their Web browser. This partially addresses discovery, and also installation and removal to the extent that Applets are essentially short-lived, temporary applications. However, by themselves Applets can't adapt to the client device, nor do they provide a billing mechanism.

11.3 The MID Profile Recommended Practice Document

The MIDP working group has created a document called "Over The Air User Initiated Provisioning for the Mobile Information Device Profile." This is a recommended practice document, which describes a provisioning model for MIDP applications. This is not part of the MIDP 1.0 specification, but is likely to be incorporated into the MIDP-NG specification. We will look at the MID Profile in Chapter 14. The document covers a number of our provisioning requirements with varying degrees of completeness. It addresses service discovery, installation, update, and removal, but does not address application push or billing, and only partially addresses device variability.

The description in this section is based on version 0.7 of "Over The Air User Initiated Provisioning for the Mobile Information Device Profile." Sun expects at least one more version before it's finalized, so check ***http://java.sun.com/aboutJava/communityprocess/jsr/jsr_118_midpng.html*** for the latest version.

MIDP applications are called MIDlets, and like Applets, are typically discovered through a Web or WAP browser. However, while Applets are temporary and run within the browser, MIDlets are installed onto a device and are generally launched in the same way as native applications. MIDlets are delivered in a JAR file. This contains one or more MIDlets and associated resources (images, sound files, localization resource bundles), which together make up a MIDlet suite.

The MIDlet provisioning model is designed to create a simple and flexible way in which MIDlet suites can be discovered by a user and easily downloaded to their WID. It describes the behavior that must be provided by the WID and the functions that support the MIDlet suite life cycle. This life cycle consists of discovery, installation, update, invocation, and removal.

The user can control the download, determine which versions of software are installed, remove MIDlet suites once downloaded, and find out about the MIDlet suites on their device. For any lengthy operation the user can monitor progress and if necessary cancel the operation. Once downloaded, MIDlets within the suite can be run by the user.

11.3.1 Discovery

The WID itself must provide the mechanism that allows users to discover or be notified of MIDlet suites that can be loaded onto the device. Discovery may be via the device's Web or WAP browser, or by a resident application written specifically to identify appropriate MIDlet suites.

A user doesn't access the MIDlet JAR file directly, but will instead initiate installation by accessing a MIDlet descriptor file (this is also known as a Java Application Descriptor file, or JAD). The descriptor is a small file that contains a description of the MIDlet suite, including its size, version number, and who

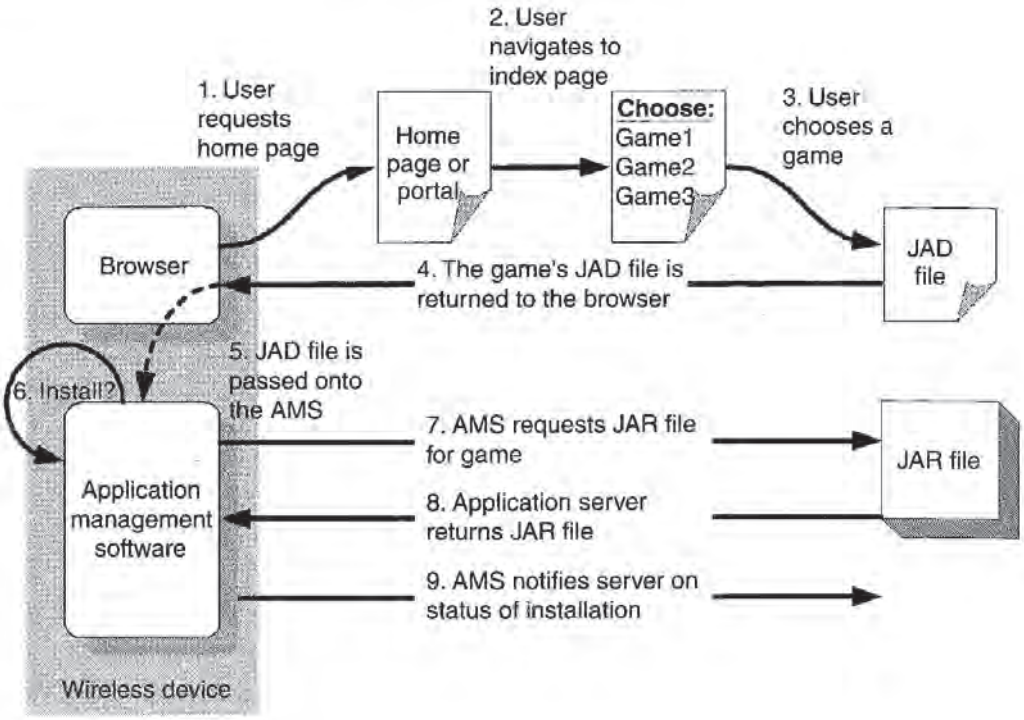


Figure 11.2 The basic MIDlet provisioning model.

provided it. The user can thus decide whether or not they want a particular MIDlet suite without having to download and install it first.

11.3.2 Installation

The installation process starts when the browser requests an application descriptor file. To help select applications which are appropriate for the device the browser provides the server with information about the device's capabilities: the HTTP request headers used to fetch an application include, as appropriate: `User-Agent`, `Accept-Language`, and `Accept`. Servers should use this additional advisory information to select the appropriate application descriptor and application JAR file for the device. For example, the request headers for a descriptor file might look like this:

```
GET http://www.cardsrus.com/app-dir/game.jad HTTP/1.1
Host: www.cardsrus.com
Accept: text/vnd.sun.j2me.app-descriptor
User-Agent: MIDP/1.0 CLDC/1.0 Nokia9210/7.29
Accept-Language: en-US, fi, fr
Accept-Charset: utf-8
```

The `User-Agent` identifies the device and its capabilities (i.e., MIDP 1.0 and CLDC 1.0). The `Accept` header must be included when installation is proxied via a WAP gateway.

The server's response headers might be:

```
HTTP/1.1 200 OK
Server: GamesServer/1.3.12
Set-Cookie: Name="cardsrusgames"; jsession="1321341324";
    VERSION="1"
Content-Length: 2345
Content-Type: text/vnd.sun.j2me.app-descriptor; charset=utf-8
```

Any `Set-Cookie` header should be treated as a session token. Cookie attribute names and values are remembered so they can be used when the installation status is reported or in a re-send of the request with authentication credentials. Client devices must be able to store a single cookie for the duration of the installation process. The cookie should then be discarded once the status report has been sent.

The application descriptor returned by the server is passed to the device's application management software which will then request the application's JAR file. The request headers might look like this:

```
GET http://www.cardsrus.com/app-dir/game.jar HTTP/1.1
Host: www.cardsrus.com
Cookie: Name="cardsrusgames"; jsession="1321341324"
User-Agent: MIDP/1.0 CLDC/1.0 Nokia9210/7.29
Accept-Language: en-US, fi, fr
Accept: application/java, application/java-archive
```

The response headers from the server might look as follows:

```
HTTP/1.1 200 OK
Server: GamesServer/1.3.12
Content-Length: 25432
Content-Type: application/java
```

Once a MIDlet suite is installed, a notice can be sent to the server to indicate successful installation. The server may use this information for accounting, billing, or other customer service needs. The descriptor file specifies a URL to which install status information is to be reported. The WID `POSTs` an `HTTP` request to this URL that indicates whether installation was successful, or if not the reason for failure e.g. lack of memory, the user cancelled installation, the descriptor file was invalid, or the JAR file was invalid).

For example, if the descriptor file contained:

```
MIDlet-Install-Notify:
http://www.cardsrus.com//status?games/cardgames.jar
```

and the application was successfully installed, the following would be posted (note that any cookie fields must also be included):

```
POST http://www.cardsrus.com//status?games/cardgames.jar HTTP/1.1
Host: www.cardsrus.com
Cookie: Name="cardsrusgames"; jsession="1321341324"
900 Success
```

The user will be able to use the application or service even if the server doesn't respond to this installation status report.

11.3.3 Authorization

The MIDP recommended practice document supports limited authorization, with the application management software at least able to support the RFC2617 Basic Authentication Scheme. When a device requests a JAR file the server may respond with a request for authentication. The device should then resubmit the request, together with the requested credentials. Typically the credentials will be a name and password entered by the user in a dialog.

11.3.4 Update

If the user tries to reinstall an application the device's application management software will tell the user whether it is an older or newer version and request confirmation before proceeding.

11.3.5 Removal

A user is prompted for confirmation when they want to remove a MIDlet suite. If the application descriptor includes the attribute `MIDlet-Delete-Confirm` then its value is included in the prompt. This will allow the application provider to highlight any specific conditions that might arise when the MIDlet suite is removed.

11.3.6 Provisioning MIDlets in a WAP Environment

Most of what we've discussed is applicable to both Web and WAP provisioning. However, there are few additional considerations for WAP clients.

If we have a Web client, provisioning will use HTTP end to end. However, if we have a WAP client it will use the WSP protocol to talk to a WAP gateway, though the provisioning server will still generally use HTTP.

Only connection-oriented WSP and only the following WAP protocol stack configurations are supported:

WSP/UDP/IPv4/PPP/CSD;

WSP/UDP/IPv4/GPRS.

SMS and USSD bearers are not supported.

Request headers must include `Accept: */*` so that any data type can be received from the WAP Gateway.

11.3.6 MIDlet Attributes

Attributes relating to provisioning are defined by both the MIDlet suite's descriptor file and by its JAR file manifest.

The JAR file manifest must contain attributes which specify:

the name, version, vendor, icon, and description of the MIDlet suite;

the name and icon of each MIDlet in the MIDlet suite, and also the name of the main class for each MIDlet (the main class will extend javax.microedition.midlet.MIDlet);

the J2ME profile (e.g. "MIDP 1.0") and configuration (e.g. "CLDC-1.0") required to run the MIDlet suite.

The JAR file manifest may in addition contain attributes that specify:

a description of the MIDlet suite and an icon that represents the suite

a URL which will take the user to a site containing further information on the MIDlet suite

the minimum amount of persistent storage required by the MIDlet suite (MIDlets in a suite share the same storage in the Record Management System).

The MIDP 1.0 specification gives the following example of a JAR file manifest:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: CardsRUS
MIDlet-1: Solitaire, /Solitare.png, com.cardsrus.org.Solitare
MIDlet-2: JacksWild, /JacksWild.png, com.cardsrus.org.JacksWild
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
```

The MIDlet descriptor file contains attributes which specify:

the name, version, and vendor of the MIDlet suite

the URL from which the JAR can be loaded

the size of the JAR file (in bytes).

the MIDlet descriptor file may additionally contain attributes which specify:

a description of the MIDlet suite and an icon that represents the suite

a URL that can provide more information about the MIDlet suite

the URL to which a POST request is used to confirm successful installation of this MIDlet suite; this must contain enough information to identify the application

a text message presented to the user when they are prompted to confirm deletion of this MIDlet suite

the minimum amount of persistent storage required by the MIDlet suite (MIDlets in a suite share the same storage in the Record Management System).

Here's an example of a descriptor file:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: CardsRUS
MIDlet-Jar-URL: http://www.cardsrus.com/games/cardgames.jar
MIDlet-Install-Notify: http://www.cardsrus.com//status?games/
    cardgames.jar
MIDlet-Jar-Size: 7378
MIDlet-Data-Size: 256
```

The MIDlet suite name, version, and vendor attributes must be duplicated in the descriptor and manifest files. If they are not identical the JAR file won't be installed. While duplication of other attributes is not required, if they are duplicated their values might differ. In this case the value from the descriptor file will override the value from the manifest file.

At runtime a MIDlet can access the attributes in the MIDlet descriptor file or JAR file using the `javax.microedition.midlet.MIDlet.getAppProperty()` method. It can obtain information such as color depth or screen size using the javax.microedition.lcdui.Display class or javax.microedition.lcdui.Canvas class respectively.

11.4 The Java Services Provisioning Manager

The recommended practice document we discussed above defines a simple application pull model for provisioning MIDP devices. However, operators and service providers want to provide content to diverse clients, who will use differing markup languages such as WAP, cHTML, xHTML, XML, or custom markup/protocols.

The content provider needs to determine which applications and services to display to a user in a context-sensitive manner, and then deliver those services to the client. During this process there are opportunities to bill for, advertise, and track these services and applications.

This creates the need for an end-to-end provisioning system that is adaptable, scalable and customizable, and not dependent on a particular remote API. Sun and various partners are working on such a model: as of writing, Sun doesn't

yet have a name for it, though Java Services Vending Machine has been suggested. For the moment we'll call it the Java Services Provisioning Manager, or JSPM.

The JSPM manages the server-side aspects of provisioning Java applications and services to both wireless and tethered J2ME clients. It processes requests from the client to determine which Java applications will be displayed to the client, and it responds to HTTP requests from the client to deliver the appropriate JAR files. The JSPM is markup-language neutral: it may use XML internally, and then use pluggable provisioning adapters to interact with clients with varying markup systems such as WML, HTML, cHTML, XML, or xHTML. And of course the JSPM will support any client that implements the MIDP recommended practice document.

The result is that content providers can focus on content and service rules, without having to worry about multiple markup languages.

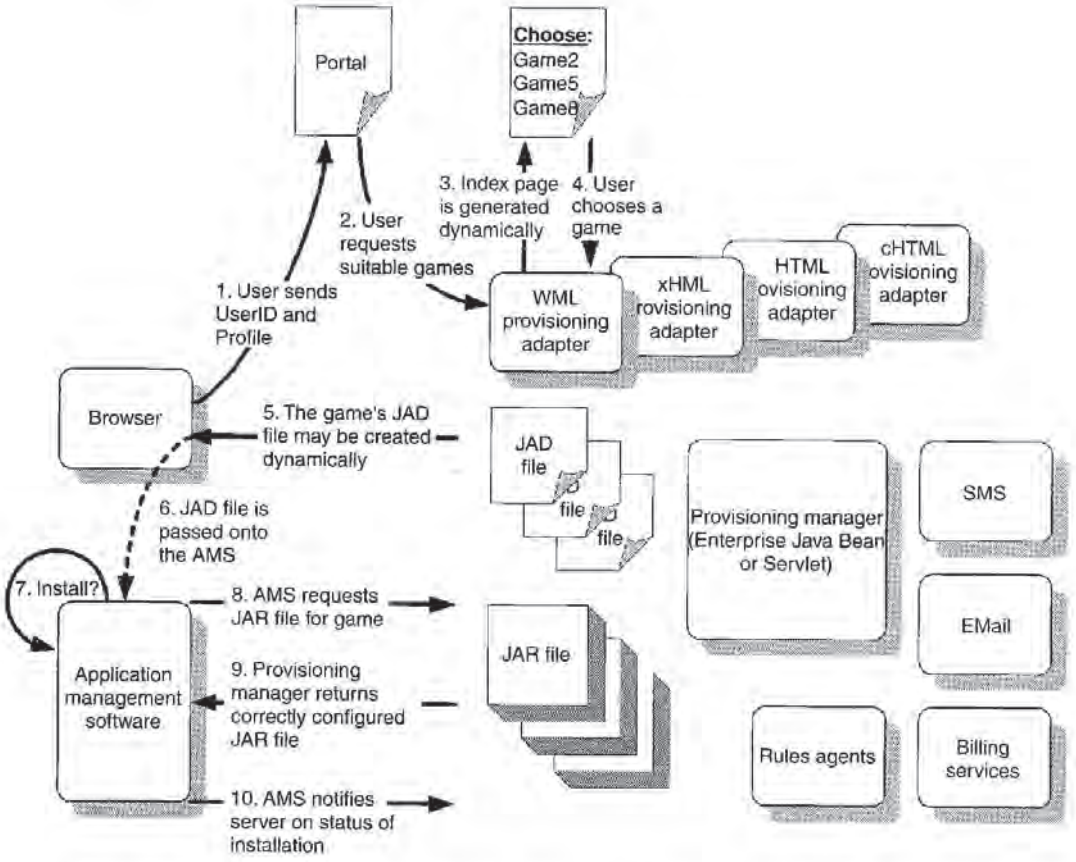


Figure 11.3 Accessing the Java Services Provisioning Manager from a MID device.

The provisioning manager determines what content should be made available to the user based on their type of device, location, and other knowledge of the user held by the provisioning manager. It then creates a

suitable index page of Java applications, in XML format, which it returns to the WML provisioning adapter. The WML adapter converts this page to WML and dispatches it to the device's browser.

Rules agents contain information that will determine what applications will be displayed to a user at a given time, such as packaging subscription information, location information, application placement order rules, advertising information, or pricing information.

Application variants are generated to match the client device, and used to dynamically create a JAR file with, say, the correct UI components.

11.4.1 Some Client Interaction Scenarios

Selecting a Game

Jon Bull starts his WID's browser, surfs to his home or portal page, and logs on using his user Id and password. He selects the "My Java games" link, which points to the JSPM.

The JSPM uses the user Id to look up Jon's user profile information. It then determines what games services Jon can access by using information in the user profile such as subscription services, phone model information from the `USER-AGENT` field of the client request, and information from the rules agent. The JSPM then generates this information as an XML document with the correct style sheet and passes it to the WML adapter for rendering by Jon's browser.

Jon selects a game from the generated page. The JSPM may receive the request for the descriptor file or the selected link may simply point directly to the file system. If it is the former, the JSPM will respond with the MIME-typed content and may log the event. The browser will pass the MIME content to the device's AMS.

Jon's AMS responds with an HTTP transaction to request the JAR file. The JSPM may receive this request (depending on the contents of the descriptor file it created), or the request may simply point directly to the file system. If the former, the JSPM will respond with the JAR file for the game and may log the event.

Jon successfully downloads the game. The descriptor file for the game contains an `APP-Install-Notify` attribute that references the JSPM, the JSPM accepts the HTTP request and processes the information in the request, which includes generating a bill through the billing interface.

Finally, the JSPM sends an SMS to Jon confirming that he has been billed.

Accessing Banking Services

Nikki Horrocks uses her WID to select "banking applications." A rules agent finds out her location and preferences and accesses the bank's advertising

content. It passes back to the provisioning manager the banking applications it knows about along with locations of banks near her, together with an advertisement for the nearest bank.

The JSPM formats this information as XML and passes it to the WML provisioning adapter for display by her WID's browser.

11.4.2 JSPM Developers

A number of companies are working with Sun to develop JSPM technologies. This includes Art Technology Group (***http://www.atg.com***), Mobilitec (***http://www.mobilitec.com***), and Ellipsus (***http://www.ellipsus.com/***).

11.5 Mobilitec

Mobilitec's technology covers content aggregation, provisioning, billing, and middleware. Applications can be downloaded over the air, though not necessarily through a browser. Their provisioning architecture is aligned with the JSPM model.

11.5.1 Content Aggregation

Content aggregation is about the workflow required to bring content together from disparate sources so that it can be accessed from a single web portal. It covers:

submission of content

approval and certification, database storage

publication: making the content available to subscribers.

11.5.2 Provisioning Capability

Provisioning covers discovery and transmission of a service or application (much of this is covered by the JSPM). The process includes:

presenting menus of available categories and applications, formatted to the target device

personalization of the service

adaptation and delivery of applications to the device.

11.5.3 Billing

The billing model allows the operator or service provider to define and implement various billing policies and charging schemes. Each content provider

can specify its own billing policy, which gives the service provider the option of charging on a percentage basis and avoiding complex billing issues.

The model uses an event-based accounting process that can receive events from various sources. Billing events might be:

download events (e.g. after I've installed an application), time events (e.g. I have to renew my license for the service every month), data transmitted events (when data is transmitted, e.g. an updated flight timetable)

events generated by back-end applications

events generated by the application, e.g. when the application performs a calculation, or when I achieve the next level in a game.

The billing events can trigger various runtime rules that form the billing policy. Each rule includes a condition and one or more actions. When the condition is met the actions are performed and as a result a billing record may be sent to the billing system.

11.5.4 Middleware

Mobilitec's middleware products cover tools and APIs for building client–server applications and services, integration with messaging services, and enabling a billing model.

11.6 Java Web Start

Java Web Start from Sun (***http://java.sun.com/features/2001/03/jws1.html***) automates the deployment of Java applications to client devices. Unlike Applets, Java Web Start applications are installed on the client device and are not tied to a browser or the browser's VM.

Applications are launched either from a browser by clicking on a URL, or from a Web Start launcher installed on the device. If the application has not yet been loaded it will be fetched from the application server, otherwise it will be run from the application cache. If the device is online a check will be made to see if the cached version is the latest version; if not, the latest version is fetched from the server and run.

By default, applications run in a restricted sandbox, with limited access to files and networks. The sandbox allows Java applications to request file loads and saves, printing, access to the clipboard, etc., by popping up a dialog. However, an application can request unrestricted access to the system, in which case Java Web Start will show a security warning dialog that provides information about the application vendor. If you choose to trust the vendor,

then the application will be launched. The information about the vendor is based on digital code signing of the JAR file.

Currently, Java Web Start is geared toward simplifying the installation of Java applications via the web onto desktop machines (for instance, Sun provides implementations for the popular desktop operating systems and browsers). However, the technology could be extended to provision services over the air to constrained wireless devices.

11.7 OSGi

OSGi is the Open Systems Gateway initiative. This is an open specification for provisioning services over wide area networks to local networks and devices. The specification covers the gateway between the wide area network and the local network: initially the focus is on gateways into home networks to access set-top boxes, home entertainment systems, metering, alarm systems, etc. However, OSGi is also applicable to mobile devices and WIDs, and indeed the OSGi members include a number of major mobile phone businesses. The OSGi specification can be found at ***http://www.osgi.org/***.

11.7.1 The Services Gateway

The services gateway that sits between client devices and a wide area network is central to the OSGi and serves as a focal point for service provisioning. OSGi services are Java applications, and the services gateway is a Java-enabled device. The specification and APIs cover downloading of software, application life-cycle management, gateway security, attached device access, resource management, and remote administration of the gateway.

The OSGi 1.0 specification requires a Java 2 platform and the JDK1.2 fine-grained security model in particular. However, the intention is also to support Personal Java and smaller J2ME environments, which would mean that the services gateway could be installed on the WID itself.

11.7.2 The Service Framework

The service framework enables developers to write service code to execute on the gateway. It provides a structured development platform for component-based software architectures, taking advantage of Java's ability to download code from the network.

The service framework:

- supports dynamic load or update of services without stopping the environment
- is usable in limited memory devices
- offers a concise and consistent component programming model for service developers

- manages dependencies between services
- is scalable.

Applications are designed as a set of services, each implementing a segment of the overall functionality. These services and other extension services are then combined into a "bundle" and downloaded to the gateway.

A bundle is a JAR file that:

contains the resources implementing zero or more services. These resources may be class files for the Java programming language, as well as any other data (such as HTML help files or icons)

states static dependencies on other resources, such as Java packages. If any dependencies are stated, the framework takes the appropriate actions to make the required resource available

optionally contains classes that help the framework install, configure, activate, and update a service

declares which class should be used to start or stop a service.

By packing all these items together in a single JAR file, the framework can uniformly download and control a bundle. The JAR manifest file format is extended to contain the additional information about the bundle, and the JAR file is signed to ensure integrity and authenticity.

The framework maintains a set of mappings from services to their implementations and has a query mechanism that enables an installed bundle to request and use any available service. The framework manages dependencies between services and bundles such that when a bundle is stopped, all the services that it registered will be unregistered automatically.

11.8 MExE

MExE defines an environment for secure application download onto mobile devices. The specification covers provisioning by WAP, by Java applications and Applets, and by MIDlets.

We'll look at MExE in more detail in Chapter 14.

11.9 iBus//Mobile Messaging Service

iBus//Mobile, developed by Softwired (***http://www.softwired-inc.com/***), implements an end-to-end message-oriented middleware, compatible with Sun's Java Message Service (JMS) standard. iBus//Mobile allows Java applications running on WIDs to communicate with Java applications running on a back-end server. The iBus//Mobile lightweight JMS client is deployed on the mobile device, and

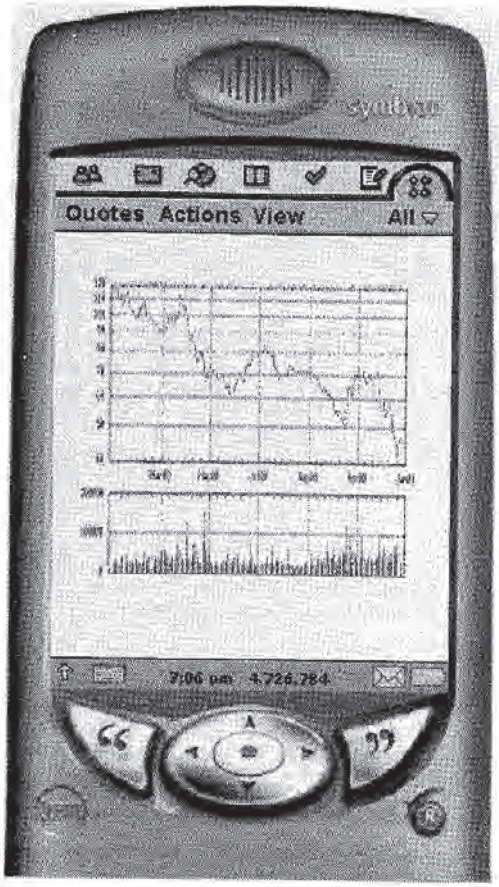


Figure 11.4 How iBus//Mobile could be used to deliver stock market information.

a J2EE application server is deployed at the back-end. iBus//Mobile ensures that JMS messages are transmitted between WIDs and the back-end in a scalable, reliable, and secure manner. Application areas are highly interactive mobile applications such as financial data services, push-notifications, mCommerce, wireless games, and so forth.

iBus//Mobile supports disconnected operation of mobile devices and operates with a variety of bearers such as SMS, GPRS, or Bluetooth, with each bearer implemented as a plug-in.

12

Case Studies

12.1 The Simple Conference Service

Jüergen Bocklage
Digia Oy (http://www.digia.com)

The first case study in this chapter describes how a Simple Conference service can be deployed using Java and Internet services. The service allows written "chat" between users as well as the ability to "share" a whiteboard by sending simple diagrams to service users. The source code of a simple implementation framework is provided to illustrate the architecture.

Sample code for both case studies can be downloaded at the Symbian website, ***http://www.symbian.com/books/***

12.1.1 Introducing Simple Conference

The *Simple Conference* service is a collaborative tool that enables textual and graphical information to be exchanged between WIDs. It is a client–server application: the user connects to a server, which resides on a known host and port. Client devices are typically WIDs, but in this case the pervasive nature of the service means they can also be desktop computers.

The current implementation supports a chat service and a whiteboard service where each user is identified by a nickname. Additional services can easily be added owing to the extensibility of the application's messaging architecture.

Connecting to the *Simple Conference* Server

The client always initiates the connection to the *Simple Conference* server, using a simple dialog. The server is identified by the host address and port number. The default port number for the server is 8080 (the host in the screen shot is fictitious).

Reproduced by permission of Telenor Communication AS, Telenor R&D
Reproduced by permission of Digia Oy



Figure 12.1 Connecting to the *Simple Conference* server from a Quartz client.

Each user must have a unique nickname and in this case, the nickname "Dr Quin" is used to identify the individual client.

The client initiates a connection with the host and port information and sends the nickname to the server. The server stores the contact information linked to the client connection for later usage. Disconnecting from the server is done manually, either when the user closes the connection or when they shut down the application.

The client requires only a single connection. Channel routing is used to run multiple services over the same connection. A channel is responsible for routing the message from client to server.

When the client has connected to the server it sends a message to the server listing the services it requires. The server then adds the client's details to each requested service supported by the server. The server supports multiple users by creating a new thread for each client. Each thread listens for messages from its client.

The Chat Service

The chat service allows a user to send text messages to other users logged onto the server (there are no chat groups in this initial version).

Text entered by a user is transmitted to the server and then published to all clients listening to the chat service. Clients display all the messages they receive, prefixed with the message sender's nickname.

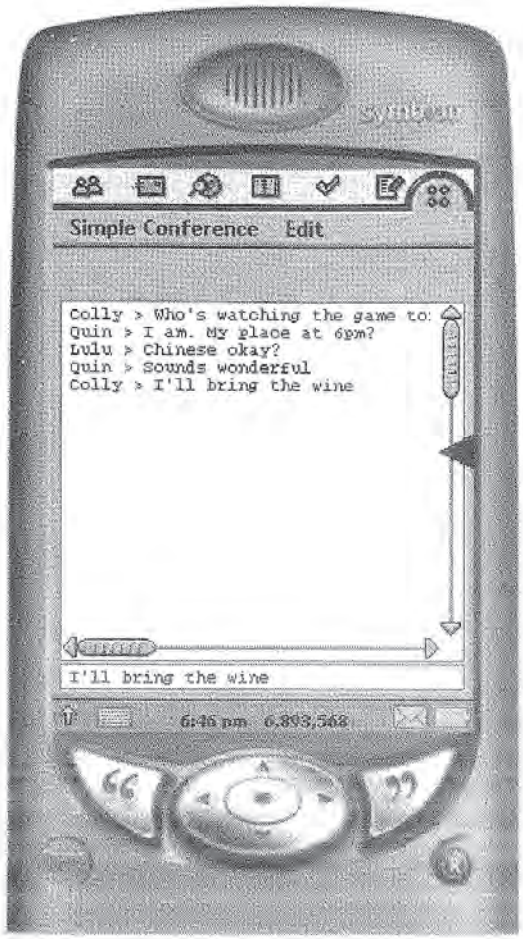


Figure 12.2 Chatting with *Simple Conference*.

The Whiteboard Service

The whiteboard is a shared graphical context. It allows multiple users to communicate asynchronously in a graphical way.

In its present incarnation, the whiteboard supports a number of simple geometrical shapes: lines, rectangles, circles, and even text. A further whiteboard attribute is that, once created, a shape can be moved around on the board. Information about adding or modifying a shape is sent to the server and from there replicated to all listening clients. A client is updated with the latest contents of the whiteboard when it first connects.

The Crystal User Interface

Crystal devices are keyboard driven, which means that the whiteboard also has to be controlled using the keyboard, rather than a pointer. Looking at the Crystal screen shot in Figure 12.4, you can see a small (red) square in the middle of the whiteboard. This is the drawing cursor. In addition, there are nine small (also red) crosses on a 3 × 3 grid.



Figure 12.3 Sharing a whiteboard as seen on Dr Quin's Quartz WID.



Figure 12.4 Sharing a whiteboard as seen on Toria's Crystal WID.

Having selected the desired drawing primitive using the menu, keys are used for drawing:

- press the space bar once to start drawing, once again to stop drawing
- the arrow key moves the cursor, ten pixels at a time. If the Ctrl key is held down the cursor will move just a single pixel
- the nine letter keys at the left-hand end of the keyboard

Q	W	E
A	S	D
Z	X	C

correspond to the nine grid points on the whiteboard. Pressing one of these keys snaps the cursor to the corresponding point.

The Quartz User Interface

Whereas the Crystal reference design is keyboard driven, Quartz uses pen input via a touch screen. This is fine for the whiteboard but means that text entry is more difficult.

The Quartz version of *Simple Conference* behaves like a typical Quartz application: tapping in the send text field enables handwriting recognition. This is indicated by the black triangle halfway up the right-hand side of the screen. (The black triangle can be seen in the first two Quartz screen shots in this chapter.)

12.1.2 Messaging

Messaging is a common way to send information in a client–server system. *Simple Conference* uses a *publish-subscribe* messaging model. This allows clients to subscribe to topics and to receive all messages sent to that topic. Other common messaging models are *point-to-point* and *request-reply*. Note that the system developed is not an implementation of Sun Microsystems' Java Messaging Services API.

Publish-Subscribe Messaging

Publish-subscribe messaging is used when multiple clients need to receive the same messages. This is the case in the chat and whiteboard services provided by *Simple Conference*. The idea of a "topic" is central to the publish-subscribe model. Multiple publishers may send messages to a topic, and all subscribers to that topic will receive all the messages sent to the topic. This model is extremely

useful when clients want to notify each other, e.g. when a text message was submitted by a chat client, or the whiteboard was updated.

Simple Conference clients are both publishers and subscribers: the *chat client* submits a text message to the server and, as a subscriber to the "chat" topic, it also receives the published messages. In other systems clients may only be recipients, for instance when subscribing to a price list or timetable.

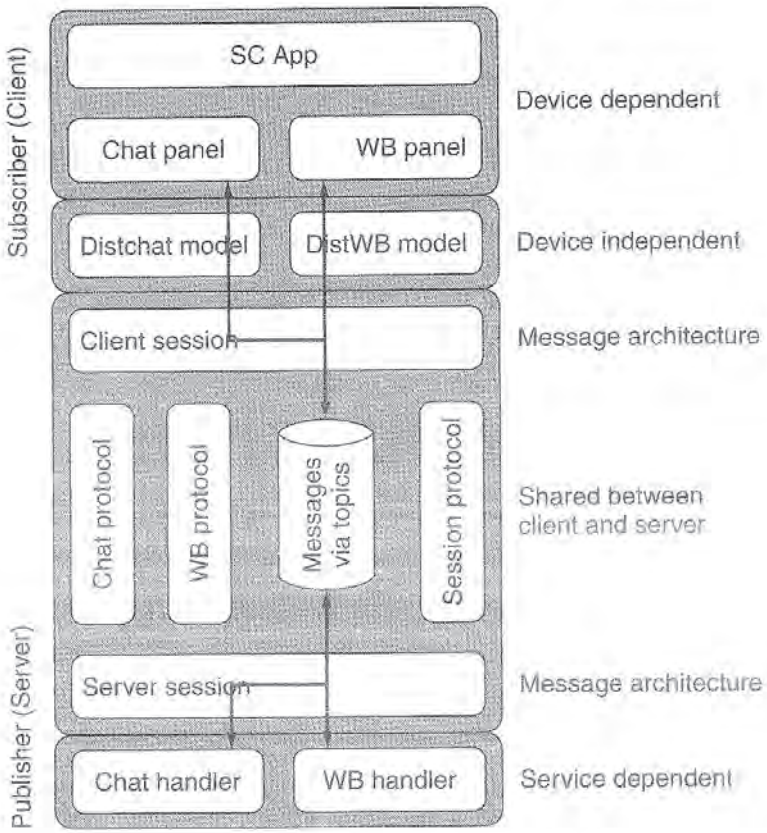


Figure 12.5 The messaging architecture.

The Message

A *message* contains a body and header information. The data to send is stored as a serializable object as the body of the message. The header contains the topic to which the message belongs and an identifier for the type of message.

For example, a chat message sent to the server would look like this:

```
Topic = "chat" (identifies the topic)
Id = SEND_TEXT (defined in the ChatProtocol class)
Body = "Bonjour mes amis" (the text to send)
```

The Message class is a simple data container class from the com.digia.j4mid.sc.msgarch.shared package:

```
public class Message implements Serializable
{
      private Object iBody;
      private String iTopic;
      private int iId;

      public Object getBody()
      {
            return iBody;
      }

      public String getTopic()
      {
            return iTopic;
      }

      public Message(String aTopic, int aId, Object aBody)
      {
            iId=aId;
            iTopic=aTopic;
            iBody=aBody;
      }

      public int getId()
      {
            return iId;
      }

      public String toString()
      {
            return "Message["+iId+","+iTopic+"]="+iBody;
      }
}
```

The following fragment from com.digia.j4mid.sc.msgarch.server.ServerSession shows how the `topic` String is used to check for the existence of the relevant service. If the service for that topic exists, `listener.handleMessage()` routes the incoming message to it:

```
private void sendToTopic(String topic, Message aMessage)
{
      if(iTopicListeners.containsKey(topic))
      {
            IMsgListener listener=(IMsgListener)iTopicListeners
                .get(topic);
            listener.handleMessage(this, aMessage);
      }
      :
```

The message Id (`id`) is used by the listener to determine the object type of the Message body and hence how the message should be handled. The following

fragment from com.digia.j4mid.sc.server.ChatHandler publishes a text message to all subscribers to the chat service:

```
switch(id)
{
        case SEND_TEXT:
                String text=(String)aMessage.getBody();
                text=aSender.getIdentification()+">"+text;
                msg=new Message( TOPIC, SEND_TEXT, text );
                ServerSession.publish(msg);
                ;
```

Publisher–Server Side

To publish a message to its subscribed clients the publisher needs to know which client is interested in which topic. This is achieved when the client first connects to the server.

A Hashtable is used to send parameters to the server the first time the client connects to the server. It is sent as a message whose Id is `INIT` using a special topic "session". This topic is only used for controlling the data flow between client and server.

The server looks up the `topics` keyword in the Hashtable, which returns a Vector of topics in which the client is interested. The client is then added to all topics to which they want to subscribe, as long as the server supports those topics. The following fragment is from com.digia.j4mid.sc.server.ServerSession:

```
;
Hashtable hash=(Hashtable)aMessage.getBody();
Vector v=(Vector)hash.get("topics");
Enumeration enum=(Enumeration)v.elements();
while( enum.hasMoreElements() )
{
        topic=(String)enum.nextElement();
        ServerSession.addSessionToTopic(topic, this);
        sendToTopic(topic, aMessage);
}
;
```

Currently the server ignores requests for unsupported topics: a future improvement in the service would be to notify the client that a particular topic is not supported.

Subscriber–Client Side

A message arriving at the client is delivered to the attached topic listener through the same mechanism as used by the server-side publisher.

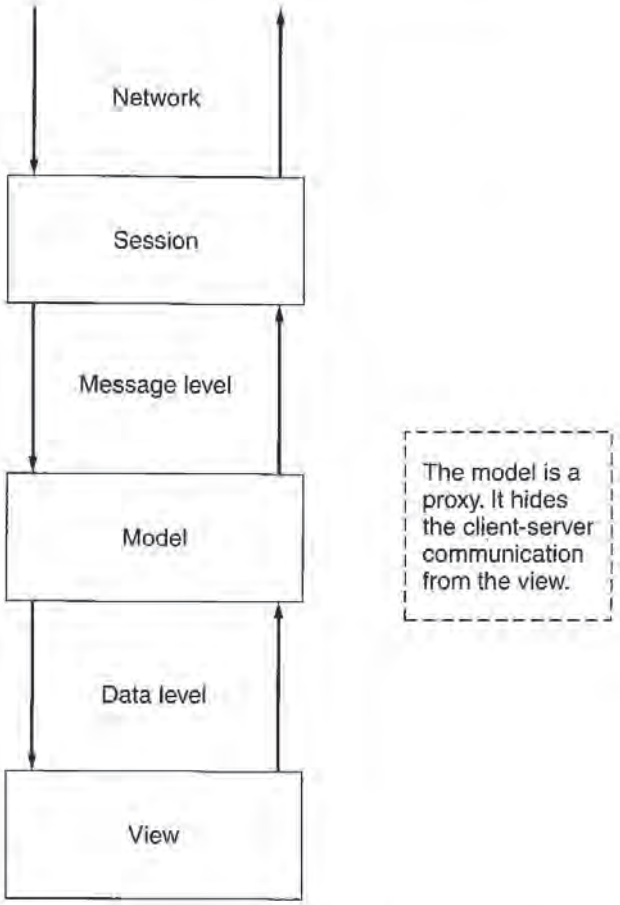


Figure 12.6 The Model-View-Controller pattern.

The listener is a Model for a service. It receives a message and notifies the attached observer. An observer is a View in the Model-View-Controller pattern, which updates the data representation.

The code, from com.digia.j4mid.sc.client.chat.DistChatModel, handles chat messages received by the client:

```
public void handleMessage(IMsgSender aSender, Message aMessage)
{
        int id=aMessage.getId();
        switch(id)
        {
              case SEND_TEXT:
                    iText=(String)aMessage.getBody();
                    iObserver.updateChat(ChatModel.DEFAULT
                       _REASON);
              break;
              ;
        }
}
```

The chat client displays the received message as a new line of text.

The model is also responsible for reporting changes in the data to the server so that other clients can be updated. To achieve this, the model wraps the data in a Message, sets the topic and Id, and sends it via the session to the server (from com.digia.j4mid.sc.client.chat.DistChatModel):

```
public void setChatText(String a Text)
{
        Message msg=new Message(TOPIC, SEND_TEXT, a text);
        iSender.send( msg );
}
```

Using distributed functionality to extend a model makes it straightforward to integrate messaging into existing applications. This is the benefit of separating data from its presentation.

Sessions

Sessions exist on the client and server as ClientSession and ServerSession. They are responsible for initiating a connection, routing received messages and sending messages to a client. On the server side there is one session per client. A session is a threaded object, which allows the session to send and receive messages asynchronously. See com.digia.j4mid.sc.msgarch.server.ServerSession and com.digia.j4mid.sc.msgarch.client.ClientSession for more information.

Following a Message's Route

To understand how a message makes its way through the messaging architecture it's beneficial to follow a message's path:

1. Text is entered in the chat panel and the send function is triggered.
2. The text is written to the model.
3. The model wraps the text in a Message with a topic (chat) and an Id (SEND_TEXT).
4. The Message object is sent to the server via the ClientSession.
5. At the server side a ServerSession responsible for this client receives the message and looks up the topic.
6. Using the topic field the ServerSession can route the Message to the attached listener, in this case the ChatHandler.
7. The ChatHandler uses the Id to decide how to handle the Message. In this case the handler prefixes the text with the client's nickname.
8. The modified text message is published to all Chat service subscribers, via the ServerSessions.
9. The message arrives at a client, which uses the topic to route it to the correct topic listener, which is a data model.

10. The data model notifies the view, which will update the text display by adding the new line of text.

12.1.3 Developing Services for Multiple Devices

Developing a client–server application for different devices requires that both packaging and bundling are thought about carefully. The *Simple Conference* client runs on both Quartz and Crystal reference devices, as well as translating to the desktop PC. These three clients have different UI requirements but nonetheless have a lot of common code.

Bundling ensures that only those packages required by a particular client are contained in the JAR (Java Archive) file delivered to the client.

Package Names

To make the package names unique, Sun recommends the use of Internet domain names, backwards, in this case com.digia. Because it is an application for Java on mobile information devices, we then use j4mid and follow this with the application name. This leads to com.digia.j4mid.sc as a common prefix for all *Simple Conference* packages.

Simple Conference is a client–server application, which introduces the next level. Client and server share some common classes, so we get the following branches:

```
com.digia.j4mid.sc.client
com.digia.j4mid.sc.shared
com.digia.j4mid.sc.server
```

Common client classes are packaged under the client level. For the device-specific UI parts a package for each device is created:

```
com.digia.j4mid.sc.client.ui.desktop
com.digia.j4mid.sc.client.ui.crystal
com.digia.j4mid.sc.client.ui.quartz
```

Each service provided on the client occupies a package, in this case the chat and whiteboard services:

```
com.digia.j4mid.sc.client.ui.chat
com.digia.j4mid.sc.client.ui.wb
```

The messaging architecture is an independent architecture, which is placed in its own branch. Because it is also a client–server architecture it gets the typical packaging signature:

com.digia.j4mid.sc.msgarch.client
com.digia.j4mid.sc.msgarch.shared
com.digia.j4mid.sc.msgarch.server

Bundling

A bundle is a combination of packages, with each bundle being a separate project with its own configuration.

Each device requires its own bundle, either client-desktop, client-crystal or client-quartz, while the server requires a fourth bundle. Common client services add one bundle to the list, the client-bundle; and a bundle for the shared classes is also needed, called the shared-bundle. Finally messaging requires three bundles, the client-msgarch, server-msgarch and shared-msgarch.

The targets are formed by a combination of bundles:

Target	Client-desktop	Client-crystal	Client-quartz	Client-msgarch	Shared-msgarch	Server-msgarch	Shared	Server
Desktop	X			X	X		X	
Crystal		X		X	X		X	
Quartz			X	X	X		X	
Server					X	X	X	X

The bundles are combined into a single jar or zip file for deployment on the target client. Simple script files automated the bundling process and reduced the likelihood of errors.

12.1.4 References and Resources

Hughes Shoffner, Hamner, *Java Network Programming, A Complete Guide to Networking, Streams and Distributed Computing*, Manning Publications, 2nd edn, July 1999, ISBN 188477749X

Hughes, *Draw the world, Create networked whiteboards with Java 1.1*, published on JavaWorld website November 1997, http://www.javaworld.com/javaworld/jw-11-1997/jw-11-step.html by ITWorld.com

Gopalan Suresh Raj, *Java Messaging Service*, first published on his website September 1999, http://www.execpc.com/~gopalan/jms/jms.html

Sun Microsystems, *Java Message Service API*, first published on java.sun.com, http://java.sun.com/products/jms/

Todd Sundsted, *Messaging Makes its Move*, first published on Java World website February 1999, http://www.javaworld.com/jw-02-1999/jw-02-howto.html

Softwired, *iBus/Mobile JMS 1.0 (Java™ Message Service)*, software can be downloaded from http://www.softwired-inc.com/

Burbeck, *How to Use Model-View-Controller in Smalltalk*, Applications Programming in SmallTalk-80™, 2nd edn 1992 http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

12.2 The Handheld Travel Assistant

Sigrid Steinholt Bygdås, Rune Henning Johansen, Øystein Myhre and Tore Urnes
Telenor R&D (http:/www.telenor.com/rd)

This section of Chapter 12 illustrates Telenor's experience building a handheld travel assistant using Java and existing Internet services. It begins by describing a travel scenario before presenting the *Handheld Travel Assistant* service. The travel assistant combines and adds value to existing third-party services, and provides an architecture for simplifying the adaptation of existing services to wireless devices is presented. To illustrate the architecture, the source code of a simple implementation framework is provided.

12.2.1 Travel Scenario

Gudrun and Ragnar are going on a holiday to Oslo. They arrive at the Central Station on the evening train from Copenhagen. Exhausted from the trip, they decide to go to their hotel right away. They know that the hotel is located only a few steps from the north-west station exit, and they decide to give the taxi queue a miss.

While travelling, Gudrun had made herself familiar with a map of Oslo on her handheld travel assistant. She followed their own movements, marked on the map as small red circles, as the train steadily progressed towards the station. Gudrun had also entered the address of their hotel and the travel assistant now maintained a green circle on the map to denote the hotel location.

The next day, Gudrun and Ragnar head for the Vigeland Park, one of the top-rated attractions listed in their travel guide. Ragnar asks the travel assistant for public transportation routes to the park. On the way back to the hotel, they decide to be a bit adventurous and set out to walk back to town. About half-way there, they decide to get a bite to eat and ask the travel assistant to point out the locations of fish restaurants within five minutes walking distance from where they are currently standing.

Back at the hotel, Gudrun wants to go and check out the old Akershus fortress down by the fjord while Ragnar remembers that he has promised to pick up a few souvenirs for friends at home. They decide to split up and meet back at the hotel in a few hours. An hour later, Gudrun is so impressed by the beautiful view of the sunset from the top of the old fortress wall that she calls Ragnar on his cell phone to see if he wants to come down. Ragnar agrees and turns on his travel assistant. He tells the assistant to pinpoint Gudrun's current location. He can now easily find his way and share the view with Gudrun.

12.2.2 The Travel Assistant Application

We have built a handheld travel assistant like the one described in the scenario. It is a pure Java implementation that runs on Windows, EPOC Release 5 (Psion 5mx and Psion netBook) and Symbian OS v6.0 (using Quartz and Crystal emulators). Most of the functionality was realized with little effort by simply combining and adding value to a set of existing services. These services are a map service, different spatial search services, a transport service, and a location service.

The map service uses maps from MapOnWeb[1], a company offering digital maps on the web. Figure 1 shows a map from MapOnWeb covering a part of downtown Oslo. Below the map there are function buttons that you can use to zoom in and out, pan, get information, select areas of interest, and get route information.

MapOnWeb supports a wide range of parameters for its maps: region, scale, level of detail, color scheme, etc. The map server has good performance. Maps



Figure 12.7 Downtown Oslo.

[1] http://www.maponweb.com

are generated as GIF (Graphic Interchange Format) or JPEG (Joint Photographic Experts Group) images on the fly. The images were originally intended for wired access. However, the limited screen size on handheld devices also limits the size of images, which makes the images suitable for transport over the wireless GSM network. MapOnWeb's map base currently covers Scandinavia, and will cover Europe during 2001.

The map in Figure 12.8 has a small red oval mark on it (marked by the arrow). We call such ovals *map marks*, and use them to denote the location of objects of interest to the user. Map marks can represent stationary objects like restaurants, bus stops, and hotels, or moving objects like persons, trains, etc. You can get information about a particular map mark by pointing at it on screen. In Figure 12.8 we can see that the map mark represents Gudrun.

The spatial search services use different (Norwegian) databases. The results of a database search are shown as map marks on the map. Figure 12.9 shows the result of an address search for a street in Oslo.



Figure 12.8 On this map you can see Gudrun, represented by an oval map mark, walking on the Western Wall of the Akershus fortress. Gudrun's name appears when you point at the map mark.

The public transport service makes use of Trafikanten[2]. Trafikanten is a public service delivered via phone, web, and WAP (Wireless Access Protocol), giving information about public transportation in and around Oslo. Our service adds value to the original service. It locates stations near at the user's' current location or in a selected area, and marks them on the map. If several routes are possible, our service calculates the best route. The user can define the 'best route' to be the fastest route, the route with the shortest total walking distance, or the route with fewest transportation changes.

We also supply our own location service. The location service keeps track of its clients' locations. However, a client is the owner of its own location. If interested, a client can become a member of a location group. A location

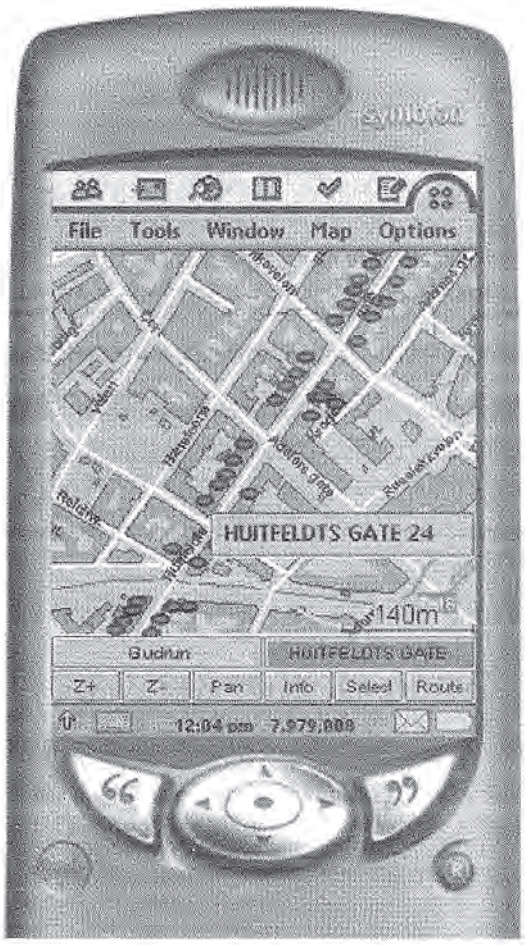


Figure 12.9 The result of an address search. The search criterion was part of the street name. The ovals are map marks representing the street numbers. To see a house number you point at the map mark at the house of interest. The tabs just above the function buttons shows that this travel assistant holds two maps for its user at the moment. One map covers the city center of Oslo, while the other covers the street the user has searched for

[2] http://www.trafikanten.no

group is quite similar to an Internet chat group, except that the information exchanged is the group members' locations. A group can be a family, friends, or trucks in a transport company. Locations are continuously updated on the map. This way it is possible for the members of a location group to follow the movements of the other members of the group, see Figure 12.10. For a client to be able to know its own location, a location sensor must be attached to it. A global positioning system (GPS) sensor can be used for this purpose.



Figure 12.10 The map marks here represent the members of a location group. Location group members exchange their locations. We can see that most of this group's members are in Oslo. By pointing at the map marks, we can see their names. At the moment we can, for instance, see that Hege is in Valdres

Our travel assistant application demonstrates how useful functionality can be realized in a simple and timely fashion by combining and adding value to existing, third-party services. To get people interested in using a service, the service content must be of high quality. A very important task when building an application like the Handheld Travel Assistant is to identify and reach

agreements or make partnerships with providers of high quality content. This may also be a time-consuming task.

12.2.3 Application Architecture

Much of the travel assistant's functionality is realized by combining and adding value to existing third-party services. Often such third-party services are not designed for the severe resource constraints of wireless devices, meaning they fail to take into account the limitations on battery life, screen real estate, CPU power, and network bandwidth. A common solution to this problem is to introduce an adaptation layer between the wireless device and the service. The adaptation layer can be thought of as sitting close to the edge of the wired network. It performs various transformations and optimizations, allowing wireless devices to interact with services in a timely and economical manner. To further increase the flexibility and ease by which wireless devices interact with all kinds of services, we introduced service proxies on the wireless device. A proxy is a local object that allows us to hide the network protocols used to interact with services.

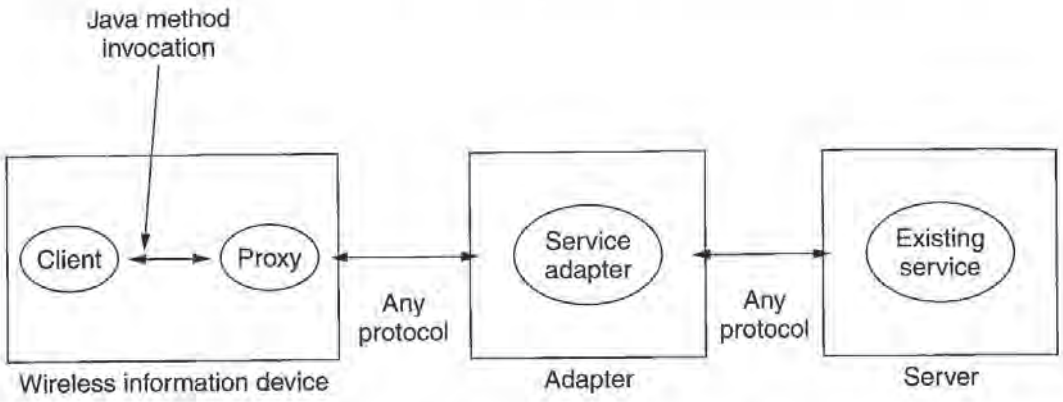


Figure 12.11 Generic service adaptation architecture

Figure 12.11 depicts the generic architecture for adapting services to wireless devices. The client application running on the wireless device interacts with a third-party service by invoking a method on the corresponding service proxy. The proxy forwards the request, using an appropriate protocol, to the adaptation component, which in turn interacts with the service, potentially employing yet another protocol. Service responses may be subject to computations, e.g. pruning, compression, transformation, aggregation, etc., in the adaptation component before being returned to the proxy. Finally, the proxy constructs a response to the service request.

By applying the generic architecture for service adaptation to the Handheld Travel Assistant, we arrive at the architecture in Figure 12.12. Some of

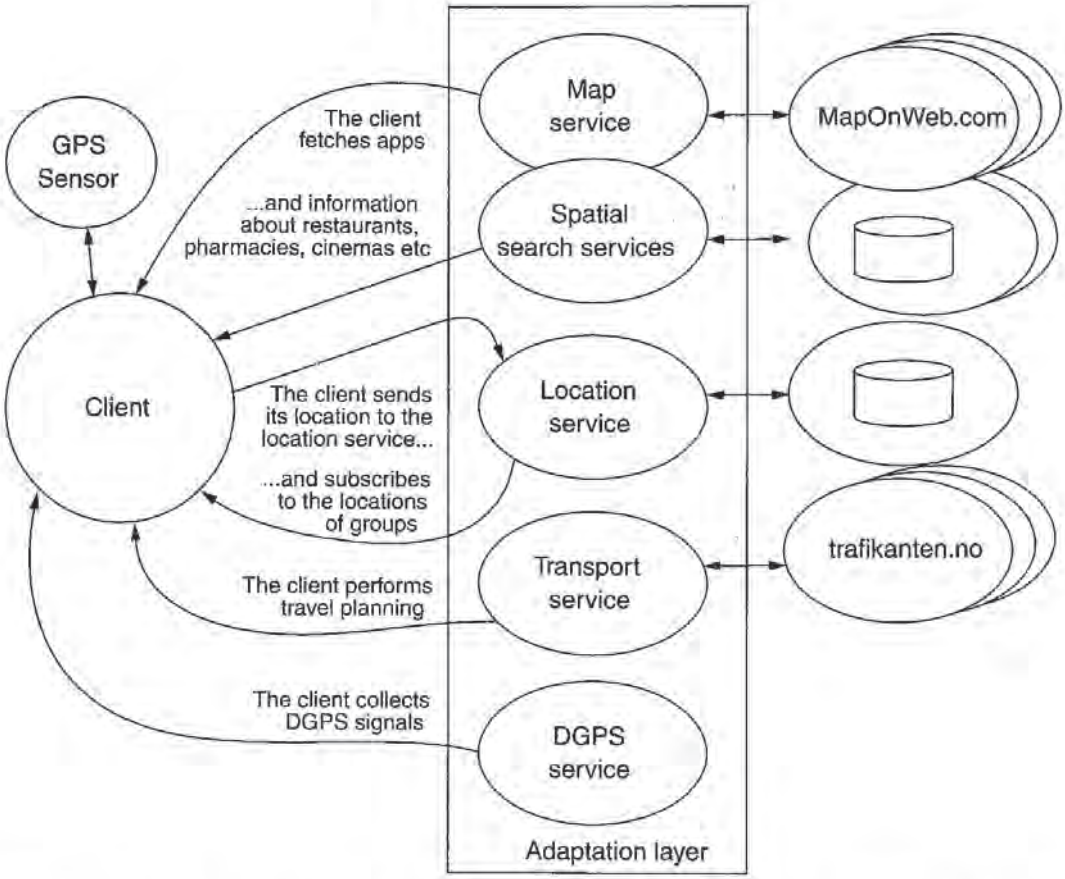


Figure 12.12 The architecture of the Handheld Travel Assistant at a glance. The application combines existing and new services into a handheld device. This way it adds value to the services. Some of the existing services are slightly adapted

the existing services need to be slightly adapted, some not. The MapOnWeb service, for instance, has a programmable web interface and needs no adaptation, while the Trafikanten service does. Our transport service needs to act as a human user towards the Trafikanten web service and in addition perform some calculations, e.g. finding the best route.

The Handheld Travel Assistant is a Java application specifically designed for a Symbian device. This is done to reduce start-up time and to enable usage of local resources such as the file system, serial I/O and databases. A Java Applet embedded in a web browser is prohibited from accessing such resources, and would therefore not be suitable.

We chose Java as the implementation language, mainly because it provides us with a platform for rapid prototyping. In addition, the Java language enables mobile code, and provides the required security mechanisms.

Servlet technology is a de facto standard for service deployment. A Servlet runs in a Servlet container, and such containers can be plugged into web servers,

in our case the Apache Tomcat[3] web server. A proxy is an object located on the client, representing a remote service. The proxy hides implementation details of the communication between client and server from the client. This means that you can change a proxy without affecting the client code, as long as the new proxy implements the same interface. You can even change proxies at runtime. By using proxies a service provider can easily adapt a service by making a new proxy that replaces the old one, e.g. adding caching or session management. Figure 12.13 illustrates how services are implemented in our application.

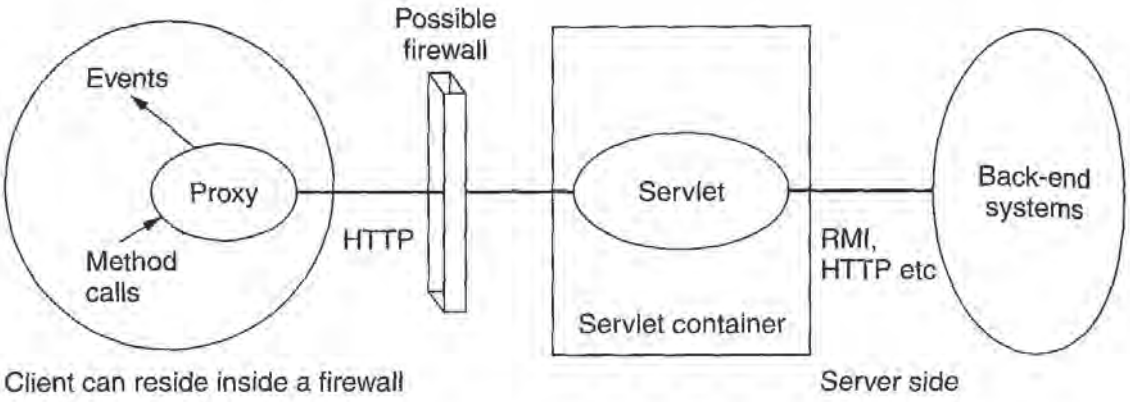


Figure 12.13 Services are implemented as servlet–proxy pairs. This gives a simple and standard way to implement and deploy services. We get plug-in possibilities on both client and server side

We use proxies on the client to interface to location sensors, too. This way we can change sensors if we like. We have made a proxy for a GPS (or DGPS) sensor. A GPS-sensor indicates your outdoor location with a precision of 10 meters. To get more accurate location information (3–5 meters) it is possible to use a differentiated GPS (DGPS) sensor. A DGPS-sensor has to be fed with special signals in order to correct the GPS location co-ordinates. To receive these signals requires the use of an FM-antenna. However, it is not always convenient to carry an antenna around. We have therefore created an Internet DGPS-service where a client can fetch the signals required. If the sensor we plug into the handheld device is a DGPS sensor, the proxy uses our DGPS service to get the signals it has to feed to the sensor. The DGPS service is implemented as a servlet–proxy pair like the other services.

The Servlet–proxy pairs of the travel assistant application use HTTP (HyperText Transport Protocol), not Java RMI (Remote Method Invocation), to exchange serialized objects. The reason for this is that Servlets cannot be accessed remotely. If we had chosen Enterprise Java Beans (EJB) as our

[3] http://jakarta.apache.org/tomcat/

component model, RMI would have been applicable. We chose Servlet technology because it gives us a standard way to deploy services today. In a year or two, EJB may be as common as Servlets. At the moment, however, using EJB means large investments on the server side, while many web servers are able to run Servlets.

In our location service implementation, the proxy polls the Servlet for location changes. The proxy can instead be implemented so it receives notifications of location changes from the server. If the Travel Assistant sits behind a firewall, on a wireless LAN in the office perhaps, notifications will not get through, so the client would have to poll. However, polling is resource consuming, and should be avoided if possible. The proxy should be made smart enough to decide if it is inside a firewall or not. If inside, it polls the server for changes, if not, it just awaits notifications. Depending on the need for information updates, it may not be necessary to poll very often. It is possible to make the proxy smart enough to poll at different time intervals, depending on information quality preferences or expected state changes. In the context of the JavaPhone APIs, an alternative solution is to enable the services to send notifications via SMS (Short Message Service).

12.2.4 Service Implementation

The Travel Assistant shows that you can add value to existing services by combining them. We have stated that it is easy to make new services this way, and even easier when we combine proxies with a component model, in our case Servlets. We will now show a straightforward implementation of a service as a Servlet–proxy pair. Our example service is called DemoService. It is a simplified participants' list-service, where everyone can become a participant. The service allows you to add a participant, get the number of participants, and the participant list. The proxy polls the servlet for changes in the number of participants. This DemoService can act as a framework for implementing a real service.

Interfaces

The DemoService implements two interfaces, a service interface and a listener interface.

The service interface defines the attribute methods of the service, and is implemented by the smart service proxy:

```
public interface DemoService
{
        public int numberOfParticipants();
        public int addParticipant(String name);
```

```
        public String[] getParticipants();
}
```

The Listener interface specifies the notifications that the service proxy can generate. When a service event occurs the relevant method in the registered Listener object is invoked. In our example, the only notification defined is `numberOfParticipantsChanged`.

The class that is interested in processing service events implements this interface, and registers itself by calling `addDemoServiceListener()`:

```
public interface DemoServiceListener
{
     public int numberofparticipantschanged();
}
```

Service Proxy

A DemoServiceProxy constitutes the client side of the service. It implements the DemoService interface, thus gaining access to the service. This implementation caches the number of participants and uses a polling thread to monitor the server-side copy of the `numberOfParticipants` attribute. Note that the polling thread is not created unless a notification listener is registered on the proxy. The attribute methods of the service are implemented using remote calls over HTTP, described later.

Using remote calls over HTTP, desecribed later. Note that the polling thread is not created unless a notification Listener is registered on the proxy.

`nParticipants` is the cached copy of the number of participants, `pollingRate` is the polling interval (once per minute).

The `webServerHost`, `webServerPort`, and `servletUri` should reference the host IP address, port number, and servlet identification:

```
import java.util.Enumeration;

public class DemoServiceProxy
extends AbstractServiceProxy implements DemoService

{
     protected int nParticipants=0;
     protected long pollingRate=60000;
     private Runnable poller;

     public DemoServiceProxy()
     {
          webServerHost= ... ;
          webServerPort= ... ;
          servletUri= ... ;
     }
```

```
public int numberOfParticipants()
{
     Object reply=remoteCallOverHttp
         ("numberOfParticipants",null);
     if(reply instanceof Integer)
         return(((Integer)reply).intValue());
     return(FAILED)1;
}

public int addParticipant(String name)
{
     Object[] args=new Object[]{name};
     Object reply=remoteCallOverHttp("addParticipant",args);
     if(reply instanceof Integer)
         return(((Integer)reply).intValue());
     else return(FAILED);
}

public String[] getParticipants()
{
     Object reply=remoteCallOverHttp("getParticipants",null);
     if(reply instanceof String[]) return((String[])reply);
     else return(null);
}

public void addDemoServiceListener(DemoServiceListener
    listener)
{
     addServiceListener(listener);
     if(poller==null)
     {
          poller=new Runnable()
          {
               public void run()
               {
                    while(true)
                    {
                         try
                         {
                              Thread.sleep(pollingRate);
                         }
                         catch(InterruptedException e){}
                         polling();
                    }
               }
          };
          new Thread(poller).start();
     }
}

protected void polling()
{
```

```
                int n=numberOfParticipants();
                if(n>0)
                {
                    if(n!=nParticipants)
                    {
                        nParticipants=n;
                        for(Enumeration e=listeners.elements();
                                e.hasMoreElements();)
                        ((DemoServiceListener)(e.nextElement()))
                                .numberOfParticipantsChange(n);
                    }
                }
            }
        }
```

Abstract Service Proxy

The class DemoServiceProxy above implements the DemoService interface and extends a class called AbstractServiceProxy. The main task of this class is to supply all subordinate service proxies with a method for performing remote calls over HTTP. The implementation chosen packs the remote method name together with the argument list into an object of class RemoteCall, and 'POSTs' a serialized version of that object over a HTTP connection. Class RemoteCall is described later. The HTTP response is interpreted as a serialized version of the remote method's product, i.e. either the return value or an exception object.

`webServerHost`, `webServerPort`, and `servletUri` are intitialized by the subclass:

```
import java.util.Vector;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.URL;
import java.net.HttpURLConnection;

public abstract class AbstractServiceProxy
{
    protected String webServerHost;
    protected int webServerPort;
    protected String servletUri;
    protected Vector listeners=new Vector();

    public synchronized void addServiceListener(Object
        listener)
    {
        listeners.addElement(listener);
    }

    protected Object remoteCallOverHttp(String methodName,
    Object[] args)
```

```
        {
            RemoteCall call=new RemoteCall(methodName,args);
            try
            {
                URL url=new
URL("http",webServerHost,webServerPort,servletUri);
                HttpURLConnection conn=
(HttpURLConnection)url.openConnection();
                conn.setRequestMethod("POST");
                conn.setDoInput(true); conn.setDoOutput(true);
                conn.connect();

                ObjectOutputStream out=new
ObjectOutputStream(conn.getOutputStream());
                out.writeObject(call);
                out.flush();

                ObjectInputStream in=new
ObjectInputStream(conn.getInputStream());
                Object reply=in.readObject();
                return(reply);
        }
        catch(Exception e) { return(e); }
    }
}
```

Remote Call

The method `remoteCallOverHttp()` of the class AbstractServiceProxy uses a RemoteCall class. When performing a remote call, the remote method's name and its corresponding argument list are packed into an object of this class. This object then constitutes a complete remote method call suitable to serialize and send over an object socket.

```
public final class RemoteCall implements Serializable
{
    private String methodName;
    private Object[] args;

    public RemoteCall(String methodName,Object[] args)
    {
        this.methodName=methodName;
        this.args=args;
    }

    public String getMethodName() {return(methodName);}
    public Object[] getArguments() {return(args);}

}
```

Servlet

The server side is implemented as a servlet that connects to the underlying service on the same or a remote server. To make it simple, the service is represented by the same interface as described above, the DemoService interface. The service can be a class that implements this interface and, for example, accesses a database using JDBC (Java Database Connectivity) or an Enterprise Java Bean (EJB).

The DemoServlet responds to an HTTP POST by interpreting the servlet input as an ObjectInputStream containing a serialized method call. The method call is unpacked and the corresponding local method is called. The result of the method call is delivered to an ObjectOutputStream. If anything goes wrong, an exception object is delivered to this stream.

`demoService` references the underlying Service:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class DemoServlet extends HttpServlet
{
     private DemoService demoService;
```

`init()` connects to the real service, e.g. a database. It is called by the Servlet container to indicate to a Servlet that the Servlet is being placed into service:

```
     public void init() {demoService=...}
```

`doPost()` Services HTTP-POST requests:

```
     protected void doPost(HttpServletRequest request,
HttpServletResponse  response) throws ServletException,IOException
     {
          Object reply;
          try
          {
               ObjectInputStream in=new
ObjectInputStream(request.getInputStream());
```

Read input and call local method:

```
               RemoteCall call=(RemoteCall)in.readObject();
               reply=
  methodCall(call.getMethodName(),call.getArguments());
          }
          catch(Exception e) {reply=e;}
```

Write output:

```
            ObjectOutputStream out=new
ObjectOutputStream(response.getOutputStream());
            out.writeObject(reply);
            out.flush();
        }
```

Method call skeleton code for this service:

```
        private Object methodCall(String methodName,Object[] args)
        {
            try
            {
                if(methodName.equals("numberOfParticipants"))
                {
                    return(new
Integer(demoService.noOfParticipants()));
                }
                else if(methodName.equals("addParticipant"))
                {
                     String name=(String)args[0];
                     return(new
Integer(demoService.addParticipant(name)));
                }
                else if(methodName.equals("getParticipants"))
                {
                     return(demoService.getParticipants());
                }
                throw(new Exception("Unknown Method name: "+
methodName));
            }
            catch(Throwable e) {return(e);}
    }
}
```

12.2.5 Conclusion

The Handheld Travel Assistant is a demonstrator, not a product. To make it a real product, not only security issues, but also session management and user interface design have to be dealt with. We also need to enable clients to download, install, and upgrade services. We believe that standard mechanisms for handling this will be available in the near future. The way we use Java means our application is prepared for mobile code and the use of discovery services such as Jini. The Handheld Travel Assistant will be usable anywhere in the world by connecting to regional map and information services based on the user's current location.

Section 4: Looking deep and looking forward

A detailed look at how the Java Native Interface (JNI) can be used to access more of the capabilities of Symbian OS, and a look ahead to how Java will develop on the platform

13. JNI page 369–431
14. Looking to the future page 433–447

13

JNI Programming for Symbian OS

Colin Turfus

Although the Java virtual machine promises and in many ways realizes the ideal of "Write Once, Run Everywhere," it does not "Run Everything"—in particular, it has very limited access to platform-specific resources. The Java Native Interface is a feature introduced by Sun Microsystems into JDK 1.1 in recognition of limitations of what could be accomplished with Java code alone in the Java Virtual Machine. The Java Native Interface superseded a number of competing solutions (by Netscape, Microsoft, and Sun themselves) to the problem of accessing native code from Java and vice versa.

13.1 Introduction

13.1.1 Java's Limitations

You may wonder, as you embark on this chapter, whether JNI is something you really need or want to get involved in. Reading this section should help you answer that question. You may well find your answer is in the negative, in which case you can skip to the next chapter.

Of course, you may know in advance that you need to work with the Java Native Interface; if, for example:

You have legacy C or C++ libraries which you plan (or are required) to access from a Java application.

You have performance issues with a computation-intensive engine written in pure Java and have an explicit requirement to compile some of the time-critical operations or loops down to native code.

But if you're uncertain as to whether you should use a JNI approach to solving your problem, it is perhaps worthwhile considering the opposite question: in what circumstances would I *not* want to use a JNI approach? For, although the Java Native Interface was introduced to provide a standard method for Java and native code to talk to each other, it is by no means always necessary to

resort to JNI programming to achieve this effect. Subject to certain limitations, it is possible for you to achieve a level of cooperation by other, less exercising approaches. The following are worth considering, even if only as partial solutions, before you embark on any JNI programming in earnest.

Calling Native Code from Java code

If the native code you want to run can be launched as an executable in a separate process, the JDK 1.1.x Runtime has a method for achieving this effect, namely the `exec(...)` method of the `java.lang.Runtime` class. For example, you might use something like the following code:

```
Runtime runtime = Runtime.getRuntime();
runtime.exec(''c:\\WebProgs\\Browser.exe -f Helpfile.htm'');
```

to display an HTML help file—associated, say, with the current Java application—in a native browser. (Notice the double backslash required because the single backslash is used by Java as an escape character.) This solution is available equally to Java programs running on Symbian devices.[1] In particular, if you wanted to launch, say, the native Contacts application on a Symbian device, the appropriate command line syntax would be:

```
z:\System\Programs\AppRun.exe z:\System\Apps\Contacts\Contacts.app
```

A limitation of this approach is that there is no means of communication between the Java Runtime and the process in which the native executable is running. If you need to make a callback, you'll have to use JNI.

Calling Java Code from Native Code

Conversely, you may want to run some Java code from inside a native executable. The Java Native Interface provides the so-called Invocation API to allow you to instantiate a virtual machine and arbitrary Java objects inside a native thread and to call arbitrary methods of arbitrary Java classes. If these methods return data to the native thread, the JNI approach will be the only one open to you. However, if you don't have such a requirement you could think about simply invoking **pjava.exe** with the name of the Java class you want to run and the arguments you want to pass to its `main()` method. A virtual

---

[1] A defect in EPOC Release 5 prevents command lines being parsed correctly, with the result that arguments cannot be passed to executables in this way. This problem has been fixed in the Personal Java Application Environment shipping with Symbian OS Version 6.0. However, the reader should be aware that this will still not work on the WINS emulator, which does not support the launch of .exe files. Applications are launched in the emulator from an `EikDll::StartAppL(...)` method exported from **Eikon.dll**. The only way to launch a native application from a Java application running in the WINS emulator is to call this function through JNI.

machine would then be instantiated as a separate process.[2] On a Symbian device, **pjava.exe** is located in the folder **\System\Programs** on the drive on which the Java Runtime was installed.

Standard Java Extension Packages

Just because the functionality you want to access is not part of the Personal Java Application Environment doesn't mean you have to go native. Numerous extensions have been made to the core capabilities of the Java Runtime with standard, documented APIs, implementations of which are certified by Sun Microsystems in much the same way as a Java Runtime would be. Such extensions will often be identified by their belonging to a subpackage of `javax.*`; all the JavaPhone packages described in Chapter 5 are of this kind. Alternatively, extensions may be implementations of interfaces specified as part of a standard Runtime environment. Because they typically serve to provide a Java API to access native functionality (provided in DLLs resident on the device and/or shipped and installed as part of the extension package), these extension packages will usually be platform specific.

You may thus be able to take advantage of the fact that someone else has already done the necessary native programming for you—and indeed been awarded a certificate confirming that they knew what they were doing! It is well worthwhile checking over Sun's website (***http://java.sun.com/products/***) to see if an extension package exists which satisfies your need; and it may be that this has already been implemented for the Symbian OS. Notable examples of third-party implementations of such Java extensions available for Symbian devices are:

- a number of databases (some pure Java) with JDBC drivers allowing them to be accessed through a standard API
- client software for the Java Messaging Service

At the time of writing, these are available mainly only for EPOC Release 5, but work is currently under way to make them available for Version 6.

Custom Java Extensions

Of course, as a Java programmer looking to extend the capability of the Java Runtime on Symbian devices without resort to JNI, you need not restrict your attention to standard Java extensions. There are plenty of proprietary extensions available as component libraries that can be purchased off-the-shelf from third parties or even downloaded from the web as freeware or shareware. Most of

---

[2] Clearly, for the reasons given above, this approach will only work on Symbian devices and not on the WINS emulator.

these are not platform-specific and do not provide access to native EPOC functionality. Various proprietary extensions specific to Symbian OS are available from the Symbian Developer Network on ***http://www.symbiandevnet.com*** and on the Version 6 Symbian SDKs for Java. These are made available in subpackages of `com.symbian.devnet.*`. They mostly make use of some JNI programming and may save you that trouble. Indeed they have formed the basis for a number of the JNI examples we will be looking at below.

If you still can't find quite what you need even among these examples, then getting to grips with JNI programming may be unavoidable for you. In which case, you should read on...

13.1.2 What can I do with JNI on Symbian OS?

To round off the picture, let's take a quick tour of some of the kinds of things you could conceivably do, or which have already been done, with JNI on Symbian OS:

We have already mentioned the introduction of JavaPhone APIs into Version 6.0. JNI programming offers the possibility of providing similar functionality. Indeed most of the EPOC Release 5 JNI example code published on ***www.symbiandevnet.com*** could be described as such.

JNI allows the possibility of Java code controlling Symbian device drivers. For example, although neither printing nor sound recording is supported from Java on Symbian OS, they are from native code, and by the same token from JNI code.

Although Java applications can conveniently handle standard data formats like text, GIF, JPEG, and even PDF files, they are intrinsically ill-suited to handling Symbian native data types, e.g. Word, DBMS, Agenda, Contacts. This is because these files are encoded in a proprietary manner and only intended to be accessed through standard APIs, on Symbian devices or, additionally, on a PC (for connectivity purposes). Although Agenda and Contacts files have now been rendered accessible through JavaPhone APIs, others have not. But JNI allows the possibility of working with files of any of the standard Symbian native data types.

Although the Java Runtime provides a method for accessing Java system properties, it naturally doesn't provide an independent mechanism for getting and setting EPOC system properties. Of course, some of these are reflected in Java system properties like locale, system language, etc. Others, like power status, may be accessed through JavaPhone APIs. However, through JNI you can get and set other EPOC system properties—like whether the user prefers an analog or digital clock display, or fine points of currency and number formatting, such as whether a currency symbol precedes or follows the amount.

Potentially more usefully, JNI also provides a window onto EPOC system events. That is, it allows you to set up a listener in a Java native thread to

wait for EPOC system events of interest and execute an event handler of your design. This event handler can furthermore make a callback to the Java Runtime and, say, notify another thread there. For example, if you are working with a `java.util.Locale` or `java.util.TimeZone` object, you may want to know when the user has changed the device's home city or some other locale setting and reinitialize the appropriate Java objects. Likewise, you might want your Java application to execute a start-up sequence when the device powers up after a period in idle mode.

Finally, in the event that some method or class in Symbian's Java Runtime does not behave according to the specification against which it was certified, there is always the possibility that JNI will provide a mechanism for you to work around that defect and provide an alternative class or method with the properties you seek.

13.1.3 What can't I do with JNI on Symbian OS?

I should avoid giving the impression that JNI is a panacea providing to the Java developer all the resources available to a developer writing native applications for Symbian OS. There remain a number of limitations of which you should be aware, mostly relating to application architecture and graphical user interface:

The main limitation of JNI programming lies in the fact that all interaction between an instance of the Java Runtime and the EPOC window server is through a single encapsulated native thread called the AWT-Server thread. This is accessible only to the private native methods called by AWT peer classes to allow AWT Components, MenuComponents, and related objects to display themselves. In other words, you can only access the EIKON environment indirectly through AWT classes, and subclasses thereof; you cannot do anything with EIKON visual components in Java windows or dialogs. It is intended that, following a planned new AWT architecture, this limitation will not apply to version 7 of the Symbian OS. But it will pose a limit to developers at least until version 6.2.

There are a few apparent exceptions here in that certain standard native dialogs such as system-modal Notify dialogs, the Internet connection dialog, and the task list can be launched from Java native methods; and can even return a value to be processed by that method. But in reality these dialogs are being launched in quite separate threads controlled by EPOC servers running in non-Java processes.

A related problem is that, on Symbian OS, shared resources are typically made available to multiple processes through EPOC servers (subclasses of the EPOC `CServer` class). These are accessed through client sessions encapsulating handles which may be thread-specific and so not readily sharable between Java threads. For more details, see section 13.6 below.

Under Quartz 6.0, a native application can register views allowing other apps to present Direct Navigation Links (DNLs). When these are selected by the user, they cause that application to be opened (if necessary) and one or other of its views to be displayed, possibly using some data passed from the DNL to initialize the view. While use of JNI allows the possibility of a Java application displaying DNLs linking to another native applications, it does not allow it to register views and so itself be activated through the DNL mechanism.

13.2 JNI and Symbian OS

13.2.1 Prerequisites

Software

Before you get started with JNI development you will need to kit your PC out with the requisite software:

- Symbian SDK for Java
- JDK 1.1.8 (or equivalent) Java compiler
- Symbian C++ SDK
- Microsoft Visual C++™ v6.0

Detailed requirements vary depending on your choice of target platform. Options currently include:

EPOC Release 5 — Psion Series 5mx, Psion Series 7, Psion netBook, Ericsson MC218

In the case of EPOC Release 5, you will need to install the C++ SDK and the SDK for Java onto the same (`subst`ed) drive on your PC. It is necessary to install the C++ SDK first, then the SDK for Java. When you install the second of these, the install shield will detect that an EPOC Release 5 SDK has been installed on that drive and offer to delete it. You must answer "No" to this offer!

You have the choice of working with v.5 or v.6 of Microsoft Visual C++. If you plan to use the latter, you should be careful to apply the required patch to the C++ SDK, which you will be asked about at the end of its installation.

Quartz v6.0

In the case of Quartz v6.0, you will need to install the C++ SDK and the SDK for Java into the same root directory on your PC. You can install these in either order (and uninstall either or both afterwards). For future Symbian SDKs (from v6.1 onwards), this process will be simplified, with Java and/or C++ development environments installed as part of a single process.

Only version 6.0 of Visual C++ is supported with Version 6.0 Symbian SDKs. Service Pack 3 should be applied.

Crystal v6.0 (Nokia 9210)
The process for Crystal v6.0 development is similar to that for Quartz, except that the SDKs are provided not by Symbian but by Nokia (***http://forum.nokia.com***), customized for the Nokia 9210.

Prior Knowledge

The demands on the JNI developer are not inconsiderable. You will need to have or gain some familiarity with all of the following:

- core Java APIs
- the Java Native Interface itself
- core EPOC C++ APIs (excluding GUI-based classes)
- the Symbian JNI project build environment
- strategies for inter-conversion of Java Strings and EPOC descriptors
- limitations imposed on Java native methods by the EPOC C++ threading model, its exception handling, and its restrictions on static global data

I shall assume that you already possess the first of these, and that you have obtained or can obtain the second or third from readily available resources.[3] Basic details will be filled in as necessary when this knowledge is used below. In particular, we shall go over the foundations of the Java Native Interface in section 13.3 below. The Symbian JNI project build environment will be presented in the remainder of this section. Inter-conversion between Java Strings and EPOC descriptors will be covered in section 13.4. Our discussion is rounded off by consideration of some more advanced topics in sections 13.5 and 13.6, and a final subsection dealing with troubleshooting.

13.2.2 Build Environment for JNI Projects

The Build Process

JNI projects are built using Microsoft's nmake and other command-line utilities, supplied with Visual C++. Note that `nmake.exe` should be on your Windows system path. The components of the overall build process are shown schematically in Figure 13.1. In that diagram:

- gray boxes indicate components that you provide
- boxes with rounded corners indicate tools
- shadowed boxes represent deliverables to be installed on the target device

[3] Details of the Java Native Interface can be obtained from ***http://java.sun.com/j2se/1.3/docs/guide/jni/index.html*** or from texts such as *Essential JNI: Java Native Interface*, Rob Gordon (Prentice Hall), ISBN 0136798950 and *The Java Native Interface: Programmer's Guide and Specification*, Sheng Liang (Addison-Wesley), ISBN 0201325772. Descriptions of core EPOC C++ APIs can be found in the documentation accompanying Symbian C++ SDKs.

The hatched gray and white box represents the **exports.def** file generated automatically by the build system (see below).

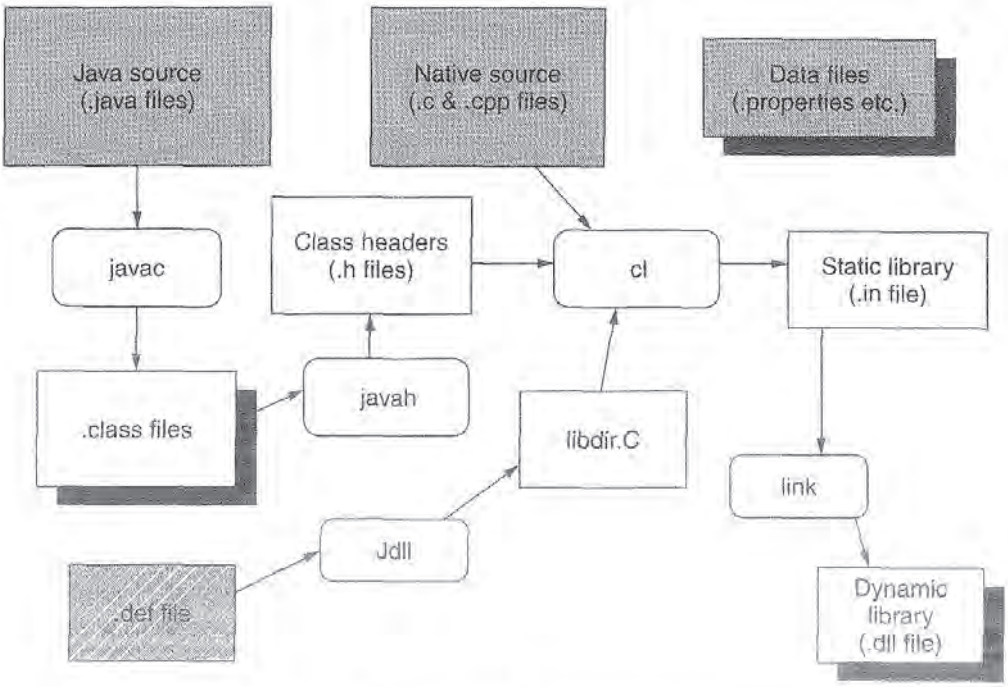


Figure 13.1 Overview of the JNI project build process.

You kick off the build process from a command prompt in the project's root folder by simply typing `nmake`. The process is controlled by a set of project-specific makefiles that you customize for your JNI project. These are described in the following section.

In fact there is slightly more to the command-line syntax than I have suggested. What I said is true if you are compiling a JNI project to be run on the udeb emulator. For the urel emulator use `nmake optimized`. To compile for a real device, use `nmake target=arm-unicode`. In the latter case, the .dll file will be created in **\erj\bin\arm-unicode**, from where it can be transferred to its final home in the **\System\libs** folder on the target device. Two other useful options are:

`nmake clean`
This removes intermediate (.obj) files. Usually, this is all you need to force clean recompilation of native code while avoiding the need to recompile Java class files.
`nmake clobber`
This removes everything, forcing clean recompilation both of native and of Java code.

More information can be found in the v6.0 Symbian SDK for Java Developer Library.

Build Control Files

To understand better this (admittedly rather complicated-looking) build system we shall focus on the InfoPrint1 example project from the Quartz v6.0 SDK (in **\erj\examples\Infoprint1**). The situation for EPOC Release 5 is similar. For details, consult the EPOC Release 5 SDK for Java documentation. Additional EPOC Release 5-specific information and examples can be found on ***www.symbiandevnet.com***.

You can build this JNI project for the Quartz udeb emulator by invoking `nmake` from a command prompt in the project's root folder. You can then run it by typing `apprun`. You should see "Hello from Java" displayed for a few seconds as shown in Figure 13.2, before the emulator closes down on the application's termination.



Figure 13.2 InfoPrint1 JNI example running in the Quartz emulator

The makefiles that control the process of building a project are as follows:

Makefile—this file provides directions used by the build system to compile the native code and build the associated .dll file.
classes.mk—this file controls the Java side of the build process.

You will need to create a version of each of these files to control your own JNI project. However, rather than creating them from scratch, it is easier if you simply modify appropriately the project-specific files **Makefile** and **classes.mk** provided for, say, the InfoPrint1 project.

If you take a look at the **Makefile** for building the InfoPrint1 native library you will see that it contains instructions for building the InfoPrint1 project. This is what it looks like:

```
binary = library
TOPDIR = ..\..
!include $(TOPDIR)\makefiles\make.defs
LIBRARY = infoprint1
FILES_obj = $(OBJ)\infoprint.$O $(OBJ)\infoprintdll.$O
!include $(TOPDIR)\makefiles\jnilib.mk
```

The variables you need to be aware of and may need to adapt from this file are as follows:

`TOPDIR`
This specifies the path from the project directory to the top-level directory **\erj** on the drive (or `subst`ed drive) on which your SDK was installed.

`LIBRARY`
This specifies the name of the generated native DLL. The name assigned to each native library you create must be unique. It should be the same as the name used in the lookup function name in **exports.def**, described later below.

`FILES_obj`
This is a list of the C++ object files to be linked in building the DLL. You can more usefully think of it as specifying the .cpp files which will be recompiled as necessary each time `nmake` is invoked. Note that `FILES_obj`, defined in the **Makefile** above, includes the stub file **E32Dll.obj**. You should copy the file **E32Dll.cpp** over verbatim to every new JNI project you create and include it in this way.

`EXTRA_LIBS`
Our example does not use this variable, but if you need to link to additional native libraries you will need to specify these. For example, to link to, say, **etext.lib** and **efsrv.lib** you would add a line of the form:

```
EXTRA_LIBS = $(TRGDIR)\etext.lib $(TRGDIR)\efsrv.lib
```

euser.lib is always linked to by default, so need not be mentioned explicitly.

If you take a look at **classes.mk** you will see that it lists all the classes used by the InfoPrint1 project. This is what it looks like:

```
FILES_class = InfoPrint.class
JNI_CLASSES = InfoPrint
CLASSPATH = .;$(CLASSES);$(LIBDIR)\classes.zip
.java.class :
$(JAVAC) -classpath $(CLASSPATH) $<
clobber :: FORCE
-@$(DEL) $(FILES_class)
```

The variables you need to be aware of and will need to adapt from this file are as follows:

`FILES_class`
This is a list of all the class files in your project, regardless of whether they have native methods or not. They must be specified together with their relative paths from the project root folder. If you modify any of the .java source files for these, they will be recompiled next time `nmake` is invoked.

`JNI_CLASSES`
This is a list of the names of the classes that declare native methods (or interfaces used by native methods). If you recompile any of these, `javah` will be invoked on them and a new associated header file generated in a subfolder **\CClassHeaders** of your project folder next time `nmake` is invoked. The class names must include the package name if the classes are not in the default package. For illustration of other changes that need to be made to deal with classes in non-default packages, see the project in **\erj\utilities\util** on the Quartz 6.0 SDK.

Source Files

Let's take a look next at the sources for the InfoPrint1 project. The Java sources are in **InfoPrint.java** and the corresponding C++ sources in **InfoPrint.cpp** and **E32Dll.cpp**. The `InfoPrint` class allows you to pass a string to a `print(...)` method which causes the string to be displayed at the top right of the screen for a few seconds, using a Symbian-specific native utility called an **InfoPrint**. The class itself is defined as follows (the `main(...)` method only having been omitted):

```
public class InfoPrint
{
      static {System. loadLibrary("InfoPrint");}

      private final static int KErrNone = 0;
      private static int err = 0;

      public static  native int print(String aText);
      ...
}
```

Two things are of note:

1. The `print(...)` method is declared to be `native`. This means that the Java class contains no implementation of the method. Rather it is implemented in native code accessed at run time from a DLL.
2. The DLLs which are to be searched at run time are specified through the `System.loadLibrary(...)` method. The code to do this is enclosed in

a static block, ensuring that the native library is loaded when the containing class is initialized, i.e. before any native methods are invoked.

All the real action, however, happens in the file **InfoPrint.cpp**. This contains the following preamble:

```
#include <e32std.h>              // RNotifier
extern "C"
{
#include "InfoPrint.h" // generated by javah
}
```

The system header file **e32std.h** is #included from **\epoc32\include** because it contains the definition of `RNotifier`, used in the body of the native method below.[4] In addition, the header files generated by the `javah` directive need to be #included, declared as `extern "C"` to indicate that the methods they define have C-style linkage (as do all Java native methods).[5] The native method itself has the following signature:

```
JNIEXPORT jint JNICALL Java_InfoPrint_print
 (JNIEnv* aJNI, jclass /*cls*/, jstring aText)
```

We note three things here:

1. The signature is prefaced with `JNIEXPORT` and the method name with `JNICALL`, rendering it consistent with that in the header file **InfoPrint.h** programmatically generated by `javah`. These macros serve merely to indicate that the method is to be exported from the DLL (as does the preface `EXPORT_C` in normal EPOC DLLs).
2. The name of the method is prefaced by `Java_InfoPrint_`. As a rule, native method names are of the form:

   `Java_<package qualifier>_<classname>_<method name>`

   where *`<classname>`* is the Java class in which the method is defined, *`<package qualifier>`* is the (underscore-separated) package in which that class is defined, and *`<method name>`* is the native method name declared in that class.

---

[4] As the methods specified in **e32std.h** are implemented in **euser.lib**, which is linked to by default in JNI builds, there is no requirement here for any `EXTRA_LIBS` entry in **Makefile**.

[5] As mentioned above, these header files are generated in a subfolder **\CClassHeaders** of your project folder. If your JNI functions make use of C++ classes defined by you as part of your JNI project (but not exported from the DLL for use elsewhere), you will probably want to define those classes in separate header files. A convenient place to put these is that same subfolder, where they will be automatically found by the build system. Note that they should not in this case be declared as `extern "C"`.

3. The Java String passed to the method from the Java Runtime as its only argument is received on the native side as its third argument. The first two arguments which have been inserted are respectively:

a `JNIEnv*` pointer which acts as a reference to the execution context in which the native method is running. It provides *inter alia* access to a function table using callbacks which can be made to the Java Runtime. This allows you, for example, to invoke Java methods and/or instantiate new Java objects from within the native method.

a `jclass` reference to the Java class on which the (static) method was called. If the method is not static, this will be replaced by a `jobject` reference to the object on which the method was invoked.

This information is summarized in Table 13.1

Table 13.1 Java v. native signatures

Instance methods	Class (static) methods
Java signature: `Public native void MyMethod(...);` Native signature: `JNIEXPORT Void ONICALL Java_MyClass_MyMethod (JNIEnv* aJNI, jobject obj, ...)`	Java signature: `public static native void MyMethod(...);` Native signature: `JNIEXPORT Void ONICALL Java_MyClass_MyMethod (JNIEnv* aJNI, jclass cls, ...)`

If you find this signature stuff at all confusing, no need to worry. You can always get the native signature you require out of the header file generated by `javah`, and cut-and-paste that into your .cpp file.

We shall delay consideration of the further details of the implementation of the native `print()` method until we have covered some more of the relevant details in ***Memory Management*** below.

13.3 Using Java Variables and Methods

Having achieved some mastery of the build mechanism for JNI projects, in this section we can turn our attention to the task of writing the code. Being an interface, the most important thing to understand about the Java Native Interface is how communication occurs across it. We shall start off by considering how primitive variables are shared across the interface, before going on to look at issues concerning objects, classes and arrays — including how to invoke static and instance methods of Java classes and objects, and how to construct new instances of classes and arrays. In the following section we go on to consider the particular complexities arising in relation to Java Strings.

13.3.1 Mapping Java Primitives onto EPOC Primitives

Java has eight primitive types, which get passed to/from native (or Java) methods by value: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`. All other types (even "Exceptions" are included here!) are passed by reference. When you pass any of these eight primitives as an argument to a Java native method, it is reinterpreted upon receipt as a C/C++ primitive. These primitives are identified in native code by having "j" prepended to their type names. And vice versa for the return variables from native methods.

So for example, in the Java signature of the `print()` method in the `InfoPrint` class above, we saw the return value was declared to be a Java `int`, whereas in the C++ signature of the implementation of the same method, the return value was declared to be a `jint`. The C++ compiler in practice interprets this name as identifying a `long` — a signed 32-bit integer; or in EPOC terms a `TInt32`. (The skeptics among you can confirm that this is the case by browsing the `typedefs` in the header files in **\erj\include\epoc32**).

In fact seven of these eight Java primitives are `typedefed` in this way to an appropriate EPOC primitive, as shown in Table 13.2,[6] `jlong`, Java's 64-bit integer type, being a special case, the story behind which we will take up at the end of this section.

Table 13.2 Equivalence relations of primitive types

Java type	C++ type	EPOC type
jboolean	unsigned char	TText8/TUint8
jbyte	char	TInt8
jchar	unsigned short int	TText16/TUint16
jshort	short int	TInt16
jint	(long) int	TInt32
jlong	<signed int64>	***
jfloat	float	TReal32
jdouble	double	TReal64

In other words, the names in the first and third columns of Table 13.1 can be used interchangeably in EPOC JNI code.[7] It might be mentioned here that if

[6] You should avoid taking these equivalence relations too literally in the context of literal strings. A Java String is in essence an array of jchars (16-bit Unicode characters). If its data is made available as ASCII on the native side, it is converted to (signed 8-bit) char* format. (Don't be misled by the common name "char" used with very different semantics between Java and C/C++!) If the data is further to be manipulated as an EPOC descriptor, it needs to be re-interpreted as an (unsigned) TText8*, as we shall go on to discuss. Of course, if the String's data is to be made available as Unicode, the conversion is the obvious one from jchar* to TText16*.

[7] This may seem like an unnecessary exercise in proliferation of technical verbiage. However, it is unavoidable, because of the need for the javah tool to be able to generate header files which are independent of the target environment; the tool cannot be assumed to know anything about how C++ primitives are referred to in EPOC or any other C/C++ programming environment, so must adopt a distinct, standard nomenclature suitable for use across all platforms.

a JNI function does not return a value it should be declared as a `void` function in the usual way—there is no `"jvoid" typedef`.

Another convention which you might find useful to know about is `jsize`, which is a synonym for `jint`. You will see this appearing in the signature of many of the predefined JNI functions declared in **\erj\include\jni.h**. It is typically used to pass the lengths of Java arrays or strings, where this information needs to be known explicitly in C++, but not in Java. Also, if you are working with `jboolean`, the values you should use to return results of `true` and `false` are `JNI_TRUE` and `JNI_FALSE`, # defined to be 1 and 0 respectively.

The Case of `jlong`

When a Java `long` integer is passed to a JNI function it is received in EPOC as a `jlong`, a 64-bit integer primitive. However, some care must be taken in manipulating or using this in EPOC code. When EPOC was formulated, there was no specification in the C++ language for a 64-bit integer primitive. For this reason it was chosen to handle 64-bit integers in EPOC not as primitives but as instances of a `TInt64` class. For most EPOC programming this is no big deal. But in JNI programming you have to beware that a `jlong` received from Java code is not a `TInt64` and lacks some of the facilities available to instances of that class. In particular:

- you cannot do division operations with a `jlong`
- you cannot pass a `jlong` to an EPOC method which specifies a `TInt64` as an argument.

However, `jlong` and `TInt64` are, conveniently, laid out in memory in identical ways. So you can convert a `jlong` to a `TInt64`, and vice versa by means of a suitable reinterpret cast on the address in memory. This is provided by a convenient EPOC macro, as is illustrated by the following code snippet which converts a `jlong` to a `TInt64` and back:

```
jlong jval1, jval2;
TInt64 int64val;
...
int64val = *REINTERPRET_CAST
      (TInt64*,&jval1);
jval2 = *REINTERPRET_CAST
      (jlong*,&int64val);
```

13.3.2 Using Java Objects in Native Code

Of course, as well as Java primitives, Java reference types can also be passed back and forth across the Java Native Interface: in particular, there is built-in support for Java objects, classes, arrays, strings, and exceptions. An object is

referred to as `jobject` and a class as `jclass`. There are various array types with names depending on the type of variables the array contains. We shall deal with strings and exceptions in separate sections below (see Sections 13.4 and 13.5). For details of the available options, see Sun's JDK1.1 JNI Specification at ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/types.doc.html # 15954***.

These types all have in common that in C++ they are no more than pointers to instances of an underlying dummy `jobject` class, representing a Java object, or else a trivial subclass thereof. In reality, of course they are not really pointers, because Java does not have such a concept. Rather they are handles into a local reference frame marshaled by the JNI environment mechanism, which converts them into references to Java objects; these the Runtime can work with directly in the usual way.

But you don't need to worry about the details of their C++ implementations when you use Java reference types in native code. Everything you need to know about their use is set out in ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/functions.doc.html***. We now discuss how you would in practice perform some of the basic operations on Java objects, classes and arrays.

Accessing the Fields of Java Objects

Having received a `jobject` as an argument in a JNI function, there are two obvious things you might want to do with it:

get or set one or more of its fields;

invoke one or more of its methods.

We shall deal with each of these in turn.

Notice nothing was said here about fields or methods being "public." It is a feature of JNI functions that they break encapsulation: once you have a reference to a `jobject`, you can access its fields and methods equally irrespective of whether they are declared with public, protected, default, or private scope.

The way you get or set a field is less than intuitive (as is most JNI manipulation of Java reference types, working as you are from "behind a fence"). For a field of a primitive type you would proceed as follows:

get a reference to the class of which the object is an instance in the form of a `jclass`;

get a reference to the field within the class in the form of a `jfieldID`;

invoke the appropriate getter and/or setter functions JNI provides for the purpose.

For example, the following method recovers an integer field `event` from a Java class `MyClass`, processes it depending on its value, then resets it to 0. Here and subsequently exception handling has been omitted for clarity. We will discuss in section 13.5 below the kinds of exceptions that can arise and strategies for dealing with them.

```
JNIEXPORT void JNICALL Java_MyClass_processEvent(JNIEnv* aJNI,
    jobject aObj)
    {
    jclass myClass = aJNI->GetObjectClass(aObj);
    jfieldID fieldID = aJNI->GetFieldID(myClass, "event", "I");
    jint event = aJNI->GetIntField(aObj, fieldID);
    ProcessEventCode(event);
    aJNI->SetIntField(aObj, fieldID, 0);
    }
```

Points to note here are:

1. Each of the JNI functions is called as a member of the `JNIEnv` instance, a pointer to which is passed in as an argument (`aJNI`) to the native method. In C++ implementations of the Java Native Interface such as Symbian's, a `this` pointer is implicitly passed on to all member functions of the `JNIEnv`, so need not be passed explicitly as it would in a C implementation. Don't therefore be confused by the appearance of an additional `JNIEnv*` pointer as the first argument to all JNI functions in Sun's JNI Specification: this relates only to C implementations.
2. The `GetFieldID(...)` function, as well as passing a reference to the relevant class, has to specify the name of the field sought and its signature, as strings. Here `"I"` indicates that the field is of integer type. Similar naming conventions exist for other primitives. For your reference, these are comprehensively documented in ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/types.doc.html#16432***. Alternatively you can discover the signatures of a class's fields and methods using the `javap` tool, as explained in the following section. The function `ProcessEventCode(...)` would be defined elsewhere in the project's C++ source file(s).

A suitable Java implementation of `MyClass` would be of the following form:

```
public class MyClass
{
     public native void processEvent();
     private int event;
     static {System.loadLibrary("mylib");}
}
```

with of course the native code linked in a library **mylib.dll**. Of course this is a rather artificial example as it would be a lot easier to pass the event code as an

argument, and reset it in Java code on return from the JNI function. But it does illustrate the principle.

The situation when you are recovering fields of reference type is similar. For example, if you want to recover a `java.util.Vector` field called `"items"` from a `jobject` reference, you need to know first that its signature is `"Ljava/util/Vector"`. (Don't forget that trailing semi-colon!) The format used here is the one used internally by the VM itself to identify classes. With that knowledge you can then recover a reference to the vector object with

```
jfieldID fieldID = aJNI->GetFieldID(myClass, "items",
    "Ljava/util/Vector;");
jobject items = aJNI->GetObjectField(aObj, fieldID);
```

Similarly if the object recovered were a Java String, except that an extra casting operation would be required in the final line

```
jstring items = (jstring) aJNI->GetObjectField(aObj, fieldID);
```

Again, if the field recovered were an array, it would have to be cast to the appropriate array reference type. The further functions required to recover array elements from such an array reference are discussed below.

Invoking the Methods of Java Objects

Now you've got your head round the JNI approach to getting and setting the fields of a class instance, it's time to look at how you would go about invoking its methods. The approach is exactly analogous, with `jmethodID` taking the role of `jfieldID` and method signatures replacing type signatures. To invoke a method of a Java `object` to which you have a `jobject` reference, you would proceed as follows:

get a reference to the class of which the object is an instance in the form of a `jclass`;

get a reference to the method within the class in the form of a `jmethodID`;

invoke the method by passing the `jobject`, the `jmethodID` and its arguments to one of the functions JNI provides for the purpose.

As with field getter functions, the JNI function you need to use to invoke instance methods depends on the return value specified in the method's signature. For example, `CallIntMethod(jobject, jmethodID, ...)` allows you to invoke a Java method taking arguments "..." and returning an `int`, and indeed returns that value as a `jint`. Distinct functions exist for each of the eight primitives and one for reference types (returning a `jobject`). Finally there is also one for invoking `void` methods. Furthermore, you can package the arguments you pass to your instance method in three different ways, so each

of these JNI functions in turn comes in three different flavors. For full details see ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/functions.doc.html# 16656***

As a concrete example, suppose we have a Java class `VectorOwner` with a native instance method `processVector()` and a `java.util.Vector` field called items:

```
public class VectorOwner
{
      public native int processVector();
      private java.util.Vector items = new java.util.Vector(10);
      static {System.loadLibrary("mylib2");}
}
```

In the native method we can recover the items field in the manner discussed:

```
JNIEXPORT jint JNICALL Java_VectorOwner_processVector(JNIEnv* aJNI,
    jobject aObj)
     {
     jclass myClass = aJNI->GetObjectClass(aObj);
     jfieldID fieldID = aJNI->GetFieldID(myClass, "items",
         "Ljava/util/Vector;");
     jobject items = aJNI->GetObjectField(aObj, fieldID);
```

To get, say, the capacity of this vector, we go on to call its capacity() method as follows:

```
     jclass vectorClass = aJNI->GetObjectClass(items);
     jmethodID methodID = aJNI->GetMethodID(vectorClass, "capacity",
         "()I");
     jint capacity = aJNI->CallIntMethod(items, methodID);
     return capacity;
     }
```

The main thing to notice here is the method signature `"()I"` passed to `GetMethodID(...)`. The closing `"I"` indicates that the method returns an `int`. The type of each of the arguments taken by the method should also appear in order between the preceding brackets in the signature; in this case there are of course none. The easiest way to get the right signature for your method is to use `javap` on the class. For the `VectorOwner` class defined above you would type at a command prompt:

```
>javap -s -private VectorOwner
```

The `-s` indicates that you are requesting field and method signatures to be printed out. The `-private` indicates that you want included all signatures down to private visibility. On Windows, the result of this command is as follows:

```
Compiled from VectorOwner.java
public synchronized class VectorOwner extends java.lang.Object
  /* ACC_SUPER bit set */
{
  private java.util.Vector items;
    /* Ljava/util/Vector; */
  public native int processVector();
    /* ()I */
  public VectorOwner();
    /* ()V */
  static static {};
    /* ()V */
}
```

A "V" in a signature indicates that the method returns void. There are of course other JNI functions allowing you to work more conveniently with the fields and methods of Java objects. In particular, it is possible to access fields and methods of their superclasses. The interested reader should consult ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/functions.doc.html***.

13.3.3 Using Java Classes in Native Code

We next take a look at two additional ways that you might use a `jclass` reference:

accessing static fields and methods;

constructing an instance of the class referred to.

Accessing Static Fields and Methods

The first of these is exactly analogous to working with instance fields and methods. The main difference is in the functions used to recover IDs, to get and set static fields and to invoke static methods: these functions take a `jclass` argument in place of a `jobject`.

As an example, suppose all the fields and methods of `MyClass` were transformed to be static:

```
public class MyClassWithStaticTypes
{
      public static native void processEvent();
      private static int event;
      static {System.loadLibrary("mylib");}
      ...
}
```

The native implementation of `processEvent()` would have to be changed as follows:

```
JNIEXPORT void JNICALL Java_MyClassWithStaticTypes_processEvent
    (JNIEnv* aJNI, jclass aClass)
    {
    jfieldID fieldID = aJNI->GetStaticFieldID(aClass, "event", "I");
    jint event = aJNI->GetStaticIntField(aClass, fieldID);
    ProcessEventCode(event);
    aJNI->SetStaticIntField(aClass, fieldID, 0);
    }
```

Notice that because we have defined the native method itself as static, a `jclass` reference is passed as an argument. So there is no need for a call to `GetObjectClass(...)` as previously. The situation is similar for invoking a static method from within native code, functions like `CallStaticIntMethod(JNIEnv*, jclass, jmethodID, ...)` taking the role of functions like `CallIntMethod(JNIEnv*, jobject, jmethodID, ...)`.

Constructing New Java Objects

You are not restricted to working with Java objects that already exist in the Runtime.[8] You can also instantiate new Java objects by invoking one or other of their constructors. There are two approaches to this:

using the `AllocObject(...)` function;

using the `NewObject(...)` function.

The first of these allocates memory only, allowing you to defer instantiation (to save time in a startup routine, for example). We will deal here, however, only with the second of these approaches, which combines allocation and instantiation. Full details of both can be found at ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/functions.doc.html# 16334***.

Let's suppose we modify our `VectorOwner` class so that `items` is not instantiated in Java code, but in the native `processVector()` method. The vector is instantiated (with a capacity of 10) in the native method using `NewObject(...)`,

```
JNIEXPORT jint JNICALL Java_ModifiedVectorOwner_processVector
    (JNIEnv* aJNI, jobject myObj)
    {
    jclass cls = aJNI->FindClass("java/util/Vector");
    jmethodID methodID = aJNI->GetMethodID(cls, "<init>", "(I)V");
    jint capacity = 10;
    jobject vector = aJNI->NewObject(cls, methodID, capacity);
```

[8] Nor are you restricted to classes that are already loaded. It is possible with the JNI function `DefineClass(...)` to take a suitably formatted byte array as input (possibly downloaded from a remote host) and load it as a new `java.lang.Class`, which is then available for use by the Runtime. The details of this are beyond the scope of the present discussion. But the capability is there for you on Symbian devices if you want it.

Then the vector is assigned to the `items` field:

```
jclass myClass = aJNI->GetObjectClass(myObj);
jfieldID fieldID = aJNI->GetFieldID(myClass, "items",
    "Ljava/util/Vector;");
aJNI->SetObjectField(myObj, fieldID, vector);
return capacity;
}
```

Points of note here are:

1. To obtain a `jclass` reference from a class name, rather than from a `jobject` reference, you must use the JNI function `FindClass(...)`. The class name passed as an argument to this (here `"java/util/Vector"`) is the full package-qualified class name, but using "/" rather than "." as the separator. Notice that unlike in field and method signatures, there is no opening "L" or closing ";".
2. You invoke the constructor for a class in a similar manner to any other Java method. To get the `jmethodID`, you must specify a method name of "<init>". All constructors are considered as `void` methods. The constructor we use here takes an integer argument — the capacity. Hence the signature passed is "(I)V".
3. The constructor can be invoked through the `CallVoidMethod (jobject, jmethodID, ...)` function if `AllocObject (jclass)` is first called providing the required `jobject` reference. Alternatively, as here, these two calls can effectively be rolled into a single call to `NewObject(jclass, jmethodID, ...)` function.
4. The assignment of the vector to the `items` field follows the pattern we discussed previously in relation to `MyClass`.
5. This is quite a painful experience to achieve what in Java was a single line of code!

For completeness, here's what the associated `ModifiedVectorOwner` class would look like:

```
public class ModifiedVectorOwner
{
    public native int processVector();
    private Vector items;
    static {System.loadLibrary("mylib4");}
```

We include a `main(...)` method which shows how you would go about calling the native method.

```
    public static void main(String[] args)
    {
```

```
        ModifiedVectorOwner cls = new ModifiedVectorOwner();
        int capacity = cls.processVector();
        System.out.println("Capacity of vector is:" + capacity);
    }
}
```

13.3.4 Using Java Arrays in Native Code

The handling of Java arrays in native code is a subject of some complexity that adds no fewer than 44 JNI functions to the total number at your disposal. This rather large total is made up five functions for arrays of each of the eight primitive types, three functions for object arrays and a generic `GetArrayLength(jarray array)` function returning a `jsize`, with obvious significance.

Object Arrays

Object arrays are the easiest to use, so let's look at them first. If an object array is passed as an argument to a native method it will be received as a `jobjectArray` (notice the upper case "A"). You can ascertain its length by passing it as an argument to `GetArrayLength(jarray array)`, although a better strategy might be to ascertain its length in Java code and pass that length to the native side as an extra argument in the native method, receiving it as a `jsize`: the length will be recovered at least four times faster in Java code than through the JNI function. You can then recover references to any individual element with the `GetObjectArrayElement(jobjectArray array, jint i)` function, which returns the `i`th element as a `jobject`. If this object is a Java string you might cast it down to a `jstring` and use the associated utility methods, but in most cases you'll just have to work with the raw object, recovering its class, methods, and fields in the (rather tortuous) manner described above.

If you're looking for a challenge, it is also possible to create and populate a Java array from within native code, starting with the JNI function `NewObjectArray(jsize length, jclass elementClass, jobject initialElement)` which returns a new `jarray` of size `length`, containing objects of type `elementClass`, all initialized to refer to `initialElement` (which will typically be `NULL`). Each element then needs to be instantiated (in the way described above) and assigned explicitly to the array with the JNI function `SetObjectArrayElement(jobjectArray, jsize, jobject)`. To assign the object `element` to be the `i`th element of array `theArray` you would call:

```
aJNI->SetObjectArrayElement(theArray, i, element);
```

As you can see, this process quickly becomes hard work, especially when you start to think about including checks for exceptions. A more likely

scenario is that you'll want to get, modify, and set individual elements of an existing array. But before going down even this road, you probably want to ask yourself whether you can't do the getting and setting on the Java side, bearing in mind here that there is a performance cost to crossing the JNI boundary.

The overhead of calling a JNI function from native code is roughly twice the overhead of calling a native function from Java, which is twice again the overhead of setting up a call to a Java method from Java.

Arrays of Primitives

The process for dealing with arrays of primitives is similar, except you have more flexibility because the data is laid out contiguously in memory, allowing elements to be recovered to the native side either all at once or in blocks or "regions," rather than one at a time as with object arrays. For concreteness, let's consider the case of an array of bytes.

Corresponding to `NewObjectArray(...)`, there is a `NewByteArray (jsize)` function serving a similar purpose: the only parameter passed is the size of the array to be constructed.

To recover all the elements of a byte array, use the `GetByteArrayElements(jbyteArray, jboolean*)` function, which returns a `jbyte*` pointer to the first element of a representation of the array. The Boolean value pointed to by the second argument will, on return of the method, indicate whether a copy of the array has been made. As you would normally not want to make use of this information, you would typically pass a `NULL` second parameter. You can then use either C-style pointer arithmetic or custom EPOC techniques (we'll illustrate both below) to access any individual element. As the pointer is not of type `const`, you can use it to make arbitrary changes to the values of existing elements, although you cannot add new elements.

When you have made any changes you wish *and* no longer need access to the data of the array, you should call the `ReleaseByteArrayElements(jbyteArray, jbyte*, jboolean*)` function. Here the second parameter is the pointer which was returned by the preceding `GetByteArrayElements(...)` call. The third parameter should always be set to `NULL`. This function then performs two tasks:

It ensures that, if the version of the Java byte array is a copy, any changes made to that copy are reflected in the original.

It then makes sure that the copied version is made available for garbage collection.

Similar considerations apply to arrays of the other primitive types.

13.4 Dealing with Java Strings

None of the previous discussion of Java objects, classes, and arrays has an EPOC-specific aspect to it, applying equally as it does to any C++-based JNI environment. However, we shall see in the present section, where we consider how Java strings can effectively be managed from within native Symbian OS code, that specific EPOC expertise starts to be needed.

Technically, since Java Strings are well-behaved Java objects, they can be manipulated and/or created from within native code by the methods discussed in the previous section. So in that sense there is nothing new to be said on the subject. However, given that string manipulation constitutes a substantial amount of the activity in most programs, to leave things this way would be to render the Java Native Interface cumbersome to the point of being well-nigh unusable. Thus a number of JNI utility functions have been provided to allow a Java String's data to be exposed as a character array on the native side. To take advantage of this under EPOC, you have to package this character array as an EPOC descriptor, thus making available a wealth of native utility functions that you can then use in processing the data. Then when you're done you have to do the backward conversions.

The basic JNI functions you have at your disposal are detailed at ***http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/functions. doc.html#4925***. If you are working with a Version 6 Symbian Runtime, you will also have at your disposal a number of extra JNI utility functions which were introduced in the JDK 1.2 JNI Specification, although we shall not go into the details of these below.

A Java String will typically be passed to a native method as a `jstring` argument (although it could equally be passed as a `jobject` which has an almost identical meaning). You manipulate its data through one of two sets of JNI functions, depending on whether you want to represent it in ASCII or Unicode format. We shall deal first with the latter case, Unicode being the required descriptor type for all system methods in Version 6.

13.4.1 Working with Unicode Data

The method you use to recover raw Unicode data is `GetStringChars (jstringaString, jboolean* iscopy), which returns a const jchar*` pointer to a contiguous array of Unicode characters. As always, this is a member of the `JNIEnv*` function table. As with arrays, the representation of the string may or may not be a copy of the version held in the Runtime. If you don't care either way, you can pass `NULL` as a second argument.

You will want at this stage to package the data up as an EPOC `TPtrC16` descriptor — EPOC doesn't have an indigenous string class — the appropriate constructor for which takes as arguments the aforesaid `jchar*` pointer *and* the length of the array. The latter can be recovered from the `jstring` by means of the `GetStringLength(jstring aString)` function which provides the length of the referenced Java string as a `jint` as its return value.

If you don't want to modify the data, the `TPtrC16` class provides all the utilities you should need for processing it. If you do need to make modifications, you must *absolutely* make a copy of the data before you try. Do this by seeding a `TBuf16` or an `HBufC16` with the `TPtr16`. The latter class is a little trickier to work with but, being defined on the heap, has the advantage that you don't need to define a maximum length *a priori*.

Never try to modify a Java String in place. You should always make a local copy of it on the native side and modify that instead.

Once you have finished with the `TPtr16`, because you either no longer have need of the data or else have made a copy of it, you have one task remaining. That is to tell the Java Runtime that you no longer have need of its copy of the String. If you do not do this, the memory reserved for it will not be made available for garbage collection, even if you set to `null` all references on the Java side to the String. This is a particularly serious memory leak if the data is a copy, i.e. (`*iscopy = JNI_TRUE`), as multiple copies of a string can thus be generated, potentially without limit. The way you notify the Java Runtime that you're done with the String is to call the `ReleaseStringChars(jstring, const jchar*)` function. The function's return type is `void`. Its first argument here is the `jstring` you passed to `GetStringChars(...)`. The second is the pointer you got back from that function.

For every call to `GetStringChars(...)` there must be a corresponding call made to `ReleaseStringChars(...)`. Otherwise a memory leak ensues. Similarly for the corresponding UTF versions of these functions discussed below.

Finally, if you want to return a string to the Java side, you will need to use the `NewString(const jchar*, jsize)` function to make a copy. The first parameter here is a pointer to an array of EPOC `TText16/TUint16` Unicode characters (which can be easily recovered from an EPOC 16-bit descriptor with its `Ptr()` method). The second parameter is the length of this array (easily recoverable from the descriptor via its `Length()` method). The return value is a `jstring` reference representing a brand new `java.lang.String` allocated on the Java side. Notice that, as with the `NewObject(...)` function discussed above, memory allocation on the Java side is done behind the scenes on your behalf.

So now we've discussed the theory, it's time to put all this together in an example.

The following Java class serves to pass the string `"Hello"` to a native method `appendString(...)` which appends `"world"` and passes it back to be written to the Java console.

```
class Strings
{
	public static native String appendString(String str);

	static {System.loadLibrary("mylib");}

	public static void main(String[] args)
	{
		String str = new String("Hello");
		str = Strings.appendString(str);
		System.out.println(str);
	}
}
```

Nothing clever here. The skill is in the crafting of the native method. The following implementation treats the string as Unicode, making use of all the methods we discussed (with exception handling again omitted):

```
JNIEXPORT jstring JNICALL Java_Strings_appendString(JNIEnv* aJNI,
    jclass, jstring aString)
	{
	const jchar* const ptr = aJNI->GetStringChars(aString, NULL);
	const jsize length = aJNI->GetStringLength(aString);
	TPtrC16 ptr16(ptr, length);
	TBuf16<20> buf(ptr);
	aJNI->ReleaseStringChars(aString, ptr);
	_LIT16(KWorld, "world");
	buf.Append(KWorld);
	return aJNI->NewString(buf.Ptr(), length + KWorld().Length());
	}
```

Points to note are:

1. It is worth making the pointer returned by `GetStringChars(...)` of type `const jchar* const` rather than just `const jchar*` so that there is no risk of it being reassigned before it is used in the corresponding call to `ReleaseStringChars(...)`.
2. The "16"s appended to the type names ensure that 16-bit versions of all methods are used. As a result, this native method compiles equally well with EPOC Release 5 (8-bit default assumption) as with v6 SDKs (16-bit default).
3. Use of a `TPtrC16` rather than a `TPtr16` helps protect the data on the Java side from accidental modification.
4. We specify that the `TBuf16` has maximum length 20. An attempt to store more than 20 characters would result in an EPOC Panic. An arbitrary amount of data can be appended if instead you use an `HBufC`.
5. `_LIT16(...)` is a macro creating a convenient non-modifiable descriptor type from a hard-coded literal string.

13.4.2 Working with ASCII data

For completeness, let's look at how this native method could be reimplemented using ASCII (8-bit) representations of the Java strings instead of Unicode. This would potentially be of greater use on the EPOC Release 5 than on the Version 6 Runtime. For convenience, I've given the method the same name as the previous version so it would work with the identical Java class as before.

```
JNIEXPORT jstring JNICALL Java_Strings_appendString(JNIEnv* aJNI,
    jclass, jstring aString)
    {
    const char* const ptr = aJNI->GetStringUTFChars(aString, NULL);
    const jsize length = aJNI->GetStringUTFLength(aString);
    TPtrC8 ptr8(REINTERPRET_CAST(const TUint8*, ptr), length);
    TBuf8<20> buf(ptr8); // copy the data to a modifiable buffer
        descriptor
    aJNI->ReleaseStringUTFChars(aString, ptr); // OK, original
        data no longer needed
    _LIT8(KWorld, ''world'');
    buf.Append(KWorld);
    return aJNI->NewStringUTF(REINTERPRET_CAST(const char*,
        buf.PtrZ()));
    }
```

The main point to note here is that JNI functions dealing with Unicode strings are replaced by corresponding UTF (8-bit) versions. The corresponding functions and their signatures are compared in Table 13.3.

Table 13.3 Comparison of JNI functions for Unicode and ASCII strings

Functions for Unicode strings	Functions for ASCII strings
`const jchar* GetStringChars( jstring, jboolean*)`	`const char* GetStringUTFChars( jstring, jboolean*)`
`jsize GetStringLength(jstring)`	`jsize GetStringUTFLength(jstring)`
`void ReleaseStringChars( jstring, const jchar*)`	`void ReleaseStringUTFChars( jstring, const char*)`
`jstring NewString(const jchar*, jsize)`	`jstring NewStringUTF(const char*)`

Other points to note here are:

1. The construction of the `TPtrC8` is complicated by the fact that unsigned data are required, while Java returns a signed `char*`. This is dealt with by means of a standard C++ reinterpret cast, provided by an EPOC macro in the manner shown.
2. The process is otherwise exactly analogous to the Unicode version of this method except in the last line. There, rather than using the `Ptr()` method

on the `Tbuf`, the `TPtrZ()` method is used instead. What's the difference? Well, it does exactly the same thing returning a pointer to the array of bytes, except that it then appends a terminating `NULL`. This is because `NewStringUTF(...)` requires a C-style string, determining the end of the string by the position of the `NULL` byte. Don't forget in this situation to ensure that the buffer descriptor you assign to hold the string is long enough to hold the appended byte.

3. A reverse reinterpret cast is also of course required to take us back from unsigned to signed data so that the preferences of the JNI function might be satisfied.
4. We have omitted in Table 13.3 mention of the new JNI string utility functions introduced in JDK1.2 which are available for use from Version 6.

13.5 Exception Handling

13.5.1 Differences Among Java, Symbian OS and JNI

The final `jobject` type we have not yet dealt with is `jthrowable`, which is used for dealing with Java exception objects in native code. We deal with these here in a separate context because there are EPOC-specific issues relating to the way the JNI utility functions provided for working with `jthrowable` are best used on the Symbian OS. So, before delving into APIs, let's look at the differences between the Java and EPOC approaches to exception handling, then go on to consider the JNI context.

Exceptions in Java

In brief, Java programs throw exceptions of pre-defined types in response to abnormal or exceptional conditions. These can be checked for by including the potentially troublesome code in a `try` block and coding appropriate responses to the various possible exceptions in `catch` blocks. Code that should be run independently of whether an exception occurs or not can be included in an optional `finally` block. More serious situations result in error conditions, which you are not under obligation to catch as the system provides default error handling by terminating the application. The idea of this is illustrated in the following pseudo-Java code sample:

```
if (livingDangerously)
try {
      String nonsense = myDodgyMethod(badParameter);
      sendMessageToBoss(nonsense);
}
catch (UnpleasantException e) {
      // deal with unpleasantness
}
```

```
catch (VerySeriousException e) {
      // hand in resignation
      System.exit(0);
}
catch (CatastrophicError e) {
      // No need to catch this one. The system will automatically
      // do the right thing and close the app down
}
finally {
      livingDangerously = false; // avoid repeating this bad
                                 // experience!
}
```

Here, the normal behavior would be for the `try` block to be run, then the `finally`. If an exception or error occurs (is "thrown"), the Java Runtime tries to "catch" it, running the code in one or other of the `catch` blocks, depending on the "seriousness." If the exception is not an instance of any of the named types, the system checks the stack context to see if this `try` block was nested inside another which could supply an appropriate `catch` clause. If none is found, a default system handler is used for the exception or error. Then the code in the `finally` block is run.

Exceptions in EPOC

Because this kind of exception handling consumes quite a lot of memory resources, a design decision was made when EPOC C++ was first implemented not to provide such functionality at a system level. Rather EPOC system methods which can misbehave will often return an integer (`TInt`) error value, a nonzero value (which can in practice be assumed negative) indicating that a problem occurred. The predetermined meanings of reserved values are defined in header files in the Symbian C++ SDKs, the most important being in **e32std.h**. You can write your methods using these error codes and others in a similar way. For example, in the InfoPrint1 example discussed previously, an attempt to create a client session with the EPOC "notifier" server can fail with one of several possible error codes being returned.

```
RNotifier notifier;
// start the notifier session
TInt err = notifier.Connect();
if (err ! = KErrNone)
      {
      // handle the error here
      }
```

Here `KerrNone` is 0, the value returned on normal completion of the method. It is possible that the notifier server is not running, in which case the `Connect()` method will return `KErrNotFound = -1`, one of the most common EPOC

error codes. Alternatively, the system may have run out of memory and have been unable to allocate the kernel-side resources necessary to set up the notifier session, in which case the value `KErrNoMemory = -4` will be returned. You can test for these values and take action appropriate to the particular error which has occurred.

Since not all methods can return an integer, EPOC also provides an alternative, more flexible approach closer to Java (and standard C++) exception handling. This involves allowing functions to "Leave" prematurely (typically through the `User::Leave(TInt)` function) and executing these functions in a "TRAP" harness. When a function leaves it does not return a value; but neither is an exception object generated. Instead the leave event simply has an integer error code associated with it (the `TInt` argument of `User::Leave(TInt)`). If the method is executed in a `TRAP` harness, the leave event is intercepted immediately. If not, the calling method leaves in turn, and so on up the call stack until a `TRAP` is found. Following the `TRAP`, the error code for the leave event can be checked. As discussed above, a method completing normally will return `KerrNone` whereas a method which leaves will return one of the predefined negative values indicating the reason why it left. For example, an EPOC descriptor buffer of length 10 can be assigned on the heap with the following code:

```
HBufC* buf;
TInt err;
TRAP(err, buf = HBufC::NewL(10);
if (err != KErrNone)
      {
      // handle the error here
      }
```

Note:

1. The fact that the static constructor method `NewL(...)` can leave is indicated by the trailing `"L"`. This is a mandatory practice in EPOC: a method which does not have a trailing `"L"` is assumed to make a contract with the rest of the world that it cannot in any circumstances leave; in other words, it is "safe" This is similar to the (compiler-enforced) requirement Java makes that you append a `"throws ..."` clause to the declaration of a method which throws any exception (other than a `RuntimeException`).
2. You may, if you wish, code your method to leave in certain circumstances from the error handling block following the trap, passing on the error code, e.g.

   ```
   if (err == KErrNoMemory) User::Leave(KErrNoMemory);
   ```

   This is equivalent to re-throwing a caught exception in Java. You will not be able to do this if your method is a Java native method, though, as these,

having no `TRAP` mechanism, are not allowed to leave. We'll return to this difficulty shortly.

Because every allocation of a heap variable can potentially fail (with `KerrNoMemory`), a convenient `new(ELeave)` operator is introduced in EPOC C++ which should be used in preference to the standard `new` operator. As might be expected, this results in a leave event with error code `KerrNoMemory` if problems occur. The allocation of more complex objects which perform secondary allocation of heap variables in their constructors is beyond the scope of this introductory review of EPOC exception handling. If you're interested, you can find out more under the heading "Two Phase Construction" in the C++ Version of the Symbian Developer Library.

Clearly a requirement to provide exception handling code in the manner just described to perform appropriate cleanup following every heap allocation would be excessive. For this reason, EPOC also provides a cleanup stack where, in the context of a method that can leave, you can have the system store pointers to any memory allocations your method makes and is responsible for deallocating. If that method should leave, you don't have to worry: appropriate destructors will be called for all the pointers deposited on the cleanup stack. As the syntax for this is a little complex, we deal with it below in a separate section.

Error conditions from which the system finds it has no way of recovering result in an EPOC `Panic` which causes the associated process to be terminated. A `Panic` name and number is in this case reported to the user in a standard dialog box. The nearest equivalent to this behavior in Java is the throwing of an unhandled `java.lang.Error` in a Java program.

Exceptions in JNI Functions

The situation with JNI functions is something of a hybrid between that of Java and EPOC methods. If a problem arises in the execution of a JNI function, a standard Java exception or error may get thrown. However, these will *not* be detected by default by the calling native method when the latter regains the stack context. Rather it will receive a value if it is expecting one and continue unabashed. Of course the value it receives will likely be `NULL`—or, if the return value is a primitive, the default value for that primitive. We will return below (**JNI Functions**) to the question of how such exceptions can be detected.

Another possibility is that the occurrence of a problem during the execution of a JNI function may be indicated by the return of a `NULL` value, with *no* exception thrown, more along the lines of the EPOC model. You will need to check the JNI Specification for details of which exceptions, if any, can be thrown by any particular method. You can then set about implementing an appropriate exception handling strategy.

For example, the `GetObjectArrayElement(...)` method discussed above can throw a `java.lang.ArrayIndexOutOfBoundsException`

and return NULL, while the GetFieldID(...) method can throw any of the following errors:

- java.lang.OutOfMemoryError
- java.lang.NoSuchFieldError
- java.lang.ExceptionInInitializerError

again returning NULL. The GetStringChars(...) method on the other hand throws no exceptions, indicating problems only by the return of a NULL value.

As we shall see shortly, a consequence of all this is that your choices are maximized as to whether you want to catch and handle exceptions or errors on the native side or on the Java side. Of course, if they are Java errors rather than exceptions, you can *only* handle them on the native side as, once they are caught on the Java side, they will be handled by the system shutting down your application.

You need to be aware that JNI functions do not check whether the object references they have received have been initialized. It is wholly your responsibility to ensure *before* calling a JNI function that no NULL references are passed as arguments. If you make a mistake here, the result is not predictable in terms of the list of possible Java exceptions or errors declared in the JNI Specification. Typically an EPOC Panic will result and your application will immediately be closed down.

13.5.2 Handling Exceptions on the Native Side

Suppose now you have detected an exception condition in the context of a native method. You have a choice of whether you want to handle it locally or in Java code. Either course may be appropriate, depending on the circumstances. In particular, if you are in a situation where you have your hands full with partially constructed objects in heap memory and something goes wrong, you will almost certainly want to do some, if not all, exception handling on the native side where you have fairly direct access to the heap through pointers or session handles. If the problem is something like a file not being found, or not being available for opening, you are less likely to want a predetermined response and might want to present the end user with a choice of what to do next. This would usually best be done from the Java side, say by posting a dialog. Let's look briefly first at the case of handling errors in native code.

EPOC Functions Which Can Leave

One of the first problems that must be considered is what to do if you decide the appropriate response in a particular situation is for your Java native method

to leave.[9] This is of course not allowed because a Java native method is not executed from within a TRAP handler. What you must do in this case is to turn the Java native method into a proxy for a worker method which does the real work (and can leave). The body of the Java native method consists simply of a TRAP in which the worker method is called. Following that you can process any error codes (exceptions) returned.

For example, consider the following skeleton of a function called `ProcessStringL(...)` which serves to perform the real work of a Java native method called `ProcessString(...)` declared in a class called `StringClass`. If the string is empty, the method leaves with a suitable error code:

```
void ProcessStringL(JNIEnv* aJNI, jclass aClass, jstring aString)
    {
    jint length = aJNI->GetStringLength(aString);
    if (length == 0)
        {
        User::Leave(KErrArgument);
        }
```

We can then allocate a buffer on the heap to hold the Java String, do some processing of the String and finally delete the buffer.

```
    HBufC* buf = HBufC::NewL(length); // leaves if out of memory
    // copy aString to buf, do some parsing, etc.
    // ...
    delete buf;
    return;
    }
```

An implementation of the driver method would then be something like the following:

```
JNIEXPORT jint JNICALL Java_StringClass_ProcessString(JNIEnv* aJNI,
    jclass aClass, jstring aString)
     {
     TInt error;
     TRAP(error, ProcessStringL(aJNI, aClass, aString));
     switch(error)
          {
          case KErrNone:
               break;
          case KErrArgument:
               // handle case where string is empty
               break;
          case KErrNoMemory:
```

[9] The reason we might want to do this will become clearer when we study the use of the EPOC `CleanupStack`

```
            // handle case where HBufC allocation fails
            break;
        }
    return error; // further error handling can be done in Java
    }
```

Notes:

1. Although `ProcessStringL(...)` may leave, any native-side cleanup required as a result of the leave can be performed in the `JNICALL` in the code following the `TRAP`.
2. It is possible to execute methods which return a value in a `TRAP` handler. For example, if this method returned a `jstring`, you would write something like:

```
TInt error;
jstring str;
TRAP(error, str = ProcessStringL(aJNI, aClass, aString));
```

3. Although we could execute `HBufC::NewL(...)` in its own `TRA` handler, we avoid doing so. Since we have named `ProcessStringL(...)` appropriately to indicate to a user that it can leave, we are quite happy in the event that `HBufC::NewL(...)` leaves to allow that event to be passed on to the `TRAP` handler in the calling method. In general it is good practice to avoid where possible the nesting of `TRAP` handlers, as they can impact significantly on performance as well as using up valuable RAM resources.
4. Finally, by returning the error code to the Java side, you get a second chance to handle the exception there, as we shall see in Section 13.5.3.

JNI Functions

I mentioned above that Java exceptions thrown in the context of JNI functions are not detected by default by the calling native method. What in fact happens when a Java exception or error occurs in a JNI function is merely that an "exception pending" flag is set on the Java side. This flag is examined and action taken only when the native method completes and full control is returned to the Java Runtime. If it is set, the last Java exception to occur gets thrown. You can find out if it is currently set or not with a call to the JNI function `ExceptionOccurred()` which returns a `jthrowable` pointer to an impending exception or error, or `NULL` if there is none. Otherwise you can use `ExceptionCheck()` which returns a `jboolean` indicating whether or not an exception (or error) is pending.[10] It is strongly recommended that you

[10] Although `ExceptionCheck()` was not defined until the JDK 1.2 JNI Specification and is not available for use with Symbian's EPOC Release 5 Runtime, it has been implemented in Symbian's v6.0 pJava Runtime.

take advantage of these functions. If you find an exception has occurred you can do one of two things:

1. Ensure your native method returns immediately, i.e. without any further calls to JNI functions (which are no longer guaranteed to work properly).
2. Clear the exception with a call to the JNI function `ExceptionClear()` and initiate your own exception handling strategy.

If you find that a Java exception has occurred in the execution of a JNI function, you should ensure either that your native method returns directly or that an `ExceptionClear()` call is made before any further JNI functions are called.

With this information, we might now reimplement the `VectorOwner.processVector(...)` native method described above as follows:

```
JNIEXPORT jint JNICALL Java_VectorOwner_processVector(JNIEnv* aJNI,
    jobject aObj)
      {
      jclass myClass = aJNI->GetObjectClass(aObj);
      jfieldID fieldID = aJNI->GetFieldID(myClass, "item",
          "Ljava/util/Vector");
```

As the preceding call can fail with an error we must introduce exception handling at this stage. The easiest approach is simply to have the method return and let the exception be thrown on the Java side.

```
if (aJNI->ExceptionCheck())
       {
       return 0; // throw the error
       }
```

As `GetFieldId(...)` returns `NULL` in circumstances where an error occurs, you could avoid the overhead of the use of the JNI function `ExceptionCheck(...)` and use `if (fieldID == NULL)`.

```
jobject items = aJNI->GetObjectField(myObj, fieldID);
jclass vectorClass = aJNI->GetObjectClass(items);
jmethodID methodID = aJNI->GetMethodID(vectorClass, "capacity",
    "()I");
```

As `GetMethodId(...)` can fail too, exception handling is required again. For variety, let's use the `ExceptionOccurred(...)` function this time:

```
if (aJNI->ExceptionOccurred() != NULL)
      {
      return 0; // throw the error
      }
```

Because its parameters are now assured valid and because the `capacity()` method is itself guaranteed to succeed, the final call cannot now fail.

```
jint capacity = aJNI->CallIntMethod(items, methodID);
return capacity;
}
```

13.5.3 Handling Exceptions on the Java Side

Developing Reusable Classes

If you can't handle an EPOC exception satisfactorily from native code, the best approach is generally, as has been suggested above, to pass the integer error code (or `KerrNone`) back to the Java side as a `jint`. The way you would construct the Java class that declared the native method depends on how you want it to be used. For example, if that class were an integral part of a self-contained application you are shipping, you might choose to work directly with the integer EPOC error codes returned, writing error handling code directly into the class that calls the native method. If intervention were required by the application user, better design practice would be to post a message/event that would be detected by a suitable UI class.

If, however, you are writing a package intended to be used by other Java developers in their application development, who might potentially know little or nothing about EPOC C++, you would want a different strategy, masking from them the details of the EPOC error codes, but allowing them (nay, encouraging them!) at the same time to introduce custom exception handling in their code.

We consider now how this might be achieved by re-implementing our Java class `MyClass` with full exception handling as `MyClassWithExceptions`. Let's look first at the implementation of the native method. The first thing we have to do is change the native method from type `void` to make it return an EPOC error code as a `jint`. Let's change its name at this stage from `processEvent` to `ProcessEvent`, for reasons which will shortly become clear. Suppose also for interest that the subsidiary `ProcessEventCode(event)` function, instead of being of type `void`, returns an integer error code, say `KerrNoMemory` or `KerrNone`.

```
JNIEXPORT jint JNICALL Java_MyClassWithExceptions_ProcessEvent
    (JNIEnv* aJNI, jobject aObj)
      {
```

The attempt to recover the class reference cannot fail, but the call following this can, as we have discussed, conceivably return `NULL` and throw errors:

```
jclass myClass = aJNI->GetObjectClass(aObj);
jfieldID fieldID = aJNI->GetFieldID(myClass, "event", "I");
```

If we detect an exception or error, we return immediately and allow it to be thrown:

```
if (fieldID == NULL)
      {
      return 0; // throw the error
      }
```

Of the remaining methods, only `ProcessEventCode(...)` can generate further problems. We deal with these by simply returning any EPOC error code generated. Otherwise we can return `KerrNone` to indicate successful completion:

```
jint event = aJNI->GetIntField(aObj, fieldID);
jint error = ProcessEventCode(event);
if (error != KErrNone)
     {
     return error;
     }
aJNI->SetIntField(aObj, fieldID, 0);
return KErrNone;
}
```

Next we provide an implementation of the Java class itself. First, the native method and error codes are declared private so third-party developers cannot access them directly:

```
public class MyClassWithExceptions
{
      private final static int KErrNone = 0;
      private final static int KErrNoMemory = -4;
      private native int ProcessEvent();
}
```

Then a public wrapper method with (almost) the same name is defined which calls the native method and initiates error handling based on the error code returned. My habit is to use an opening upper case letter to indicate the private native method (EPOC-style) and an opening lower case letter to indicate the associated public wrapper method (Java convention). You might prefer to distinguish the private native method by prepending or appending a "_", following naming conventions for private Java methods common elsewhere. You should follow the example here of declaring explicitly all the exceptions and errors that can occur in the throws clause of JavaDoc comments for the public wrapper method. This will serve as a flag to encourage the final user of your package to include appropriate handling in their code.

```
/**
* Processes an event in native code
* <p>
```

```
 * @throws NoSuchFieldError: this should not occur in practice.
 * @throws ExceptionInInitializerError: this should not occur in
 * practice.
 * @throws OutOfMemoryError if the system runs out of memory.
 * @throws RuntimeException if an unexpected EPOC error occurs
 * in native code
 */
public void processEvent()
{
     int error = ProcessEvent();
     if(error != KErrNone)
     {
          HandleEPOCError(error);
     }
}
```

Finally, the error handling method examines any EPOC error code returned in a switch statement and throws an appropriate Java exception, with a (hopefully) meaningful message. Make sure you provide a `default` option in your switch statement as you may not have foreseen all the possible EPOC errors which could occur in your native method.

```
private void HandleEPOCError(int error)
{
     switch(error)
     {
     case KErrNoMemory:
          throw new OutOfMemoryError();
     default:
          throw new RuntimeException("Unexpected EPOC error:"
              + error);
     }
}
```

It is by no means necessary to process the errors in a separate method like `HandleEPOCError(int)`. We have taken this approach because it lends itself more readily to generalization: in practice there may be several native methods, each with their own public wrapper method, but sharing a single error handling method.

Throwing Java Exceptions from Native Code

As we have seen, Java exceptions and errors can occur during the execution of certain JNI functions. Although our strategy above was to handle exceptions by canceling them and returning EPOC error codes, it may be that ultimately you want that exception to be thrown on the Java side anyway. In such a case it would be wasteful of resources to cancel it in native code, i.e. delete the `Exception` object, then instantiate and throw an almost identical new `Exception` object on return to Java control. Instead your strategy should be, as

soon as the occurrence of an exception is detected, to have the native method return. The value returned is immaterial as it will not be received. Instead control will pass immediately to the Java exception handling mechanism which will handle the pending exception.

Remember that what in fact happens when a Java exception or error occurs in a JNI function is merely that an "exception pending" flag is set on the Java side. This flag is examined only when the native method completes and full control is returned to the Java Runtime.

Although it is an approach generally to be avoided as less efficient, Java exceptions and errors can also be instantiated and thrown from within native code. This can conveniently be accomplished with the `ThrowNew()` JNI function. For example, if you wanted to intercept a `NoSuchFieldError` thrown by a `GetFieldID(...)` JNI call and throw a `NoSuchFieldException` instead, you could try something like the following:

```
// This code shows why ThrowNew() has limited utility!
fieldID = aJNI->GetFieldID(myClass, "item", "Ljava/util/Vector");
if (fieldID == NULL) // exception occurred
      {
      aJNI->ExceptionClear();
      jclass excClass = aJNI->FindClass
          ("java/lang/NoSuchFieldException");
      jint notThrown = aJNI->ThrowNew(excClass,
            REINTERPRET_CAST(const char*, _S
          ("items field not found")));
      if (notThrown)
            {
            aJNI->ExceptionClear();
            }
      return KErrNotFound;
      }
```

The `ThrowNew(...)` function takes two arguments:

1. a `jclass` reference to the subclass of `java.lang.Throwable` which you want to throw, usually recovered beforehand with the `FindClass(...)` function.
2. a C-style null-terminated string representing the detail message passed to the exception. This can conveniently be generated with the EPOC `_S(...)` macro and a reinterpret cast to convert from unsigned to signed data format.

Two problems are however evident with the above approach:

- The `FindClass()` function is not guaranteed to succeed and may indeed throw an exception. A NULL return value ought therefore to be checked for and appropriate error handling code included.
- The `ThrowNew()` function can also fail, a situation which will be indicated by a nonzero (in fact negative) integer value being returned to `notThrown`.

Fortunately it cannot generate any exceptions other than the one you are trying to throw!

We have indicated how the value of `notThrown` might be tested. But this and the `FindClass(...)` problem illustrate the point: your attempts to throw the exception of your choice may themselves fail. So you are obliged to resort to EPOC-style error reporting. In which case you have to handle these as well as the Java exceptions on the Java side. So why bother with the Java exceptions in the first place?

One last point we should consider is declaring the exceptions that a native method can throw. If you delay throwing any exceptions until *after* control is returned to the Java Runtime, the compiler will force you to declare these exceptions in a `throws` clause. So they cannot be accidentally overlooked. But if you throw them from the *native* side, a Java compiler has no way to know that these exceptions can arise so will not care if you declare them as thrown or not by the native method. It is strongly advised that, if you do for any reason decide to allow a Java exception to be thrown from native code, you declare that fact explicitly by attaching an appropriate throws clause to its Java declaration. That way the compiler will ensure for you that any method calling your method implements an appropriate level of exception handling.

13.6 Memory Management

As our final topic we consider how the memory management models of EPOC and Java interact, giving attention to some of the issues that can arise when Java objects are created, or EPOC server sessions opened, in Java native methods. But first we must introduce the EPOC cleanup stack.

13.6.1 Using the EPOC Cleanup Stack

We saw above (**Functions which can leave**) how a `TRAP` harness can be used to deal with EPOC functions that can leave. However, it is inconvenient and inefficient to surround every such function by a `TRAP` harness. A better mechanism is needed. The solution EPOC provides is the cleanup stack. This is used as follows to protect memory allocated on the heap from being orphaned in the case of a leave:

```
void doExampleL()
      {
      CExample* myExample = new (ELeave) CExample();
```

No problem so far. If a leave occurs in this allocation it is safe. We are free to proceed and do something that cannot leave:

```
myExample->iInt = 5;
```

But before we do something that can leave, we must protect the `CExample` we have successfully created on the heap: for this we push the pointer onto the cleanup stack:

```
CleanupStack::PushL(myExample);
myExample->DoSomethingL();
```

If `DoSomethingL()` leaves, the cleanup stack mechanism will ensure that `myExample` (and any other objects on the clean up stack) get deleted and any necessary destructors called. Otherwise we can pop it off the cleanup stack and delete it explicitly.

```
CleanupStack::Pop();
delete myExample;
}
```

Alternatively the last two lines of this method can be combined into the single command:

```
CleanupStack::PopAndDestroy();
```

Further information on use of the EPOC cleanup stack can be found in the Symbian Developer Library C++ edition on the SDK. We will make extensive use of this idiom in the remainder of our JNI code examples.

13.6.2 Revisiting Java Strings

At this point we pick up again our discussion of the handling of Java Strings. With understanding of JNI exception handling strategies and the EPOC cleanup stack at our disposal, we are in a position to cover two further, more advanced, topics.

Java Strings and the Cleanup Stack

A problem you will face if you want to use Java Strings in native methods that can leave is how to guarantee that all the necessary calls to `ReleaseStringChars()` and/or `ReleaseStringUTFChars()` will be made, and the associated memory resources allocated by the Java Runtime freed up, even in the event of a leave occurring. The EPOC strategy for dealing with such a problem is of course the cleanup stack. The problem is that, since EPOC doesn't understand how to clean up Java Strings (which process of course needs two pointers, not just one), putting a pointer to the string on the cleanup stack is not an available option.

Not surprisingly, Symbian's Java engineers confronted precisely this problem in implementing the Java Runtime on the Symbian OS. The solution that they came up with and that I'd like to share with you now was to provide a wrapper class around a `jstring` with a constructor which:

- obtains a copy of the string's data and pins it in heap memory with a call to `GetStringChars(...)`,
- converts the resultant `jchar*` into a `TPtrC16` pointer descriptor,

and a destructor which releases the memory associated with the pinned Java String with a call to `ReleaseStringChars(...)`. You can instantiate this class either on the stack or on the heap. In the former case the destructor is guaranteed to be run whether the method returns or leaves. In the latter case, you can ensure the destructor is run in the case of a leave by putting a pointer on the cleanup stack.[11]

Below is an implementation of a class called `RJavaString` with this functionality. It should be placed in a header file in **\erj\include**.

```
# include <e32std.h>
# include <jni.h>

class RJavaString: public TPtrC16
      {
public:
      IMPORT_C RJavaString(JNIEnv* aJNI, jstring aString);
      inline ~RJavaString();

private:
      // prevent the following inherited methods from being used
      RJavaString(const RJavaString&);
      RJavaString& operator=(const RJavaString&);
      void Set(const TUint16*,TInt);
      void Set(const TDesC16&);
      void Set(const TPtrC16&);

private:
      JNIEnv* iJNI;
      jstring iString;
      jboolean iInitialised;
      };

inline RJavaString::~RJavaString()
      {
      if (iInitialised) iJNI->ReleaseStringChars(iString, Ptr());
      }

RJavaString::RJavaString(JNIEnv* aJNI, jstring aString)
     : iJNI(aJNI), iString(aString), iInitialised(JNI_FALSE)
     {
     if (iString)
```

[11] There is an issue if you pop the pointer off the cleanup stack and don't destroy it, but cache it for use in a subsequent native method call. As discussed in the following section, you will have to take care the second native method doesn't get called on a different Java thread from the first. This is not a trivial task.

```
        {
        this->TPtrC16::Set(iJNI->GetStringChars(iString, NULL),
              iJNI->GetStringLength(iString));
        iInitialised = JNI_TRUE;
        }
    }
```

You should `#include` the above header file in any source files using this functionality. Instantiate an instance of the class on the stack as follows:

```
void MyMethodL(JNIEnv* aJNI, jstring aText)
    {
    RJavaString myString(aJNI, aText);
```

Check the conversion was successful before continuing:

```
if(myString.Ptr() == NULL) User::Leave(KErrNoMemory);
```

There is no problem now if the following leaves:

```
DoSomethingL(myString);
}
```

Notice that the situation where the constructor's call to `GetStringChars(...)` fails can be detected by the `RJavaString` having a `NULL` pointer. This should be checked in the way shown immediately following the `RJavaString`'s construction. As such failure is only likely to occur if the system is out of memory, a leave with `KerrNoMemory` is an appropriate response.

As a final note, if you are doing intensive work with arrays of Java primitives, you may want to write yourself a similar utility to ensure all array elements pinned in memory on the Java side by a native function get released if that function leaves.

Handling cp1252 Data

A problem we faced when we sought to handle ASCII-based Java Strings in native code was that the character arrays returned by the JNI function `GetStringUTFChars(...)` are of *signed* data where EPOC descriptors require *unsigned* data. This forced us to perform inelegant reinterpret casts. But there is a good reason why EPOC descriptors are perverse in this way: the removal of the sign bit allows an extra bit for encoding so that 256 different characters can be encoded instead of ASCII's 128. The extra 128 characters can be assigned in a number of different ways depending on the encoding you choose to work with. The default used for screen display in the emulators in Symbian SDKs is "code page 1252," which allows representation of the alphabetic characters required for most West European languages and

includes the euro symbol €. Under EPOC Release 5 you can see the 128 non-ASCII characters in cp1252 (numbered 128–255) displayed when you type SHIFT+CTRL+C.

You can take advantage of this attribute of EPOC 8-bit descriptors and represent Java strings involving, e.g. cp1252 data compactly in native code, not as 16-bit Unicode but rather as 8-bit data. This would be particularly advantageous if, for example, you were communicating with another Symbian device under a constraint of limited bandwidth. The Symbian OS provides a mechanism to carry out the necessary conversions in both directions: `CCnvCharacterSetConverter` class.

To go from a non-ASCII Java Unicode string to an EPOC 8-bit descriptor:

1. Recover the Unicode data of your Java string as described above into an `RJavastring`.
2. Prepare a `CCnvCharacterSetConverter` instance with a connected file server session specifying the encoding of your choice, e.g. cp1252.
3. Use it to convert the `RJavaString` (a 16-bit pointer descriptor) to an 8-bit descriptor and store the result in a prepared `TBuf8` buffer. This should succeed provided the buffer is long enough and the Java string contains only characters covered by the specified encoding.

Likewise, to go from a non-ASCII EPOC 8-bit descriptor to a Java Unicode string:

1. Prepare a `CCnvCharacterSetConverter` instance with a connected file server session specifying the encoding of your choice.
2. Use it to convert the 8-bit descriptor to a 16-bit descriptor and store the result in a prepared `TBuf16` buffer. This should succeed provided the buffer is long enough.
3. Create a Java String from the `TBuf16` in the standard way.

We can use these techniques to modify our `String` class above to handle French Strings. Here's our new Java class:

```
class FrenchStrings
{
      public static native String appendString(String str);

      static {System.loadLibrary("mylib");}

      public static void main(String[] args)
      {
            String str = new String("Voilà");
            str = FrenchStrings.appendString(str);
            java.awt.Frame frame = new java.awt.Frame();
            frame.add(new Label(str));
            frame.pack();
```

```
            frame.show();
        }
    }
```

Notice I have chosen to display the string returned in an AWT context rather than sending it to `System.out`. This is because on the emulator if a DOS command prompt is used for console output the non-ASCII characters do not get interpreted according to the cp1252 convention as we wish. The following implementation of the native method `appendString(...)` appends `"mon frère"` to the incoming string `"Voilà"` and returns the result, but working with 8-bit descriptors for the string manipulation.

```
#include <e32std.h>          // for descriptors
#include <f32file.h>         // for RFs
#include <charconv.h>        // for CCnvCharacterSetConverter
#include <javastrings.h>     // for RJavaString

extern "C"
{
#include "FrenchStrings.h"
}

jstring AppendStringL(JNIEnv* aJNI, jstring aString)
    {
    RJavaString ptr16(aJNI, aString);
    TBuf8<20> buf8;
```

The `NewLC()` function in the following line serves to instantiate a new `CcnvCharacterSetConverter` *and* to put it on the `CleanupStack`:

```
CCnvCharacterSetConverter* converter = CCnvCharacterSetConverter
    ::NewLC();
```

A file session has to be connected and made available to the converter to allow it to open the file specifying the chosen encoding, in this case cp1252. Because file session resources are held and controlled kernel-side, the session should be `Close()`ed rather than deleted when no longer needed. You ensure cleanup of the session in the event of a leave by use of the `CleanupClosePushL(...)` method.

```
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
converter->PrepareToConvertToOrFromL(KCharacterSetIdentifierCode
    Page1252, fs);
```

The conversion to an 8-bit descriptor can then be made and the string modified.

```
User::LeaveIfError(converter->ConvertFromUnicode(buf8, ptr16));
_LIT8(KWorld, "mon frère");
buf8.Append(KWorld);
```

Finally we can make the conversion back to a 16-bit descriptor, then to a Java String. The extra integer passed to the converter is an advanced feature, the normal input value being 0. The `CleanupStack` call closes the file server session and deletes the converter object.

```
TInt state = 0;
TBuf16<20> buf16;
User::LeaveIfError(converter->ConvertToUnicode(buf16,buf8,state));
CleanupStack::PopAndDestroy(2); // fs, converter
return aJNI->NewString(buf16.Ptr(), buf16.Length());
}
```

The JNI method itself then simply calls the above function in a trap handler.

```
JNIEXPORT jstring JNICALL Java_FrenchStrings_appendString (JNIEnv*
    aJNI, jclass, jstring aString)
     {
     jstring str = NULL;
     TInt err;
     TRAP(err, str = AppendStringL(aJNI, aString));
     if (err)
          {
          // throw a Java exception here if wished
          }
     return str; // NULL if AppendStringL fails
     }
```

13.6.3 Revisiting Java Arrays

The array manipulation methods we described previously are adequate to allow you to do anything you want to with Java arrays. However, you may find it more efficient on occasions, particularly if the array is large and you only need to access a small part of it, to take advantage of the array region getter and setter functions. For the case of byte arrays these are:

- `GetByteArrayRegion(jbyteArray array, jsize start, jsize len, jbyte* pArray)`
- `SetByteArrayRegion(jbyteArray array, jsize start, jsize len, jbyte* pArray)`

Here `start` is the index of the first element of your array region, `len` is the length of the region, namely the number of elements therein, and `pArray` is a pointer to a region which *you* have allocated in native code for the array region of interest to be copied to or from. Notice that *you* are totally responsible for allocating `pArray` before manipulation with either or both of these methods and for freeing it afterwards. With an understanding of how the EPOC cleanup stack works under our belt, we are now in a position to do just this.

Below is a function that illustrates the use of these two functions. It cyclically permutes the first five elements of a Java `byte` array, passed in from a Java native method. As it can leave, either with `KerrNoMemory` should the allocation fail, or with `KerrArgument` should the array be null or have too few elements, the function would be called in practice from within a trap handler. Although the JNI functions called can throw an `ArrayIndexOutOfBoundsException`, we are confident our checks on the array length will forestall such problems.

```
jint ProcessArrayL(JNIEnv* aJNI, jbyteArray aByteArray)
     {
     if (aByteArray == NULL) User::Leave(KErrArgument);
     const jint KStart = 0;
     const jint KMinSize = 5;
     jint size = aJNI->GetArrayLength(aByteArray);
     if (size < KMinSize) User::Leave(KErrArgument);
     jbyte* const pArrayStart = (jbyte*) User::AllocLC
         (KMinSize * sizeof(jbyte));
     aJNI->GetByteArrayRegion(aByteArray, KStart, KMinSize,
         pArrayStart);
     jbyte* pArray = pArrayStart;
     jbyte temp = *pArray;
     TInt pArrayMax = pArrayStart + KMinSize - 1;
     for ( ; pArray < pArrayMax; pArray++)
          {
          *pArray = *(pArray+1);
          }
     *pArray = temp;
     aJNI->SetByteArrayRegion(aByteArray, KStart, KMinSize,
         pArrayStart);
     CleanupStack:: PopAndDestroy(); // pArrayStart
     return KErrNone;
     }
```

Here the EPOC `User::AllocLC(TInt)` function, returning a `TAny*`, is equivalent to the `malloc()` function more familiar to C users, except in that the EPOC version additionally puts a pointer on the cleanup stack. No corresponding call to `User::Free(TAny*)` is made because this is handled by the subsequent call to `CleanupStack::PopAndDestroy()`. We are obliged to use the rather ugly (and potentially unsafe) pointer arithmetic to access individual array elements, because EPOC does not support the [] -indexing approach for C-style arrays. For those object-oriented programmers among you who take exception to this, here is the same function implemented using EPOC's built-in array-handling mechanisms. This approach has obvious advantages if sophisticated manipulations are taking place, e.g. with insertions. There is a small additional overhead incurred in instantiating the EPOC `CArrayFixFlat` class (and in the time taken to read the Symbian Developer Library documentation about EPOC array-handling!).

```
jint ProcessArrayL(JNIEnv* aJNI, jbyteArray aByteArray)
      {
      if (aByteArray == NULL) User::Leave(KErrArgument);
      const jint KStart = 0;
      const jint KMinSize = 5;
      const int KMax = KMinSize - 1;
      jint size = aJNI->GetArrayLength(aByteArray);
      if (size < KMinSize) User::Leave(KErrArgument);
```

We instantiate in contiguous memory a byte array of granularity 5; that is, one that expands 5 bytes at a time when more capacity is needed.

```
CArrayFixFlat<jbyte>* array = new(ELeave) CarrayFixFlat<jbyte>
    (KMinSize);
CleanupStack::PushL(array);
```

Having placed our array safely on the cleanup stack, we use `ResizeL(...)` to allocate dummy elements which will hold the data from the Java byte array, then get a pointer to the start of the data area with a call to `Back(0)`.

```
array->ResizeL(KMinSize);
jbyte* const pArrayStart = (jbyte*) array->Back(0);
```

We can then copy the region of the Java byte array straight into the EPOC array. It is vital that the latter has been sized large enough or an EPOC panic will ensue. The data can then be manipulated with the many utility functions of `CArrayFixFlat`.

```
aJNI->GetByteArrayRegion(aByteArray, KStart, KMinSize,pArrayStart);
jbyte temp = array->At(0);
for (int i = 0; i < KMax; ++i)
      {
      array->At(i) = array->At(i+1);
      }
      array->At(KMax) = temp;
```

Next we copy the region of the Java array with the new data. There is nothing to stop you from writing back an expanded array region, provided only `KStart + KminSize` does not exceed the length of the original Java array. This done, the native buffer can be deleted.

```
aJNI->SetByteArrayRegion(aByteArray, KStart, KMinSize,pArrayStart);
CleanupStack::PopAndDestroy(); // array
return KErrNone;
}
```

13.6.4 Writable Static Data

The Problem

Another memory management issue which before long you are likely to encounter if you get seriously into JNI programming is writable static data, or

rather the lack of support for it in the Symbian OS. For example, if you are making many conversions to cp1252 format in successive calls to the native method discussed above, you would rightly want to consider caching the references to the file server session and the converter between calls, rather than re-creating them each time, then releasing these resources when you were done with them. You might expect to be able to do this by writing something like:

```
// Don't do this!
CCnvCharacterSetConverter* converter;
RFs fs;

jstring AppendStringL(JNIEnv* aJNI, jstring aString)
      {
      RJavaString ptr16(aJNI, aString);
      TBuf8<20> buf8;
      if (converter != NULL)
            {
            converter = CCnvCharacterSetConverter::NewLC();
            User::LeaveIfError(fs.Connect());
            converter->PrepareToConvertToOrFromL
                (KCharacterSetIdentifierCodePage1252, fs);
            }
      // etc.
```

Although this code will run happily on the PC-based emulator, it will not compile for ARM as the DLL structure employed by EPOC on the latter does not support writable static data: objections will be raised to the declaration of `fs` and `converter` outside of a class or function body.[12] A successful caching strategy in EPOC requires a little more work.

Towards a Solution

The necessary elements of a successful caching strategy are as follows.

1. First, instantiate everything you want to cache into heap memory, rather than on the stack, to avoid it being deleted when the native method returns.
2. Next, if you are caching more than one object, define a suitable C-style struct in which you can store *pointers* to the various objects you need.
3. Then, in your native method, create an instance of this pre-defined struct on the heap and initialize it with the pointers to the objects you want to cache.

We are not quite out of the woods yet. Essentially we have exchanged a difficult caching problem for an easier one. There remains the final task.

---

[12] The reason for this is that EPOC DLLs are designed to be run in place from ROM, in order to conserve RAM at run time. They have therefore to allocate any static (or "global") variables in a DLL internally when they are compiled. This is only possible if the variables are constant

4. Find a safe place to keep the pointer to the struct.

For this, two basic strategies are available to you:

- Cast the pointer to a `jint` and return it to the Java side to be held there.
- Cache the pointer in EPOC thread local storage (TLS).

These approaches are not without their limitations. Let's look at each approach in turn.

Java-Side Caching

An example of the Java-side caching approach is provided in the Infoprint3 project on the v6.0 Quartz SDK for Java. This differs from the Infoprint1 project we discussed above in that it provides three native methods instead of one.

```
public class InfoPrint
{
    static {System.loadLibrary("InfoPrint3");}

    private final static int KErrNone = 0;
    private static int err = 0;

    public static native int print(String aText, int aSessionId);
    public static native int connect();
    public static native void close(int aSessionId);

    public static void main(String[] argv) // test script
    {
        int sessionId = 0;
        try
        {
            // give the notifier server time to get started.
            Thread.sleep(15000);
            sessionId = connect();
            if(sessionId < 0)
            {
                System.err.println("Notifier session
                    failed. EPOC error: " + sessionId);
                System.exit(0);
            }
            err = print("Hello from Java ", sessionId);
            if(err != KErrNone)
            {
                System.err.println("EPOC error: " + err);
            }
            // give the infoprint some time to be displayed
            Thread.sleep(3000);
        }
```

```
            catch(InterruptedException e)
            {
                  System.err.println(e.getMessage());
            }
            close(sessionId);
            System.exit(0);
      }
}
```

Before calling the `print()` method, you call `connect()` to initiate a client session with the kernel-side EPOC notifier server. This returns a (positive) integer representation of the session handle, or a (negative) integer EPOC error code should a problem have occurred. The `connect()` method is implemented as follows:

```
const TInt KJavaHandleShift = 2;

jint ConnectL()
      {
      RNotifier* notifier = NULL;
      notifier = new(ELeave) RNotifier;
      CleanupStack::PushL(notifier);
      User::LeaveIfError(notifier->Connect());
      CleanupStack::Pop(); // notifier
      return REINTERPRET_CAST(TUint, notifier)>>KJavaHandleShift;
      }

JNIEXPORT jint JNICALL Java_InfoPrint_connect(JNIEnv* /*aJNI*/,
    jclass /*cls*/)
      {
      jint ret = 0;
      TInt error;
      TRAP(error, ret = ConnectL());
      return (error == KErrNone) ? ret : error;
      }
```

The implementation listed above differs slightly from that published with the Quartz v6.0 SDK for Java in that the latter simply cast the `RNotifier*` directly to a `TInt`. The problem with that approach is that memory addresses in EPOC are not 100% guaranteed to parse to positive integers. The approach taken here is one commonly used in Symbian Java system code, using the fact that allocations made on the user heap in EPOC are always aligned with a 4-byte boundary; in other words, the address is always a multiple of 4. Dividing by 4 (a 2-bit right shift) ensures that the parsed result will be positive. If the value returned is indeed positive, the handle is good: you can proceed to call the `print()` method, passing the handle as an additional parameter. Indeed you can call it as many times as you wish without there being any need to create and destroy notifier sessions (an expensive process requiring a stack context shift to kernel side). The `RNotifier*` is conveniently recovered with a 2-bit left-shift and a reinterpret cast:

```
JNIEXPORT jint JNICALL Java_InfoPrint_print
  (JNIEnv* aJNI, jclass /*cls*/, jstring aText, jint aSessionId)
      {
      RNotifier* notifier = REINTERPRET_CAST(RNotifier*, aSessionId<<
          KJavaHandleShift);
      const jchar* const textPtr=aJNI->GetStringChars(aText,NULL);
      if(textPtr == NULL) return KErrNoMemory;
      TPtrC16 text(textPtr, aJNI->GetStringLength(aText));
      notifier->InfoPrint(text);
      aJNI->ReleaseStringChars(aText, textPtr);
      return KErrNone;
      }
```

Finally, when you are done using the notifier client session for `infoprints`, you can call the `close(...)` method which frees up your client session resource (stored on the heap by the `connect()` method) *and* the associated kernel-side resources (created on your behalf):

```
JNIEXPORT void JNICALL Java_InfoPrint_close(JNIEnv* /*aJNI*/, jclass
    /*cls*/, jint aSessionId)
      {
      // recover the session handle
      RNotifier* notifier = REINTERPRET_CAST(RNotifier*, aSessionId<<
          KJavaHandleShift);
      // close the session if necessary and tidy up
      if(notifier != NULL) notifier->Close();
      delete notifier;
      }
```

Obviously a disadvantage of the Java-side caching approach is that it precludes the native method returning any useful data other than the pointer, or perhaps an error message. A second disadvantage is that it is fragile: you need to know what kind of object the pointer is actually pointing to so that you can cast it appropriately after recovering it at run time, and to be sure that the heap memory has not already been deleted (possibly by another thread).

A third, less obvious problem is that, since client sessions with the EPOC notifier server are thread-relative, there is a restriction that the `print(...)` and `close(...)` methods must be called on the same thread the `connect()` method was called on. This is difficult to police, especially if you are trying to create a robust library for third-party use, because of course you can't know in advance on which Java thread a native method, even with a public wrapper round it, will get called. For this reason, EPOC servers were extended in Version 6.0 to allow process-relative, as well as the more restrictive thread-relative, client sessions. But as the notifier server was not "upgraded" in this way, there remains an issue in this particular example.

A word of warning: think twice before you cache the `JNIEnv*` pointer. This may seem like a good idea since it avoids you having to keep passing it on to each subsidiary method that gets called from your native method. Indeed

it is essentially what we did in our `RJavaString` class above, by making it a member variable. But, in doing so we relied implicitly on the assumption that our `RJavaString` was created on the stack, so would be deleted *before* the native method returned. The problem here is that a `JNIEnv*` is only valid on a single thread. If, for example, we created a `RJavaString` on the heap in one native method and then tried to use it in another method on a different thread, it would fail.

A `JNIEnv*` pointer is only valid on the thread in which it was created.

EPOC Thread Local Storage

Thread local storage (TLS) is an EPOC device whereby each thread which attaches to a DLL is able to store a word of data—a generic 32-bit `TAny*` pointer which can be used to point to an arbitrary structure on the heap. This is done with the static `Dll:SetTls(TAny*)` method. You can then access that heap memory from within any function exported from that DLL through the `Dll::Tls()` method which returns a `TAny*` pointer, *provided* that function is called on the thread which cached the pointer.

The main advantage of this approach is that it allows a value to be returned by the native method to the Java side. (Should an exception arise it can be thrown from native code.) A disadvantage is that, even if the session handles which you are caching are not thread-relative, your native method calls making use of these handles will be. Another disadvantage which should be borne in mind is that the TLS mechanism is considerably slower than the alternative approach of passing the handle to and from Java as an integer and reinterpreting it.

The TLS approach is illustrated nicely in the Infoprint2 project on the v6.0 Quartz SDK for Java. The details of the Java class are almost identical to Infoprint3 which we have already looked at, except in that the session ID is not in this case passed back and forward across the Java Native Interface.

```
public class InfoPrint
{
      static {System.loadLibrary("InfoPrint");}

      private final static int KErrNone = 0;
      private static int err = 0;

      public static native int print(String aText);
      public static native int connect();
      public static native void close();
      ...
```

To see how the native methods are implemented, in particular the TLS mechanism, take a look at **InfoPrint.cpp** in the Infoprint2 project, and the associated

documentation in the Symbian Developer Library on the v6.0 Quartz SDK for Java. Further documentation on the TLS mechanism can be found in the v6.0 Quartz C++ SDK.

We can now return to the problem we originally posed of how to cache references to the file server session *and* to the converter between calls to `AppendStringL(...)`, rather than re-creating them each time. Adopting a TLS approach, this can be accomplished as follows.

First, we define a struct to hold pointers to all the session variables:

```
struct SConverterSession
      {
      CCnvCharacterSetConverter* iConverter;
      RFs* iFs;
      };
```

One of these structs can then be instantiated on the heap, cached in TLS and used as a holding place for the session handles between calls to `AppendStringL(...)`. Notice that the file server session handle must be declared on the heap as `RFs*` in this case, to prevent it being deleted when the method returns.[13]

```
jstring AppendStringL(JNIEnv* aJNI, jstring aString)
{
      RJavaString ptr16(aJNI, aString);
      TBuf8<20> buf8;
      SConverterSession* session;
      CCnvCharacterSetConverter* converter;
      RFs* fs;
```

If the session is already open, all that is needed is to recover a pointer to the session struct from TLS, then to access the converter from that:

```
if (Dll::Tls())
      {
      session = (SConverterSession*) Dll::Tls();
      converter = session->iConverter;
      }
```

If the session has not yet been opened, this must be done:

```
else
    {
    session = new (ELeave) SConverterSession;
```

[13] Although strictly speaking the file server session handle is not in this simple example needed again after the converter session has been initiated, it might be if we wanted later to use a character set other than cp1252: keeping the RFs handle available in this example serves to illustrate the general principle.

```
    CleanupStack::PushL(session);
    converter = CCnvCharacterSetConverter::NewLC();
    fs = new (ELeave) RFs;
    CleanupStack::PushL(fs);
    User::LeaveIfError(fs->Connect());
    CleanupClosePushL(*fs);
    converter->PrepareToConvertToOrFromL(KCharacterSetIdentifier
        CodePage1252, *fs);
```

Then we can assign the associated handles to the session struct and cache a pointer to the struct in TLS:

```
session->iConverter = converter;
session->iFs = fs;
Dll::SetTls(session);
CleanupStack::Pop(4); // *fs, fs, converter, session
}
```

Notice particularly the use of the cleanup stack in the above.

1. `fs` is pushed onto the cleanup stack twice, the first time with `PushL(...)` to ensure that the client session on the user heap is deleted, the second time with `CleanupClosePushL(...)` to ensure that the server-side resources associated with the client session are released—with the `Close()` method—should a leave occur.
2. The four items on the cleanup stack are popped but *not* destroyed: references to the associated resources are all now cached elsewhere and the cleanup stack is no longer responsible for their cleanup.

We can now do the conversions, leaving with impunity if any problems occur:

```
User::LeaveIfError(converter->ConvertFromUnicode(buf8, ptr16));
_LIT8(KWorld, "mon frère");
buf8.Append(KWorld);
TInt i = 0;
TBuf16<20> buf16;
User::LeaveIfError(converter->ConvertToUnicode(buf16, buf8, i));
return aJNI->NewString(buf16.Ptr(), buf16.Length());
}
```

Finally, we need to provide an additional method to perform the cleanup when the session is no longer needed. This would be invoked by a Java native method called from Java *on the same thread* as created the session.

```
void Close()
      {
      if(Dll::Tls())
            {
            SConverterSession* session = (SConverterSession*)
```

```
        Dll::Tls();
    session->iFs->Close();
    delete session->iFs;
    delete session->iConverter;
    delete session;
    Dll::SetTls(NULL);
    }
}
```

A more robust solution would ensure that the above methods were always called on the same thread. This is possible but is a non-trivial task beyond the scope of the present survey of JNI strategies.

13.6.5 Managing Java References

Global References

On the subject of caching variables, if the variable you want to cache from native code is a Java object or array (so can be referenced as a `jobject`), there is another issue you need to be aware of.

A `jobject` is by default a local reference. In other words, it is valid only for the duration of the native method in which it is defined or used.

In order to understand the implications of this, we need to take a look first at how local references are shared across the Java Native Interface. What in fact happens when you run a native method is that a local reference frame is set up *for that call* with a series of slots, each of which can contain a reference to a Java object, array, or class. These slots are accessed by handles, 32-bit addresses which are interpreted on the native side as instances of `jobject`. They are populated with references to objects and arrays in a number of ways:

- if references to any Java objects or arrays are passed as arguments to a native function, or if one is returned therefrom, these will each be assigned their own slot in the reference frame for that native method call.
- if a reference to a *pre-existing* object or array on the Java side is recovered via a JNI function, this reference will be assigned a new slot in the local reference frame.
- if any Java objects are *created* during the execution of your native method by means of a JNI function which has a `jobject` handle as its return value, these objects are likewise assigned a slot.

When the native call returns, the frame is popped and all the `jobject` handles become invalid. The objects in the first two of the above categories will remain accessible since they will by definition have local Java variables referencing them. Any object in the third category may, however, become inaccessible

and, as a consequence, a candidate for garbage collection, unless you create an explicit reference on the Java side by:

1. assigning the object to be a property of an object already referenced on the Java side, or
2. making the `jobject` your native method's return value.

An alternative strategy the Java Native Interface allows is the creation of a so-called *global* reference to a Java object with the JNI function `NewGlobalRef(jobject)`. What this does is to assign your Java object reference a slot in a global reference frame and return a handle thereto. This global reference frame does not, like local reference frames do, exist only for the duration of the current method call, but for the lifetime of the Runtime itself. Neither is it accessible only from the thread on which it was created, but can be accessed equally from any thread. This protects the Java object from garbage collection until you choose explicitly to release it. Suppose, for example, you have used the `NewString(...)` method to instantiate a `jstring` called `myJavaString`. You could cache this in thread local storage with the following code:

```
jobject mySavedObject=aJNI->NewGlobalRef(myJavaString);
Dll::SetTls(mySavedObject);
```

The first line protects the String from being garbage collected when the native method returns. The second keeps a reference to the string alive in native code. Notice that it is the `jobject` returned that you cache, *not* the original. You would then recover this `jstring` for later use in a subsequent native call as follows:

```
jstring myString=(jstring) Dll->Tls();
```

Bear in mind that, although the String is in principle accessible from native methods running on *any* thread now, this second call would in fact need to be made on the same thread as the original because only in that case would TLS return the value cached in the original call.

Finally, when you are done you can delete the global reference to the Java String using the JNI function `DeleteGlobalRef(jobject)` — recalling that `jstring` is a subclass of `jobject` — which allows the Java object to be garbage collected:

```
aJNI->DeleteGlobalRef(myString);
myString=NULL;
```

Note that both local and global references are equally instances of `jobject` so, until deleted, can be used interchangeably in Java native methods.

JNI functions that expect Java objects accept both global and local references. A native method may return a local or global reference to the VM as its result.

Managing Local References

The `NewGlobalRef (...)` and `DeleteGlobalRef(...)` functions are useful when you want to preserve a Java object *beyond* the duration of the Java native method call in which it was created. But what if you want the opposite effect, namely, to *avoid* having to preserve all the Java objects you create in a native method until that method returns, allowing the garbage collector to reclaim all those which have not been globally referenced? Then you can, and indeed probably should, take advantage of the `DeleteLocalRef(jobject)` function. Calling this method on an object that was created in the context of a Java native method signals to the garbage collector that the object is ready for collection. As the garbage collector is running on a different thread from your native method, it is quite possible that it will be scheduled to run during the execution of the latter. Due attention to details like this can have significant impact on the memory usage of your application. Note, however, that you cannot make objects passed to your native method as arguments available for garbage collection in this way as a separate reference to each such object is maintained on the Java side.

There is one other situation where use of the `DeleteLocalRef(jobject)` function can assist you with effective memory management. This is where a large number of local references to Java objects, classes, or arrays are created in the course of executing a native method. For example, you might set up a service thread in a native method that executes the following loop:

```
jstring str;
TInt err;
while (true)
      {
      str=GetNextName();
      TRAP(err, ProcessNextNameL(str));
      if (err !=KErrNone) return;
      }
```

Suppose the `GetNextName()` function is retrieving successive String objects through callbacks to the Java side, say from an input stream, to be processed by the `ProcessNextNameL(jstring)` function until the latter method leaves on account, say, of a `NULL` value being returned by `GetNextName()`. We are not actually creating any new Java objects here. Nor are we creating any new references or pointers on the native side, since each new `jstring` returned is assigned to the same variable `str`. Nonetheless, the amount of memory consumed in the execution of this loop can grow without bound. The reason is that, each time the function `GetNextName()` is executed and a new Java string referenced, a new slot is added to the local reference frame. The `jstring` variable `str`, which is here efficiently reused, merely holds a handle pointing to the appropriate slot in the reference frame; the slot itself holds the reference to the Java string. Reassigning the variable `str` to hold a new handle

does *not* result in the (32-bit) slot being made available for reuse (other than being able to provide another local reference subsequently to the object it already references). Rather the reference frame keeps a reference to each of the previously referenced Strings and continues to grow without bound. The only way to prevent this, other than having the native method return, is to use the `DeleteLocalRef(jobject)` function. So, modifying the above code to:

```
jstring str;
TInt err;
while (true)
      {
      str=GetNextName();
      TRAP(err, ProcessNextNameL(str));
      if (err !=KErrNone) return;
      aJNI->DeleteLocalRef(str)
      }
```

would result in the slot associated with `str` being made available for reuse after each successful call to `ProcessNextNameL(jstring)`. The potential explosion of RAM usage is prevented.

Another point worth bearing in mind is that although the JDK 1.1 JNI Specification does not require that duplicate references in the local reference frame to a single Java object be collapsed, Symbian's implementation does collapse such duplicates, having only one slot in a local reference frame for each distinct Java object referenced. So, if you are reassigning a native `jobject` variable to point to a new Java object but expect to reuse the original Java object before the native method returns, you need *not* in this case call `DeleteLocalRef(...)` on the variable before doing the reassigning.

13.7 Troubleshooting

We conclude our discussion of JNI with a collection of some typical questions or problems that might arise in getting started with JNI development for the Symbian OS.

13.7.1 Can I access the EPOC window server from JNI code?

It is not currently possible through JNI code to access the EPOC window server session handle used by the Symbian Runtime to service AWT requests. Thus you cannot make EIKON widgets like number or date editors available for display in Java windows. In other words, the only way to draw in a Java window or container is via elements derived from `java.awt.*`. However, it is possible to open your own window server session from JNI code and get a handle to the EPOC window server that way. Using this, events can be sent to particular window groups, e.g. bringing an arbitrary window to the foreground.

13.7.2 Can I use the JDK 1.2 extensions to the JNI specification?

A number of new JNI functions were added when the JNI Specification was upgraded for JDK 1.2. These include support for so-called weak global references, for finer-grained control over the marshaling of local references, and for the recovery of only part of a Java string for manipulation on the native side. Full details can be found at ***http://java.sun.com/products/jdk/1.2/docs/guide/jni/jni-12.html***. These enhancements are not available for use with the EPOC Release 5 Runtime but are all supported from Version 6.0 onwards.

13.7.3 "CL.EXE—Unable to locate DLL"

If you see this error message when you try to use `nmake`, you may not have set the environment for Microsoft Visual Studio™. To remedy this, invoke the batch file **vcvars32.bat** located in **VC98 Bin**.

13.7.4 "JAVAC.EXE—Application Error"

You may see this message when you first try with `nmake` to compile your Java classes in your JNI project. There is a known problem with the build system on v6.0 SDKs in that there is confusion between the `classpath` variable used therein and the Windows system variable of the same name. To work around this, you should before using `nmake` nullify the Windows `classpath` variable by typing "`set classpath=`" at the command prompt.

13.7.5 "ERROR LNK2001: Unresolved External Symbol"

If you have problems with linking against a method in an EPOC DLL, check that the DLL that method is exported from is included in an `EXTRA_LIBS` line in your **Makefile**. If there are problems linking to one of your native methods, check whether the method signature of your implementation is at variance with that declared in the header file generated by `javah`. In particular, take care that the second argument is declared in your implementation as a `jobject` for all instance methods and as a `jclass` for all static methods. If problems continue after checking these, try using `nmake clobber`.

13.7.6 java.lang.UnsatisfiedLinkError for Java native method at run time

Check the method was declared correctly in **exports.def**. If you add, or modify the name of, a native method in your Java class, you have to manually edit the **exports.def** file accordingly. Under v6.0 you have the alternative of deleting the file and having it regenerated automatically as part of the build process. Also

check that the latest build of your library is where it should be; in particular, that you have built the DLL for the version of the emulator (release or debug) you are running the code on. Note too that, although on real Symbian devices the app folder of a C++ application will be scanned for DLLs, libraries exporting Java native methods will only be found by Java applications if placed in the default location, namely the **System Libs** folder.

13.7.8 Unexplained Crash on the Emulator?

If the errors reported when your native code causes a crash are not sufficiently detailed for you to trace the origin, try stepping through your code with the Visual C++ debugging tools. You can do this as follows:

1. Open your **.cpp** file in Visual C++ and hit F5.
2. Select "Yes" to the offer of a default project and "No" to the offer of a **.exe** file.
3. Check the active configuration ("Build | Set Active Configuration") is set to "debug".
4. Hit Alt-F7 ("Project | Settings").
5. On the "Debug" tab under "General," set the executable (**pjava_g.exe**) and the program arguments appropriately
6. Under "Additional DLLs", add the DLL(s) you want to debug.
7. Add a breakpoint (F9), hit F5 again and reply "No" to the offer to rebuild.

If you subsequently modify your source code, don't forget to save it and rebuild it from a command prompt with `nmake` before attempting any further debugging.

To identify the source of a crash (EPOC Panic), cancel the "Find Source" dialog with which you will be presented when control switches to the Panic handling code, which you will have to debug in assembler. The Panic category will be available from a variable `aCategory` in the "Disassembly" window and the associated number from `aReason`. You can ascertain the significance of these by examining the Panic documentation in the C++ Edition of the Symbian Developer Library.

13.7.9 Spurious "KERN-EXEC 0" Panics on Debug Emulator?

It is recommended practice when writing Symbian OS code to surround methods which allocate and deallocate memory on the user heap with calls to the macros `__UHEAP_MARK` and `__UHEAP_MARKEND`. These cause the application, when run in debug mode, to panic (with `KERN-EXEC 0`) if any of the allocations made during the method's execution are not deallocated. This practice is not, however, recommended in the case of methods making use of JNI functions.

The problem is that there is a mismatch between the Symbian OS and the Java memory management strategies. JNI functions may make allocations on your behalf that depend for their deallocation on the intervention of the garbage collector. If this does not get run in time to satisfy your heap allocation checks, your Java native methods may generate spurious reports of memory leaks. The best strategy here is simply not to use these macros with code making calls to JNI functions.

13.7.10 Other Problems or Queries?

Information about other JNI issues can be found in Symbian's Java Knowledge Base on ***http://www.symbiandevnet.com***. If you can't find what you are looking for there, you can post a question on one of the Java discussion forums linked to from the Symbian DevNet website.

14

Looking to the Future

Jonathan Allin

Up to now we have concentrated mainly on Symbian OS v6.0, though we've also looked at EPOC Release 5. This chapter looks at the road map beyond Version 6.0: we'll start with Symbian's next major release, and then see how future Java releases will develop within the J2ME framework and integrate with technologies such as MExE and WAP.

14.1 The Next Release

The principal drivers behind the next major release of Symbian OS are mass market compatibility, enabled by MIDP support, and increased performance. This release will be based on the same Java implementation as Version 6.0, i.e. PJAE 1.1.

14.1.1 MIDP Compatibility

MIDP is a minimal Java environment that is being adopted by many operators and device manufacturers. Examples include RIM's Blackberry wireless email devices, J-Phone, Japan's second largest mobile phone operator, and other feature phone manufacturers.

Symbian is therefore implementing MIDP to achieve compatibility with these mass-market, low-cost devices. Initially, it will be implemented on top of Personal Java via a CLDC adaptation layer: the CLDC adaptation layer is needed because MIDP is designed for implementation on the CLDC (the CLDC and MIDP are described in **The Java 2 Micro Edition**).

14.1.2 Performance Improvements

We saw in Chapter 4 that Version 6.0 is perhaps 30% faster, like for like, than Java on the Psion Series 5mx. This is due primarily to an improved memory model, which makes it unnecessary to continually flush the code cache. This is relevant for split cache processors, such as ARM9 and StrongARM.

The next release will contain significant enhancements to Java's graphics and embedded performance.

AWT enhancements include:

- buffered graphics operations
- optimized graphics context management
- image format optimized for EPOC bitgdi
- caching of FontPeers
- replacement of heavyweight synchronization with lightweight thread locking in the graphics methods

This has resulted in significantly faster graphics operations (often doubling), with some primitive operations such as line drawing reaching near native performance.

Enhancements to the runtime include:

- further optimizations to the byte code interpreter loop. One benefit of this is faster method dispatch
- improved thread locking to increase the performance of synchronized method dispatch
- optimizations to the floating point support libraries
- some of the core API classes have been optimized (e.g. Hashtable, String, StringBuffer)
- memory allocation has been improved
- class lookup is much faster. This is important when loading classes as a test is always made first to see if a class is already loaded. Methods related to class look up, such as `class.forName()`, will also be quicker.

Last, and by no means least, the next version will support ARM's Jazelle hardware acceleration (details can be found at ARM's website: ***http://www.arm.com***). The ARM 9J core has three states, executing Java byte code alongside ARM (32-bit) and Thumb (16-bit) code. When in the Java state it is a Java processor, fetching and decoding Java byte codes and maintaining the Java operand stack. The top four words in the Java stack are kept in registers, as is the first local variable. Most byte codes (140 of them) are executed directly by the processor; the remaining byte codes (there are 96) are less rarely used and so are emulated in software. The byte code execution is

thus more tightly bound to the processor's core than would be the case with a coprocessor accelerator.

14.2 The Java 2 Micro Edition

Our aim in this section is to understand how Symbian's Java road map fits in with the J2ME, so we won't go into a detailed description of the J2ME; details can be found at ***http://www.java.sun.com/j2me/***.

In 1999 Sun realigned its existing and upcoming Java technologies into "editions." The purpose was to make it easier to develop, and develop for, identifiable markets and classes of devices. Three Java editions were defined:

- Java 2 platform, Standard Edition (J2SE). This is Java 2
- Java 2 platform, Enterprise Edition (J2EE). Aimed at developers who want to write distributed, multi-tier, transactional applications for the enterprise. Additions over J2SE include Servlets, Enterprise Java Beans, Java's naming and directory interface, transaction APIs
- Java 2 platform, Micro Edition (J2ME). This is targeted at consumer electronics and embedded devices.

J2ME includes wireless information devices, and so is of great interest to us.

14.2.1 Configurations and VMs

A Configuration is a horizontal specification that is intended to cover a broad range of devices with similar capabilities. It defines the requirements of the VM and a relevant set of core classes. J2ME has defined two Configurations, the CLDC and the CDC. These are subsets of the J2SE core APIs.

Sun has developed the KVM and CVM virtual machines specifically for the CLDC and the CDC respectively.

The CLDC

The CLDC is the Connected, Limited Device Configuration. It is a lean Configuration with reduced VM requirements that underlies profiles targeted at small devices such as feature phones, pagers, and small digital assistants.

To meet the required footprint a number of changes were required:

The standard IO, network, and storage classes are much too big for CLDC devices. These are therefore replaced with a set of classes known as the Generic Connection Framework. For instance, the following code fragment reads the first line from a web page and writes it to a file:

```
UniversalInputStream is = Connector.openInputStream("http://www.
   symbian.com");
```

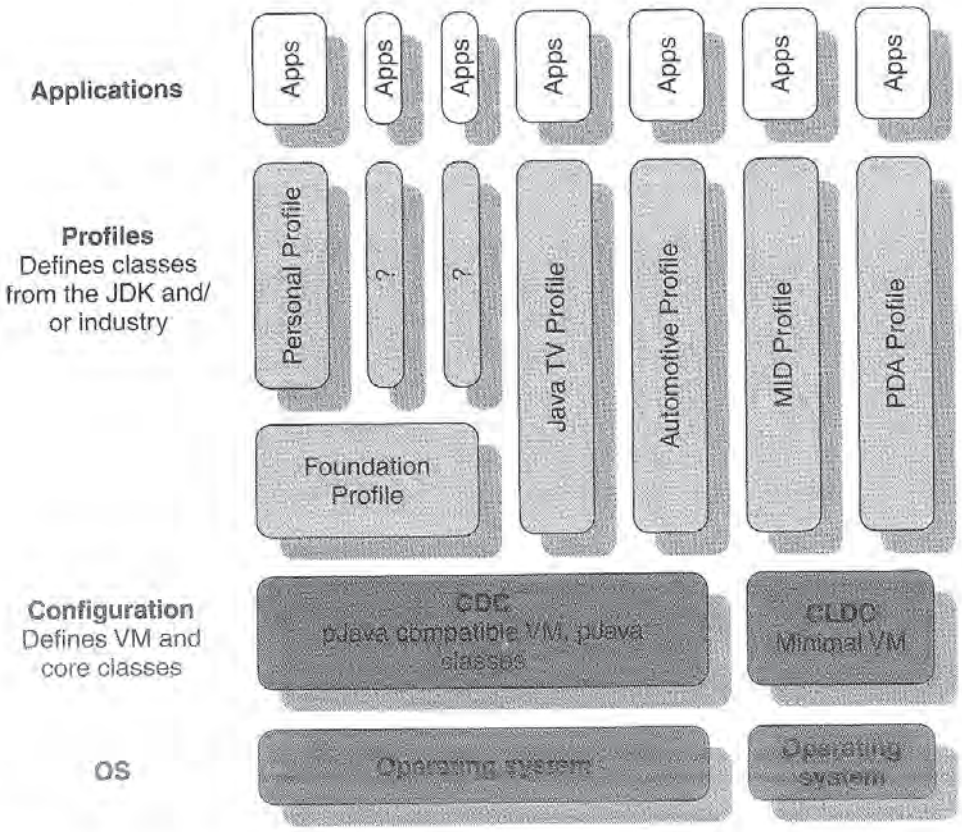


Figure 14.1 The J2ME architecture. J2ME is a layered architecture, with applications supporting horizontal "Configurations" which then support device- or industry-specific Profiles. Applications are written to a Profile.

```
UniversalOutputStream os = Connector.openOutputStream("file://
   webpage.dat", true);
os.println(is.scanLine());
is.close();
os.close();
```

The CLDC does not support:

- fine-grained security
- reflection
- Java Native Interface (JNI)
- user-defined class loader
- thread groups or daemon threads
- weak references
- finalization

Of these, the lack of fine-grained security is the most significant restriction to delivering rich Java services, though the loss of reflection and the ability

to define a class loader may prove problematic for some applications. Error handling is also more limited in the CLDC.

The CLDC provides a new, two-pass, implementation of the byte code verifier. Not only is the standard verifier bigger than the whole of the KVM, it also requires over 100 KB of dynamic memory for a typical application. The CLDC therefore uses a preverifier, typically run on a server, for the space- and performance-intensive processing. This then annotates the byte code so that all the device has to do is check the results for correctness: the annotations cannot be spoofed and code signing is not required. The on-device footprint is about 10 KB and requires less than a hundred bytes of run-time memory. The downside is a 5% increase in class file size.

The CDC

The CDC is the Connected Device Configuration. It is the larger Configuration, requiring a fully featured JVM. It underlies the profiles appropriate for more capable devices such as WIDs, set-top boxes, and car entertainment and navigation systems.

The CDC is a strict superset of the CLDC, but at the same time can be extended, through the addition of Profiles, to provide all the capabilities of the J2SE. It includes a number of features from the J2SE 1.3 APIs, the most important of which is the J2SE's fine-grained security model. It also provides the collections framework from the J2SE.

The KVM

The KVM fulfils the minimum requirements of a VM for the CLDC. It has been aggressively optimized and scaled back in order to fit on small devices like feature phones and pagers. Sun wrote it from scratch: it's not a port or cut-down version of an existing VM.

The KVM was designed to be easy to understand and to port. It's as simple as possible, consisting of just 24,000 lines of well-documented source code with the minimum of non-portable code. There is no MMU: neither the threading model nor the garbage collector requires one.

Multithreading is fully deterministic. The threading model uses green threads, where the thread context is switched after a certain number of byte codes have been executed. The garbage collector is non-compacting, which keeps it very simple, but also means that blocks of free memory are not merged. The consequence is that, as objects are created and deleted, the heap will become increasingly fragmented and eventually unusable. Applications and Profiles that use the KVM must therefore be carefully designed to create as few objects as possible and to reuse objects.

This simplicity comes at a price: the speed of the KVM is about half that of the JDK1.1 VM without a JIT (i.e. as used in Psion Series 5mx devices).

Verification is a code-intensive process so the KVM model allows verification to be split between server and client. The harder part of the verification process

can be carried out on the server, with the client just checking that a loaded class has been correctly verified (as an analogy, finding a route through a maze can be very difficult; however, checking that a route you have been given is indeed the correct route is a much easier process).

The KVM's memory requirements are modest (as befits a VM for small devices). Useful applications can be run in as little as 128 KB of memory, including the VM and classes.

Sun's source code for the KVM can be easily configured to include or exclude many of the CLDC's optional features, such as support for large data types (longs, doubles, floats) and multidimensional arrays. Limits can also be placed on the number of classes that can be loaded or the number of methods in a class. There is no JNI support: any native interfaces that are required must be linked in when the KVM is built.

Sun's KVM 1.0.2 will ship in the first or second quarter of 2001. It will be faster, provide better debugging support, and an exact garbage collector that can merge free space (though it still uses a simple "sliding block" algorithm optimized for a relatively small number of objects).

Symbian and the KVM

The KVM is excellent at what it does and it is fit for purpose, providing a Java capability for restricted devices. However, the KVM was not intended to scale to support large environments, and its security environment is essentially a "sandbox" model.

Symbian's goal is to create a Java environment which provides a richer, more open platform, capable of delivering fully functional applications and services. Such an environment can be built on the CDC, but is much harder to do on the CLDC and KVM.

The CVM

The "C" in CVM can stand for Consumer, Connected, or Compact. However, it doesn't stand for "Classic"! The Classic VM refers to the JVM provided with JDK1.1.

The CVM is the basis for Sun's CDC reference implementation. Like the KVM it was implemented from scratch, though it's intended for more capable consumer, embedded and real-time systems. These devices are still much smaller than desktop computers and so typically lack the memory needed for a full-blown JIT compiler or HotSpot technology. Whereas the KVM is about 24,000 lines of source code, the CVM is around 72,000.

The garbage collector in JDK1.1 was conservative: for instance, if an entry on the stack looked like it pointed to an object on the heap, it was assumed to be a reference to that object even though the match might be coincidental. This meant that an object could be retained even if no references to it remained. The CVM's garbage collector, on the other hand, is exact. Stack maps are used

to determine which items really do point to objects on the heap. It is also faster through using pointers rather than references. The GC algorithm is replaceable, for instance by one with real-time characteristics.

Other CVM benefits include faster, lower overhead synchronization, an optimized byte code interpreter loop, and ROM preloading. In other words, all the features that Symbian had to spend so much effort implementing in EPOC release 5 and Version 6.0!

Overall, the CVM will be faster than the 6.0 PersonalJava VM. The cost of implementing fine-grained security checks are outweighed by improvements to synchronization, the garbage collector, and the interpreter loop, together with the TypeID system which makes signature lookup much faster.

CVM 1.0 shipped in December 2000. CVM 1.1 will FCS (First Customer Ship) in December 2001. Benefits will include adaptive compilation and support for Java's profiling interface, the JVMPI.

14.2.2 Profiles

A Profile defines the APIs available to an application. These APIs are a subset of the standard Java classes, together with classes developed for a particular industry or class of device.

Sun specifies a process for developing Configurations and Profiles known as the Java Community Process or JCP (see ***http://java.sun.com/aboutJava/communityprocess/***). Profiles are defined by stakeholders in the industry, one of which takes on the role of specification lead. The process delivers both a specification for the Profile and a Technology Certification Kit (or TCK) which is used to certify all implementations of the Profile.

This section describes a number of Profiles of particular interest to Symbian.

The Personal Profile

The Personal Profile comprises a subset of the J2SE APIs and is a replacement for Personal Java. It builds on the CDC and so provides a complete application environment for more capable applications.

The Personal Profile APIs include PersonalAWT, networking, and Applets. JDBC and RMI support is optional.

The Foundation Profile

The Foundation Profile is a "headless" profile because it doesn't provide a UI. It is basically the Personal Java APIs but without AWT, RMI, or JDBC support. Its purpose is to provide support for the Personal Profile and other extension profiles such as the RMI Profile.

The MID Profile

The Mobile Information Device Profile builds on the CLDC. It is intended for feature phones, pagers, and small PDAs. The minimum display requirements

are 96 × 54 pixels, with 1-bit colour depth. It is suitable for games and simple services but, at least in the current (1.0) version, lacks the security features and the functionality to deliver significant wireless commerce applications. The MIDP provides support for user interface, life-cycle model, persistent data storage, networking, and timers for events.

The user interface is designed to support single-handed use of a keypad, and doesn't require a pointing device.

The display toolkit is the LCDUI, or Limited Connected Device User Interface. This provides both a high-level UI and a low-level graphics interface. The latter is intended for games, animations, and similar activities that require fast or detailed control of the screen. Applications that use the low-level APIs may not be portable. In contrast, the high-level UI provides an abstract interface. For instance, the developer may want to present the user with a choice of four items, but how this is presented is determined by the platform: it might be as a menu, as soft buttons, or as some sort of choice box.

MIDP applications are known as MIDlets and are typically discovered through a Web or WAP browser. Unlike Applets, however, they are stored on the device and launched like conventional native applications. MIDlets are delivered in a Jar file, which can contain more than one MIDlet. This group of MIDlets is called a MIDlet suite.

Persistent data storage is provided by the RMS, or Record Management Store. This is an abstraction of the underlying storage mechanism, which might be a conventional file system or a database-type store as found in Palm devices. MIDlets within the same MIDlet suite can share storage; however, in general MIDlets cannot share data.

Networking support is HTTP 1.1 only. This provides a familiar application model that makes use of the existing server infrastructure. The actual transport will be network specific, for example SMS or GPRS.

We learned earlier that the CLDC garbage collector is non-compacting. MIDP therefore takes great care to avoid creating unnecessary objects. For instance, the LCDUI avoids creating the thousands of short-lived objects typically generated by the AWT.

Further information on the MIDP, including the final specification, can be found at ***http://java.sun.com/products/midp/***. A good starting point for developers is ***http://developer.java.sun.com/developer/technicalArticles/wireless/#midp***, which provides a number of articles covering various aspects of the MIDP.

Symbian and the MID Profile

Symbian's MIDlet installer behaves like the native installer, but with extra dialogs to ask about upgrading the MIDlet, memory requirements, etc. Icons and the MIDlet's caption are extracted from the MIDlet suite's JAR file for use

by the installer, application launcher, and possibly the filing system. MIDlets can be uninstalled in the same way as native applications.

The installer will recognize MIDlet JAR files. So while, in general, MIDlets will be installed via a browser, it will also be possible to copy a MIDlet JAR file onto a device and install it directly (Chapter 11 discusses MIDlet provisioning and the Recommended Practice Document). MIDlets can either be removed one by one, or the complete MIDlet suite can be removed. In the former case the MIDlet will no longer be visible to the launcher, however the MIDlet suite's JAR file will not be removed until the last MIDlet in the suite is removed.

Each MIDlet in a MIDlet suite will be launched as a separate application, in exactly the same manner as a native application. For instance, each MIDlet will be displayed with its own icon.

Multiple MIDlets can be run concurrently. Those from the same MIDlet suite will run in the same VM: when the last MIDlet in a suite exits, the VM will be shut down and resources returned to the system.

Symbian's MIDP implementation uses native, rather than Java, components for the LCDUI. This makes the implementation both faster and smaller. We talked earlier about the CLDC's two-pass verifier. However, the next release of Symbian OS will use the single-pass verifier within the Personal Java VM.

The PDA Profile

The PDA profile will provide enhanced capabilities on CLDC devices, particularly Palm and Palm-type devices. It will provide additional features that are currently only available on CDC devices in a form suitable for CLDC devices, such as PIM access. It is likely to subset the AWT.

The profile is currently under specification.

. . . and Other Profiles

There are a number of Profiles currently under specification which are of particular interest to Symbian. The following list is taken from ***http://www.javasoft.com/aboutJava/communityprocess/search.html***:

000066	J2ME RMI Profile	The J2ME RMI Profile provides Java platform to Java platform remote method invocation for Java devices and interoperates with J2SETM RMI. Adds RMI back into the platform
000080	Java USB API	This specification provides a Java API for communicating with devices that are attached via the USB (Universal Serial Bus). It will allow Java applications to communicate with and manipulate USB devices attached to the bus

000082	Java APIs for Bluetooth	The specification will standardize a set of Java APIs to allow these Java-enabled devices to integrate into a Bluetooth environment

In addition, the SyncML organization is planning to provide Java APIs.

Profiles into Building Blocks

The original intention behind the Profile concept was that a developer would create applications for a particular Profile, without caring too much about the underlying platform. This is perhaps true for the MID and Personal profiles, which provide complete environments that include input and display capabilities, but is clearly not true for the Foundation profile or the JavaPhone APIs. The consequence is that Sun increasingly regards both Profiles and Configurations as building blocks, albeit with hierarchical dependencies so that, for instance, the Personal Profile builds on the Foundation Profile, which builds on the Connected Device Configuration.

14.3 Symbian's Road Map and the J2ME

Having taken a quick tour of the relevant J2ME technologies we need to see how these fit into Symbian's road map.

The next release of Symbian OS is about performance and MIDP compatibility. The following major version will be about exploiting J2ME technologies and standards, allowing Java developers to create rich, value-added services and applications, and enabling operators to deliver these services and applications.

This release may build on the CDC. It will thus include fine-grained security (as well as the J2SE collection classes) together with the Generic Connection Framework from the CLDC. The CDC will use the CVM or equivalent technology.

The Personal Profile will replace Personal Java. This builds on top of the Foundation Profile, rather than the CDC directly.

It's likely that the functionality of the JavaPhone APIs will be extended, providing enhanced messaging and communications capabilities. This would probably mean that JavaPhone would have to be developed as a Profile within the JCP.

The MID Profile and probably the PDA Profile will be supported to maintain compatibility with feature phones and other CLDC devices. However, these would run on top of the CDC implementation, rather than a separate CLDC implementation.

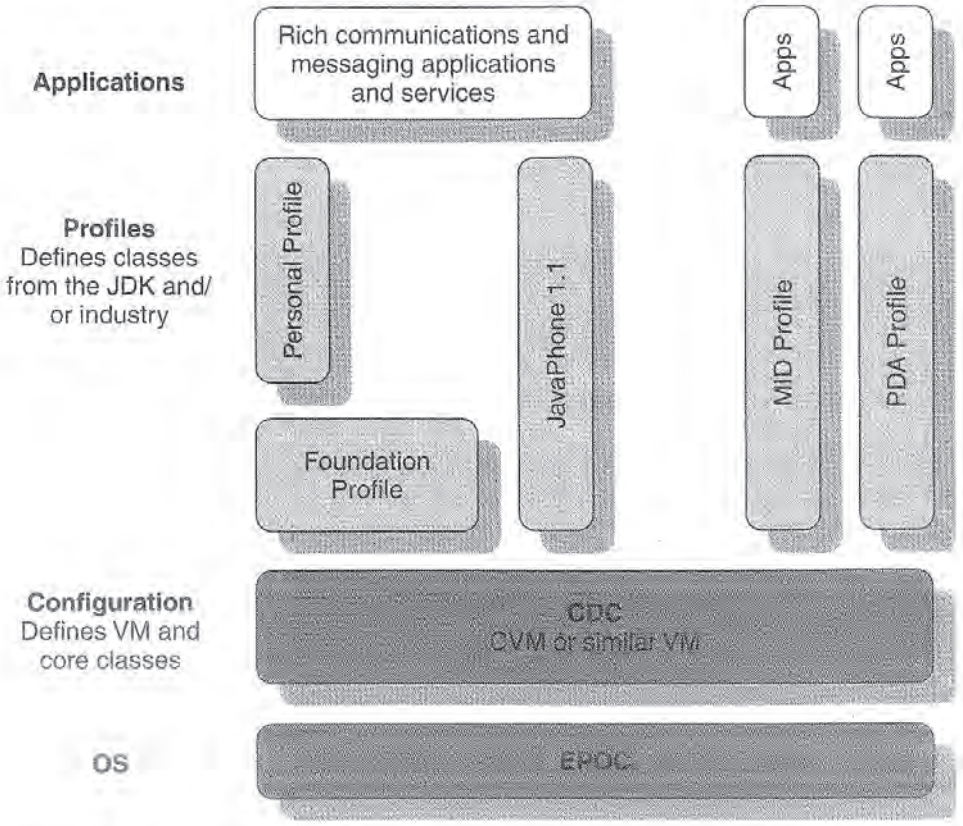


Figure 14.2 Symbian's next major release, and its fit within the J2ME architecture.

14.4 WAP

The Wireless Application Protocol is the de-facto world standard for the presentation and delivery of wireless information and telephony services on mobile phones. It is specifically designed for the low bandwidth, high latency characteristics of the wireless environment. WAP is based on existing standards where possible and develops new extensions where these are needed. In the future we can expect to see convergence with XHTML and HTTP standards.

WAP's ability to operate under low bandwidth, high latency conditions is enabled by a number of modifications to the HTTP1.1 protocol. For instance:

- HTTP's plain text headers are translated into binary code
- a lightweight session establishment protocol allows sessions to be easily suspended and resumed. This allows a session to be suspended to free network resources or save battery power
- WAP's Wireless Transaction Protocol provides similar services to TCP and means that a TCP stack isn't needed. However, it's lighter weight because, for example, it doesn't have to worry about packet ordering: there is only one route between a WAP proxy and the handset.

Further information can be obtained from the WAP forum: ***http://www.wapforum.org***.

The key features of the WAP specifications include:

- a markup language (WML) and a script language (WMLScript) designed to create applications on the small displays of handheld devices. WML does not assume that a QWERTY keyboard and a mouse are available for user input. Unlike the flat structure of HTML documents, WML documents are divided into a set of well-defined units of user interactions. One unit of interaction is called a card, and services are created by letting the user navigate back and forth between cards from one or several WML documents. WML has a smaller set of markup tags that makes it more appropriate to implement in handheld devices, than, say, HTML
- a lightweight protocol stack to minimize the required bandwidth and to guarantee that a maximum number of wireless network types can run WAP applications; for example, GSM SMS and USSD, circuit switched data (CSD), and GPRS. The WAP stack comprises: WAE, WSP, WTP, WTLS, and WDP. Each layer can be developed independently of the others, making it possible to switch to new bearers or to use new transport protocols without major changes to the other layers
- a framework for Wireless Telephony Applications (WTA) which allows access to telephony functionality such as call control, phone book and messaging from within WMLScript scripts. This allows operators to develop telephony applications integrated into WML or WMLScript services.

14.4.1 Security in WAP

WAP can provide confidentiality, integrity, and authentication. Authorization and non-repudiation require integration with a public key infrastructure (PKI).

WTLS is the wireless version of the industry standard Transport Layer Security (TLS), which is equivalent to the widely used Secure Sockets Layer (SSL) 3.1. TLS provides a secure network connection session between a client and a server, most commonly between a web browser and a web server.

The transformation of TLS to WTLS is based upon the need to support datagrams in a high latency, low bandwidth environment. To operate within this environment WTLS provides an optimized handshake to initiate a secure session through dynamic key refreshing. Dynamic key refreshing allows encryption keys to be updated on a regular and configurable basis during a secure session. This not only provides a higher level of security, but also provides considerable bandwidth savings on the relatively costly handshaking procedure.

Most WAP content is provided by a WAP gateway which converts between HTTP and WSP. At this point, information has to be decrypted and reencrypted because of the different encryption standards. This can lead to a variety of security attacks aimed at the gateway.

14.4.2 Java and WAP

Java and WAP meet in several areas:

- WAP is a set of protocols for delivering mobile content. It shouldn't be thought of as the Internet for mobile phones. Some of these protocols are available to Java for developers. For instance, we saw in Chapter 5-5.9, how WDP over SMS can be used to deliver Datagrams
- Java technologies such as servlets and Java Server pages can be used to create dynamic WAP applications
- Java classes can be accessed from WAP applications using URLs
- Java is appropriate for the more complex applications (see ***http://gartner3.gartnerweb.com/public/static/hotc/hc00092199.html***). For instance, a WAP form cannot be validated until it has been submitted, and it has to be submitted in its entirety. A Java application can validate each entered character, and can submit a partially completed form.

14.5 MExE

MExE is an acronym for Mobile Station Application Execution Environment. It is a 3GPP specification which defines an environment for secure application download onto mobile devices. The MExE specification can be found at ***http://www.3gpp.org/ftp/Specs/2000-12/Rel-4/23_series/***. Applications covered by the specification include WAP, Java applications and Applets, and MIDlets.

MExE defines several classmarks that address the needs and capabilities of different classes of device, capabilities and content negotiation, and a domain-based security environment.

All MExE devices must allow the user to look for, install, and remove applications. Devices are required to keep a journal of potentially chargeable network events, such as SMS messages, initiating a connection, or sending data over a connection. The length, format, and longevity of the journal is undefined and it's not accessible to MExE executables.

MExE uses HTTP 1.1 or protocols derived from HTTP1.1 such as WSP/B. The actual bearer might be SMS, IP, infrared, Bluetooth, etc.

14.5.1 MExE Classmarks

Currently MExE has defined four classmarks. MExE-compliant devices must support one or more of these classmarks:

1. *MExE classmark 1* is based on WAP (Wireless Application Protocol). Client devices require limited input and output facilities (e.g. as simple as a 3 lines × 15 characters display and a numeric keypad), and is designed to

provide quick and cheap information access even over narrow and slow data connections

2. *MExE classmark 2* is based on PersonalJava and JavaPhone. It provides and utilizes a run-time system requiring more processing, storage, display and network resources, but supports more powerful applications and more flexible user interfaces. The required JavaPhone APIs are those defined by the JavaPhone mobile phone profile (identified in Chapter 5)
3. *MExE classmark 3* is based on the Java 2ME CLDC and MIDP environment. It supports Java applications running on resource-constrained devices
4. *MExE classmark 4* will be aimed at mobile devices utilizing Microsoft technologies such as PocketPC.

Symbian devices are theoretically able to support classmarks 1 to 3. They are perhaps the only devices that can support classmark 2, which requires Personal Java and JavaPhone implementations.

Much of the MExE implementation is optional: for instance, a MexE-compliant device is not required to support the three security domains, but can regard all mobile code as untrusted.

14.5.2 Capability and Content Negotiation

The method for capability and content negotiation is based on the Composite Capability/ Preferences Profiles (CC/PP) specification. The properties and the actual schema are based on the WAP UAProf group specification.

At the very least the server must be able to establish the MExE classmarks and versions supported by the MExE device. Recommended capabilities include the device's manufacturer, model number, and software version; screen size and color depth, input capability (keyboard, keypad, pointer device), supported audio and video encoders (e.g. G.711, MPEG-2), and supported bearers (e.g. SMS, GPRS).

Negotiation can occur when a device first connects to a MExE server, or during content negotiation to determine, for example, whether a high quality color GIF or a monochrome low resolution image should be downloaded.

Classmark 3 devices (i.e. MIDP devices) can also use the Java Application Descriptor (JAD) file for capability negotiation. Content negotiation capabilities can use appropriate HTTP/1.1 or WSP request headers.

14.5.3 MExE Security Domains

MExE provides three trusted secure execution domains:

- operator
- manufacturer
- trusted third party

A MExE device must support all, or none, of these three security domains. The differences between the domains are fairly small: the operator has access to certain network properties and the manufacturer can install core system software.

Trusted applications can access the device's telephony capabilities, including initiating and receiving calls, and sending messages. They can create and read files, access the phonebook, and access system properties. They can install and remove digital certificates if they have the appropriate certified public key, can create and verify digital signatures, and can encrypt data.

Any application that cannot be identified with any of these three domains is regarded as untrusted. Untrusted applications essentially are restricted to the Applet or MIDP "sandboxes," though with a few extra privileges. They can, for example:

- initiate a phone call with user permission
- create a phone book entry with user permission
- generate DTMF tones within the current call.

14.6 Last Word

This book has examined many aspects of wireless Java on Symbian OS, though much of what we have learned is relevant to all constrained devices. The scope has ranged from detailed API discussions to understanding the opportunities for wireless services, and why Java is so important in realizing these opportunities.

Appendix A

Developer Resources and Bibliography

Download code for this book from ***http://www.symbian.com/books/***

Symbian websites

Corporate	***http://www.symbian.com***
Developer resources	***http://www.symbiandevnet.com***

Symbian licensee websites

Ericsson	***http://www.ericsson.com***
Kenwood	***http://www.kenwood.com***
Matsushita (Panasonic)	***http://www.panasonic.com***
Motorola	***http://www.motorola.com***
Nokia	***http://www.nokia.com***
Philips	***http://www.philips.com***
Psion	***http://www.psion.com***
Sanyo	***http://www.sanyo.com***
Siemens	***http://www.siemens.com***
Sony	***http://www.sony.com***

Symbian licensee/partner developer resources online

Nokia	***http://www.forumnokia.com/***
Motorola	***http://www0.motorola.com/developers***
Ericsson	***http://www.ericsson.com/developerszone***
NTT DoCoMo	***http://www.nttdocomo.com/imode***
Mobile Applications Initiative	***http://www.mobileapplicationsinitiative.com***

At-a-glance guide to resources used in this book

Chapter 1 Getting Started

Java Development Kit 1.1.8
http://java.sun.com/products/jdk/1.1/

EPOC Unique Identification Numbers
http://www.symbiandevnet.com/techlibrary/technotes/uidinfo.htm

Chapter 1 Java coding conventions

Sun's coding conventions
http://java.sun.com/docs/codeconc/html/codeconvTOC.doc.html

Netscape software coding standards guide for Java
http://developer.netscape.com/docs/technote/java/codestyle.html

Other recommended Java coding style guides
Geotechnical Software services
http://www.geosoft.no/javastyle.html
AmbySoft Inc coding standards for Java v17.01d
http://www.ambysoft.com/javacodingstandards.pdf

Chapter 5 JavaPhone

JavaPhone 1.0 API specifications
(a useful starting point for JavaPhone and some JavaPhone use cases)
http://java.sun.com/products/javaphone/

Javax.com specifications
http://java.sun.com/products/javacomm/

Javax.net.ssl specifications
http://java.sun.com/

Chapter 6 Wireless Java Games

The future of Wireless Gaming, September 2000, by Datamonitor,
http://www.datamonitor.com

Chapter 7 Building User Interfaces for WIDs

JForge GUI builder
http://www.tek-tools.com/

Adobe Acrobat viewer
http://www.adobe.com/products/acrviewer/main.html

Chapter 8 Writing optimized code for constrained devices

Java Obfuscators are described in JavaWorld
http://www.javaworld.com/javaworld/tools/jw-tools-allbyproduct.html

Chapter 9 Creating wireless Java services

[Buschman et al.] *A System of Wireless Patterns* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. Wiley, ISBN 0-471-95869-7

[Haggar] *Practical Java — Programming Language Guide*, by Peter Hagar. Addison-Wesley, ISBN 0-201-61646-7

[Larman et al.] *Java 2 Performance and Idiom Guide*, by Craig Larman and Rhett Guthrie, Prentice Hall, ISBN 0-13-014260-3

[Sosnoski] *Smart Object-Management Saves the Day*, by Dennis Sosnoski
http://www.javaworld.com/javaworld/jw-11-1999/ jw-11-performance.html

[Tasker et al.] *Professional Symbian Programming*, by Martin Tasker, Jonathan Allin, Jonathan Dixon, Tim Richardson, Mark Shackman, Wrox, ISBN 1-861003-03-X

Chapter 10 Java Security on Symbian OS

Non-repudiation summary
http://www.it.isy.liu.se/~asa/publications/nonrepudiation.html

PKI Public Key Infrastructure
http://dev.baltimore.com/pki/pki.html

Thumbprint online tutorial for JDK 1.2 security APIs
http://web2.java.sun.com/docs/books/tutorial/security1.1/index.html

Chapter 11 Provisioning Java Services

JSPM Developers
Art Technology Group ***http://www.atg.com***
Mobilitec ***http://www.mobilitec.com***

Java Web Start
Automates the deployment of Java applications to client devices
http://java.sun.com/features/2001/03/jws1.html

iBUS/Mobile
Implements an end-to-end message oriented middleware compatible with Sun's Java Message Service (JMS)
http://www.softwired-inc.com

Chapter 12 Case Studies

The Simple Conference Service

[Hughes et al.] *Java Network Programming, A Complete Guide to Networking, Streams and Distributed Computing*, Hughes, Shoffner, Hamner, Manning Publications 2nd edn, Jul 1999, ISBN 188477749-X

Draw the world, Create networked whiteboards with Java 1.1, Hughes, published on JavaWorld website Nov 1997, ***http://www.javaworld.com/javaworld/jw-11-1997/jw-11-step.html*** by ITWorld.com

Java Messaging Service, Gopalan Suresh Raj, first published on his website September 1999,
http://www.execpc.com/~gopalan/jms/jms.html

Java Message Service API, Sun Microsystems, first published on java.sun.com, ***http://java.sun.com/products/jms/***

Messaging Makes its Move, Todd Sundsted, first published on Java World website February 1999,
http://www.javaworld.com/jw-02-1999/jw-02-howto.html

Softwired's *iBus/Mobile JMS 1.0 (Java™ Message Service)*, software can be downloaded from ***http://www.softwired-inc.com/***

How to Use Model–View–Controller in Smalltalk, Borbeck, Applications Programming in SmallTalk-80™ 2nd edn 1992
http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

Chapter 13 JNI Programming on the Symbian OS

Standard Java extension packages
http://java.sun.com/products

Crystal v6.0 SDKs are provided by Nokia (the 9210 is the only product currently announced using Version 6.0 of the Symbian platform)
http://forum.nokia.com

Java Native Interface details and specifications
http://java.sun.com/j2se/1.3/docs/guide/jni/index.html

and

[Gordon] *Essential Java Native Interface*, Gordon, Prentice Hall ISBN 0136798950

[Liang] *The Java Native Interface: Programmer's Guide and Specification*, Sheng Liang, Addison-Wesley, ISBN 0201325772

Java functions
http://java.suncom/products/jdk/1.1/docs/guide/jni/spec/functions.doc.html

Java reference types
http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/types.doc.html#16432

JNI functions for JDK 1.2
http://java.sun.com/products/jdk/1.2/docs/guide/jni/jni-12.html

Chapter 14 Looking to the future

ARM
http://www.arm.com

Java 2 Micro Edition (J2ME)
http://www.java.sun.com/j2me/

Java Community Process
http://java.sun.com/aboutjava/communityprocess/

MIDP and specifications
http://java.sun.com/product/midp/
and
http://developer.java.sun.com/developer/technicalarticles/wireless/# midp

Profiles under specification
http://www.javasoft.com/aboutjava/communityprocess/search.html

Java and WAP
WAP Forum
http://www.wapforum.org

Java Apps over WAP
http://gartner3.gartnerweb.com/public/static/hotc/hc00092199.html

Java and MExE
http://www.3gpp.org/ftp/specs/2000-12/rel-4/23 series/

Appendix B

Symbian Licensee Devices

Since Symbian's formation in 1998, the following devices have been launched or announced by its licensees.

EPOC Release 5

Psion Series 5mx

The Series 5mx and 5mx Pro were both launched by Psion Computers in June 1999 and use EPOC Release 5. Symbian's Java VM is provided on CD with all Series 5mx devices.

Psion Series 5mx	
Manufacturer	Psion Computers
OS Version	EPOC Release 5
Processor	ARM710T 36 MHz
ROM	10 MB
RAM	16 MB
Screen	640 × 240 16 grey scale
Ports	2 serial (1 × RS2321 × IrDA)
Pointing Device	Pen
Expansion	Compact Flash
Power	2 × AA or 6V DC
Size	170 × 90 × 23 mm
Weight	354 g
Website	www.psion.com

Psion Series 5mx Pro	
Manufacturer	Psion Computers
OS Version	EPOC Release 5 *The operating system and custom application suites are loaded from CF card at system boot time*
Processor	ARM710T 36 MHz
ROM	0 MB
RAM	24 MB
Screen	640 × 240 16 grey scale
Ports	2 serial (1 × RS232, 1 × IrDA)
Pointing Device	Pen
Expansion	Compact Flash
Power	2 × AA or 6V DC
Size	170 × 90 × 23 mm
Weight	354 g
Website	www.psion.com

Ericsson MC218	
Manufacturer	Ericsson
OS Version	EPOC Release 5
Processor	ARM710T 36 MHz
ROM	12 MB
RAM	16 MB
Screen	640 × 240 16 grey scale
Ports	2 serial (1 × RS232 1 × IrDA)
Pointing Device	Pen
Expansion	Compact Flash
Power	2 × AA or 6V DC
Size	170 × 90 × 23 mm
Weight	354 g
Software	The MC218 includes Ericsson's WAP browser and phone applications
Website	www.ericsson.com

Psion netBook	
Manufacturer	Psion Enterprise Computing
OS Version	EPOC Release 5 *The operating system and custom application suites are loaded from CF card at system boot time*

Psion netBook	
Processor	StrongARM1100 190 MHz
ROM	0 MB
RAM	32/64 MB
Screen	640 × 480 colour
Ports	2 serial (1 × RS232, 1 × IrDA)
Pointing Device	Pen
Expansion	Compact Flash, Type I/II PCMCIA, IBM MicroDrive
Power	Removable Li-Ion rechargeable or 15.5 V DC
Size	235 × 182 × 37 mm
Weight	1150 g
Website	www.enterprise.psion.com

Psion Series 7	
Manufacturer	Psion Computers
EPOC Version	*EPOC Release 5* *The Java VM is provided on CD as with all Series 5mx devices*
Processor	StrongARM1100 133 MHz
ROM	16 MB
RAM	16 MB (upgradable to 32 MB)
Screen	640 × 480 colour
Ports	2 serial (1 × RS232 1 × IrDA)
Pointing Device	Pen
Expansion	Compact Flash, Type I/II PCMCIA and 2 × Internal memory slots
Power	Removable Li-Ion rechargeable or 15.5 V DC
Size	235 × 182 × 37 mm
Weight	1150 g
Website	www.psion.com

Psion Revo/Revo+	
Manufacturer	Psion Computers
OS Version	EPOC Release 5 *The original Revo's 8 MB RAM did not support Java. The Revo+, with 16 MB does*
Processor	ARM710T 36 MHz
ROM	8 MB
RAM	8/16 MB
Screen	480 × 160 16 grey scale

Psion Revo/Revo+	
Ports	2 serial (1 × RS232, 1 × IrDA)
Pointing Device	Pen
Expansion	None
Power	Inbuilt rechargeables or 6V DC
Size	157 × 79 × 18 mm
Weight	200 g
Website	www.psion.com

EPOC Release 5u

Ericsson R380s/R380 World	
Manufacturer	Ericsson
EPOC Version	EPOC Release 5u *The R380 is open only to content developers using WML. The "u" refers to a Unicode implementation of EPOC Release 5*
Operating Standard	R380s GSM900/1800 and R380 World GSM900/1900
Processor	ARM7TDMI 36 MHz
ROM	8 MB
RAM	8 MB (1.2 MB user data)
Screen	Flip closed: 120 × 120 pixels, Flip open: 120 × 360 pixels, grey scale
Ports	2 × Serial (1 × RS232, 1 × IrDA SIR)
Pointing Device	Pen
Expansion	None
Power	950 Li-Ion Standard BST-10 (Talktime 4–5 hrs, standby 104–107 hrs)
Size	159 × 51 × 26 mm
Weight	164 g
Website	www.ericsson.com

Symbian OS v6.0

Nokia 9210 Communicator	
Manufacturer	Nokia
OS Version	Symbian platform v6.0 *PersonalJava 3.01 (based on JDK1.1.6)*

Nokia 9210 Communicator	
Processor	32-bit ARM9-based RISC CPU
Mobile Standard	EGSM 900/1800 (Enhanced GSM), HSCSD at up to 43.2 kbps
ROM	14 MB
RAM	2 MB + 16 MB MultiMediaCard
SD-RAM	8 MB
Screen	Color screen (4096 colors) 640 × 200 pixels
Ports	IrDA, Ir-TranP, Cable (DLR-2L)
Pointing Device	Not needed (soft keys)
Expansion	Memory card slot (MultiMediaCard standard)
Power	Battery BLL3 1300 mAh Li-Ion, Talktime 4–10 hrs, standby 80–230 hrs
Size	158 × 56 × 27 mm
Weight	244 g
Website	www.nokia.com

Glossary

Term	Definition	See also
1G	In mobile telephony, first-generation systems were analog and circuit-switched. Voice links were poor, handoff unreliable, capacity low, and security non-existent. 1G systems are not now under active development — indeed, in some areas 1G spectrum is being auctioned for 2G and 3G use.	2G, 2.5G, 3G, AMPS, Analog, Circuit-switching, E-TACS, NMT, TACS
2G	Second-generation protocols use digital encoding and include GSM, D-AMPS (TDMA) and CDMA. 2G networks are in current use around the world. These protocols support high bit rate voice and limited data communications. They offer auxiliary services such as data, fax and SMS. Most 2G protocols offer different levels of encryption.	1G, 2.5G, 3G, CDMA, Circuit-switching, D-AMPS, Digital, GSM, TDMA
2.5G	2.5G protocols extend 2G systems to provide additional features such as packet-switched connection (GPRS) and enhanced data rates (HSCSD, EDGE).	1G, 2G, 3G, EDGE, GPRS, HSCSD
3G	Third-generation protocols support much higher data rates, measured in Mbits/second, intended for applications other than voice. 3G networks are expected to be	1G, 2G, 2.5G, IMT-2000, UMTS, W-CDMA, ***http://www.3gpp.org***

Term	Definition	See also
	starting in Japan by 2001, in Europe and parts of Asia/Pacific by 2002, and in the US later. 3G will support bandwidth-hungry applications such as full-motion video, video-conferencing and full Internet access.	
AMPS	Advanced Mobile Phone System: a 1G standard which operates in the 800-900 MHz frequency band. It is still widely used in the United States.	1G
Analog	The simple way to transmit speech, which is translated into electronic signals of different frequency and/or amplitude. The first networks for mobile phones, as well as broadcast transmissions, were analog. Due to being longer established in some countries, analog networks may offer better coverage than digital networks, however analog phones are less secure and suffer more from interference where the signal is weak. Analog systems include AMPS, NMT and ETACS.	1G, AMPS, Digital, E-TACS, NMT, TACS
API	Historically, "application programming interface". Practically, an API is any interface which enables one program to use facilities provided by another, whether by calling that program, or by being called by it. At a higher level still, an API is a set of functionality delivered by a programming system, and as such the mix of APIs in a particular system tells you what that system can do.	C++, Java, Platform
Bluetooth	An open specification for seamless wireless short-range communications of data and voice between both mobile and stationary devices. For instance, it specifies how mobile phones, WIDs,	***http://www.bluetooth.com***

Term	Definition	See also
	computers and PDAs interconnect with each other, with computers, and with office or home phones. The first generation of Bluetooth permits exchange of data up to a rate of 1 Mbps per second, even in areas with much electromagnetic disturbance. It transmits and receives via a short-range radio link using a globally available frequency band (2.4 GHz ISM band).	
bps	Bits per second: a way of quantifying data transmission throughput. It is the number of pieces of information (bits) transmitted or received per second.	
C++	An industry standard object-oriented compiled language, formally standardized in 1998, but tracing its history back to the early 1980s, with a heritage in C and BCPL going back to the late 1960s. C++ is the world's most widely understood system programming language. Symbian OS, is written in C++.	API, EPOC, Java, OPL
CDMA	Code Division Multiple Access: a digital wireless telephony transmission technique. CDMA allows multiple frequencies to be used simultaneously (Spread Spectrum). The CDMA idea was originally developed for military use over 30 years ago. The CDMA standards used for second-generation mobile telephony are the IS-95 standards championed by QUALCOMM.	2G, GSM, TDMA, W-CDMA
Cellular radio	The technology that has made large-scale mobile telephony possible. Current cellular networks reuse the same radio frequencies by assigning them to cells far enough apart to	

Term	Definition	See also
	reduce interference. A cell is the geographical area covered by one radio base station transmitting/ receiving in the center. The size of each cell is determined by the terrain, transmission power, and forecasted number of users. Service coverage of a given area is based on an interlocking network of cells, called a cell system.	
Circuit-switching	Means of creating a connection by setting up a dedicated end-to-end circuit which remains open for the duration of the communication.	Packet-switching
CDC	J2ME Connected Device Configuration: this configuration is intended to be used by devices requiring a complete implementation of the Java virtual machine (JVM), and an API set that may, via the addition of profiles, include the entire Java 2 Platform, Standard Edition API.	CLDC, J2ME, KVM
CLDC	J2ME Connected Limited Device Configuration: the CLDC serves the market consisting of personal, mobile, connected information devices. This configuration includes some new classes, not drawn from the J2SE APIs, designed specifically to fit the needs of small-footprint devices.	CDC, J2ME
Communicator	An information centric device with voice capability. Mobile by nature, Communicators include richer functionality than a Smartphone: the ability to link seamlessly with a PC, with printers, with the Internet — to send and receive files, to synchronize seamlessly and to be practical to carry around. In effect a fully featured	DFRD, Smartphone

Term	Definition	See also
	palmtop and digital phone in one unit. Symbian licenses EPOC, the Symbian OS, for Communicators.	
Configuration	A J2ME configuration defines a minimum platform for a horizontal class or family of devices, each with similar requirements on total memory budget and processing capabilities. A configuration defines the Java language and virtual machine features and minimum libraries that a device manufacturer or a content provider can expect to be available on all devices. Configurations are nestable, so that any software able to execute on a less capable configuration is able to execute on a more capable one. Two configurations are being defined, the CLDC and the CDC.	CLD, CLDC
Content Provider	A company that provides services to mobile phone users or network operators. These services could be shopping, web surfing, chatrooms, playing games, accessing data such as music and books through a server.	
Crystal	Code-name for Symbian's half-VGA screen size, keyboard-based, Communicator reference design.	Communicator, DFRD, Quartz
D-AMPS	Digital AMPS (Digital-Advanced Mobile Phone Service) is the digital wireless standard widely used throughout the Americas, Asia Pacific and other areas. D-AMPS uses digital TDMA on the one hand, and is required to be compatible with installed AMPS base station networks on the other. D-AMPS operates on the 800 and 1900 MHz bands.	2G, AMPS, Analog, Digital, TDMA

Term	Definition	See also
DCS 1800	Digital Communications System: another name for GSM working on a radio frequency of 1800 MHz. Also known as GSM1800 or PCN, this digital network operates in Europe and Asia Pacific.	2G, Digital, GSM
DFRD	Device Family Reference Designs (DFRDs) are Symbian's reference designs on which mobile computing and cellular phone manufacturers can base their devices. Each reference design is designed for optimal flexibility, giving consumer electronics manufacturers broad scope for differentiation and innovation in user interfaces, hardware designs and connectivity.	Communicators, Crystal, Quartz, Smartphones
Digital	A way of encoding information. On digital networks, data doesn't need to go though the extra step of being converted to analog signals, voice is sampled and coded in a way similar to how it is recorded on a CD. Digital networks are fast replacing analog ones as they offer improved sound quality, secure transmission and can handle data directly as well as voice. Digital networks include mobile systems GSM, D-AMPS, CDMA and TDMA.	2G, 2.5G, 3G, Analog
Dual band	Dual band mobile phones can work on networks that operate on different frequency bands. This is useful if you move between areas covered by different networks. Some networks operate on two bands, for instance GSM-1800 in town centers and GSM-900 in the rest of the country.	GSM
Dual mode	Dual mode mobile phones have more than one air interface and hence can work on more than one network. One	GSM, AMPS

Term	Definition	See also
	example is phones that operate on both digital and analog networks. They are quite useful if you want the advantages of a digital phone, but regularly visit areas where analog is the only service available.	
EDGE	Enhanced Data Rates for GSM Evolution: an enhanced modulation technique designed to increase network capacity and data rates in GSM networks. EDGE should provide data rates up to 384 Kbps. EDGE will let operators without a 3G license compete with 3G networks offering similar data services. EDGE is not expected before 2001 at the earliest.	2.5G, GPRS, GSM, HSCSD
EPOC	The old name for Symbian OS. EPOC Releases is found in devices like the Psion SMX and NetBook. Symbian OS technology provides an operating system, color support, fit-for-purpose application suites, Internet connectivity, software development kits and PC connectivity software, providing key commercial and technological advantages for manufacturers of wireless information devices.	WAP, Bluetooth, Java
E-TACS	Extended Total Access Communications System: a 1G mobile phone network developed in the UK and available in Europe and Asia.	1G, AMPS, TACS
GPRS	General Packet Radio Service: a radio technology for GSM networks that adds packet-switching protocols, shorter set-up time for ISP connections, and offers the possibility to charge by amount of data sent rather than	2.5G, EDGE, GSM, HSCSD, Packet-switching

Term	Definition	See also
	connect time. GPRS promises to support flexible data transmission rates typically up to 20 or 30 Kbps (with a theoretical maximum of 171.2 Kbps), as well as continuous connection to the network. A 2.5G enhancement to GSM, GPRS is the most significant step towards 3G, needing a similar business model, and service and network architectures. GPRS is expected to appear in some networks during 2000.	
GSM	Global System for Mobile communications, the most widely used digital mobile phone system and the de facto wireless telephone standard in Europe. Originally defined as a pan-European open standard for a digital cellular telephone network to support voice, data, text messaging and cross-border roaming. GSM is now one of the world's main 2G digital wireless standards, present in more than 140 countries. GSM is a time division multiplex (TDM) system. Implemented on 900, 1800 and 1900 MHz frequency bands.	2G, Digital, TDMA, ***http://www.gsmworld.com***
HSCSD	High Speed Circuit Switched Data: dedicated circuit-switched data communications technology for GSM which boosts data throughput up to 14.4 Kbps in a single channel, and by aggregating channels, up to 57.6 Kbps. An asymmetrical service can be offered where, for instance, one channel is allocated for the uplink and several are aggregated for the downlink. HSCSD can provide a fixed bit rate (transparent mode) or a variable one (non-transparent mode). In most cases HSCSD is available to network	2.5G, EDGE, GPRS, GSM

Term	Definition	See also
	operators as a pure software upgrade. HSCSD started to appear in some networks in 1999.	
iCalendar	Internet Calendaring and Scheduling Core Object Specification: a calendaring standard that specifies common event descriptions as Multi-part Internet Mail Extension (MIME) objects so that any iCalendar-compliant application can send or receive calendaring and scheduling information to or from any other iCalendar-compliant application. iCalendar is a revised/updated version of the vCalendar specification.	vCalendar, vCard, ***http://www.imc.org/rfc2445***
IMT-2000	International Mobile Telecommunications-2000: term used by the International Telecommunications Union (ITU) for the specification for projected third-generation wireless services. Formerly referred to as FPLMTS, Future Public Land-Mobile Telephone Systems.	3G
i-mode	Proprietary packet-based information service for mobile phones. i-mode delivers information (such as mobile banking, and train timetable) to mobile phones and enables exchange of email from handsets on the PDC-P network. Launched in 1999 by NTT DoCoMo, i-mode is very popular in Japan (especially for email and transfer of icons), but is not currently being used elsewhere.	PDC, WAP
IrDA	A suite of protocols for infrared (IR) exchange of data between two devices, up to 1 or 2 meters apart (20 to 30 cm for low-power devices). IrDA devices typically can have throughput of up to either 115.2 Kbps or 4 Mbps.	OBEX, ***http://www.irda.org***

Term	Definition	See also
	IrDA protocols are implemented by Symbian devices, in addition to Palm, Nokia communicators, printers from Hewlett-Packard and others. The Infrared Data Association, the industry body that specifies IrDA protocols, was originally founded by Hewlett-Packard and others.	
Java	Industry standard object-oriented language and virtual machine, invented by Sun Microsystems and formally released in 1996. Java is an ideal language for network applications and Applets. Sun's Java specifications include many Java APIs and platforms, including the JavaPhone API and PersonalJava platform, which Symbian is including in its own technology platform.	API, C++, EPOC, OPL
J2ME	Java 2 Platform, Micro Edition: the edition of the Java platform that is targeted at small, standalone or connectable consumer and embedded devices. The J2ME technology consists of a virtual machine and a set of APIs suitable for tailored runtime environments for these devices. The J2ME technology has two primary kinds of components — configurations and profiles.	Configuration, profile
JDK	Java Development Kit (JDK) is a software development environment for writing Applets and application in the Java programming language.	Java
JavaPhone	A Java API specification that provides access to agenda, contacts, power management, call control, and phone-book management. It is intended specifically for the prog-rammability requirements of Smartphones and Communicators.	API, Communicator, Java, Smartphone

Term	Definition	See also
JTAPI	The Java Telephony API is an extensible API that offers an interface to all call control services (from those needed in a consumer device up to those of call centers). JTAPI is part of the JavaPhone API.	Java, JavaPhone
Kernel	Core of an operating system, a kernel manages the machine's hardware resources (including the processor and the memory), and provides and controls the way any other software component can access these resources. The kernel runs with a higher privilege than other programs (so-called user-mode programs). The power and robustness of an OS's kernel play a major role in shaping overall system design and reliability.	EPOC
KVM	K Virtual Machine: a small-footprint, highly optimized VM that meets the CLDC's minimum requirements. It is targeted at small connected devices and scales from 30 KB to approximately 256 KB, depending on the target device's functionality. KVM implementations provide only a subset of the standard Java APIs. Areas such as I/O, graphics, networking, and security may not be supported and the number of class instances that can be loaded or the number of methods in a class may be limited. Floating point and 64-bit integer support is optional. Limitations on memory management require that care is taken when writing applications and make it difficult to port or run larger applications.	CDC, CLDC

Term	Definition	See also
MExE	Mobile Station Application Execution Environment (GSM 02.57): a framework to ensure a predictable environment for third-party applications in GSM or UMTS handsets (ie the Mobile Station). MExE does this by defining different technology requirements called "classmarks". MExE classmark 1 is based on WAP, and classmark 2 on PersonalJava and JavaPhone. Other classmarks may be defined in the future. MExE specifies additional requirements for all classmarks, for instance a security environment, capability and content negotiation, a user profile, user interface personalization, management of services and virtual home environment. A handset can support any number of classmarks.	GSM, Java, JavaPhone, PersonalJava, UMTS, WAP, ***http://www.etsi.org/smg/smg4/mexe.htm***
MIDP	Mobile Information Device Profile: set of Java APIs which is generally implemented on the Connected Limited Device Configuration (CLDC). It provides a basic J2ME application runtime environment targeted at mobile information devices, such as cellular phones and two-way pagers. The MIDP specification addresses issues such as user interface, persistent storage, networking, and application model.	CDC, CLDC, J2ME, API
Network operator	Company with a license to provide wireless telephony services.	Service provider
NMT	Nordic Mobile Telephone: one of the earliest 1G cellular network developed jointly in Denmark, Finland, Iceland, Norway and Sweden. Originally operated in the 450 MHz band. Later the 900 MHz band was used as well.	1G, Analog, Circuit-switching

Term	Definition	See also
OBEX	Object Exchange: a set of high-level protocols allowing objects such as vCard contact information and iCalendar schedule entries to be exchanged using either IrDA (IrOBEX) or Bluetooth. EPOC implements IrOBEX for exchange of vCards, for example with the Nokia 9110 or Palm III; and vCalendar, for instance with the Palm III.	Bluetooth, EPOC, iCalendar, IrDA, vCard
OS	Operating System: historically, the minimal set of software needed to manage a device's hardware capability and share it between application programs. Practically, "OS" is now used to mean all software including kernel, device drivers, comms, graphics, data management, GUI framework, system shell application, and utility applications. This would define Windows, PalmOS and MacOS as OSes. Symbian provides an OS and application suite tailored specifically for WIDs.	EPOC
OPL	A BASIC-like programming language, for rapid application development, used in the Symbian OS.	C++, EPOC, Java
Packet-switching	Technique whereby the information (voice or data) to be sent is broken up into packets, of at most a few Kbytes each, which are then routed by the network between different destinations based on addressing data within each packet. Use of network resources is optimized as the resources are needed only during the handling of each packet. This is an ideal model for ad hoc data communication, and works well also for voice, video and other streamed data. Devices using	Circuit-switching

Term	Definition	See also
	packet-switched communication appear to be "always connected" to the data network, whereas in the case of circuit-switched connections, setup time takes around 30 seconds to connect from a wireless information device through a GSM phone to an ISP (and it is harder to use). Use of a packet-switched network can be charged according to the volume of data transferred and not to any notion of time spent online.	
PCN	Personal Communications Network: another name for GSM 1800 (it is also known as DCS 1800). It is used in Europe and Asia Pacific.	2G, Digital, GSM
PCS	Personal Communications Service: an American generic term for a mass-market mobile phone service, emphasizing personal communication, independent of the technology used to provide it. PCS includes such digital cellular technologies as GSM 1900, CDMA and TDMA IS-136.	2G, CDMA, Digital, GSM, TDMA
PDC	Personal Digital Cellular: the 2G TDMA-based protocols used in Japan, owned by NTT DoCoMo. PDC services operate in the 800 and 1500 MHz bands.	TDMA
Personal Profile	Provides the J2ME environment for more capable devices. It provides the facilities currently defined by the PJAE specification.	PJAE, J2ME
PJAE	PersonalJava Application Environment (PJAE) is a Java platform optimized for the requirements and constraints of mobile devices. A Java runtime environment for networked applications running on personal consumer	Java, platform

Term	Definition	See also
	devices such as Smartphones rather than desktop computers.	
Platform	A set of technology which acts as a foundation for real-world applications, or higher-level platforms. Symbian's technology platform includes EPOC with its C++ APIs, a leading Java implementation, an applications suite and integration with wireless and other communications protocols.	API, C++, EPOC, Java
Profile	Specification that details the Java technology APIs, built on top of and utilizing the underlying configuration, necessary to provide a complete runtime environment for a specific kind of device or industry sector. Developers will write applications to the APIs defined by a Profile.	API, J2ME
Quartz	Code-name for Symbian's tablet-like, quarter-VGA portrait screen size, pen-based, Communicator reference design.	Communicator, Crystal, DFRD
Service provider	A company that provides, to phone users, services and subscriptions to mobile phone networks.	Network operator
SIM	Subscriber Identity Module: the SIM card is the smart card inserted inside all GSM phones. It identifies the user account to the network, handles authentication and provides data storage for basic user data and network information. It may also contain some applications that run on a compatible phone (SIM Application Toolkit).	GSM
Smartphone	A voice centric device with information capability. "Smartphones" is the generic name for digital phones on wireless networks that have the capability to browse the Internet, and	Communicator, DFRD

Term	Definition	See also
	receive and send SMS and email. Smartphones might even have scheduling and contact management software built-in. Symbian licenses Symbian OS for Smartphones.	
SMS	Short Message Service: available on digital GSM networks allowing text messages of up to 160 characters to be sent and received via the network operator's message center to your mobile phone, or from the Internet, using a so-called "SMS gateway" website. If the phone is powered off or out of range, messages are stored in the network and are delivered at the next opportunity.	
Symbian Connect	The PC-based Symbian Connect is a system for data synchronization, file management, printing via PC, application installation from PC, and other utility functions allowing Symbian OS phones to integrate effectively with PC and server-based data.	EPOC
Symbian Developer Network	The developers' support network for Symbian OS, where you will find all the resources to enable you to develop world class applications for Symbian devices.	***http://www.symbiandevnet.com/***
SyncML	Synchronization Markup Language, an industry-wide effort to create a single, common data synchronization protocol optimized for wireless networks. SyncML's goal is to have networked data that support synchronization with any mobile device, and mobile devices that support synchronization with any networked data. The SyncML structured data layer will use XML wherever appropriate. SyncML is	ICalendar, OBEX, vCard, WAP, ***http://www.syncml.org/***

Term	Definition	See also
	intended to work on transport protocols as diverse as HTTP, WSP (part of WAP) and OBEX, and with data formats ranging from personal data (e.g. vCard & iCalendar) to relational data and XML documents. The SyncML consortium was set up by IBM, Nokia and Psion among others and is supported by Symbian.	
TACS	Total Access Communication System: a British 1G analog mobile telephone standard based on the US AMPS system. It was later adopted in other countries including Hong-Kong and Japan.	1G, Analog, Circuit-switching
TDMA	1. Time Division Multiple Access: a digital wireless telephony transmission technique. TDMA allocates each user a different time slot on a given frequency. GSM, D-AMPS, PDC and DECT use TDMA in one form or another. 2. A name generally used for D-AMPS. TDMA networks are operated in the US, Latin America, New Zealand, parts of Russia and Asia Pacific.	D-AMPS, GSM, PDC, AMPS
UMTS	Universal Mobile Telecommunications Service: part of the IMT-2000 initiative. A 3G standard supporting a theoretical data throughput of up to 2 Mbps. It will be available commercially initially in Japan in 2001, and should be rolled out in most of the world by 2005.	3G, ***http://www.umts-forum.org***
Unicode	A 16-bit character encoding scheme allowing characters from Western European, Eastern European, Cyrillic, Greek, Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Urdu, Hindi and all other major world languages, living and dead, to be encoded in a	***http://www.unicode.org***

Term	Definition	See also
	single character set. The Unicode specification also includes standard compression schemes and a wide range of typesetting information required for worldwide locale support. Symbian's DFRDs are based on full Unicode implementations.	
vCalendar	Defines a transport- and platform-independent format for exchanging calendar and schedule information. For instance, users with devices running vCalendar-aware applications can schedule meetings automatically over an infrared link. Replaced by iCalendar.	iCalendar, vCard, ***http://www.imc.org/pdi/***
vCard	Standard defining the format of an electronic business card. All devices supporting vCard can exchange information such as phone numbers and addresses. For instance a user with a vCard-aware phonebook application on a handheld computer can easily transfer names and phone numbers to a vCard-aware mobile phone.	iCalendar, vCalendar, ***http://www.imc.org/pdi/***
VGA	Video Graphics Array: "standard" screen size of 640x480 pixels.	
W-CDMA	Wide-band CDMA: a CDMA protocol originated by NTT DoCoMo and now adopted for third-generation use by ETSI in Europe. W-CDMA supports very high-speed multimedia services such as full-motion video, Internet access and video conferencing.	3G, CDMA
WAE	Wireless Application Environment. Top layer of the WAP stack. Provides an environment for developing and executing portable applications and services. (see also HTML, JavaScript)	

Term	Definition	See also
WAP	Wireless Application Protocol: a set of communication protocol standards to make accessing online services from a mobile device simple. WAP can be used for emailing, browsing the Internet, access to newsgroups, and Internet Relay Chat (IRC) on mobile phones, handhelds or WIDs. WAP was conceived by four companies: Ericsson, Motorola, Nokia, and Unwired Planet (today called Phone.com). The WAP Forum is an industry association with over 200 members.	EPOC, ***http://www.wapforum.org***
WID	Wireless Information Devices represent the next revolution in mobile communication: voice and computing technology combined in a small, portable computer-telephone with wireless access to both voice and information services.	
WSP	Wireless Session Protocol. Provides methods to exchange content between client and server (see also HTTP)	
WTP	Wireless Transaction Protocol. Provides different methods for performing transactions, to varying degrees of reliability (cf. HTTP)	
WTLS	Wireless Transport Layer Security. An optional layer that, when present, provides authentication, privacy, and secure connections between applications (see also SSL/TLS)	
WDP	Wireless Datagram Protocol. Bottom layer of the WAP stack. Runs on a bearer such as SMS or GPRS and shelters the upper layers from the bearer services offered by the operator (cf TCP, UDP)	SMS, GPRS

Term	Definition	See also
WIM	Wireless Identity Module. A WAP standard for "cryptographic tokens," namely devices which handle secure storage, cryptographic operations, and access control	
WML	Wireless Markup Language (cf HTML)	
WMLScript	WML scripting languague (cf ECMAScript or JavaScript)	
WTA	Wireless Telephony Applications. Applications which interact with the device's telephony features	
WTAI	Wireless Telephony Applications Interface: These are the libraries that implement the telephony functionality. There are three types of libraries covering public WTAI functions, network-common WTAI functions, and network-specific WTAI functions. The only public WTAI functions are for making a call, sending DTMF tones, and for adding a phone book entry. Libraries can be accessed either through URIs of the form `wtai://<library>/<function>;<parameter>![<result>];[<result>]p` or from WMLScript.	

Index

Accountability 304
Action games, developing for WIDs 225
  Advantages of WIDs 225
  Bearer technology 226
  Constraints of WIDs 225
  Latency, dealing with 225
  Optimizing for mobile environments 227
  Rethinking action 226
ActionInputPreferred interface 102
Adaptation component 356
Adaptive compilers 288
Address Book API, JavaPhone 143
  ContactCard class 144
  ContactDatabase class 143
  ContactGroup class 144
  ContactTemplate class 144
AddressBookIntroModel class 36
Adobe Acrobat Viewer 235
Advanced client 3, 226
AgendaWriter 237
AIF Builder 30
  Application page 30
  Captions page 32
  DFRDs page 31
Arrays, optimizing use of 260, 415
Auditing 304
Authentication 303
Authorization 304
Automatic garbage collection 10
AWT, implementing 12
AWT, avoiding the 272

Benchmarking PDA devices 248

Caching 283
Calender API, JavaPhone 156
  CalendarEntry class 156
  CalendarToDo class 156
  CalenderDatabase class 156
  Symbian's Calendar Implementation 157
CalendarIntroModel class 36
CaffeineMark 248
CBAEvent class 50
CBAHandler class 112, 53
CBAListener class 50
CDC 437
Composite Capability/Preferences Profiles spec 446
ContactCard class 144
ContactDatabase class 143
ContactGroup class 144
ContactTemplate class 144
Certificates (digital) 306
Certification of Symbian JVM port 13
Chat service 340
CKONToolkit class 47
CL.EXE—Unable to locate DLL 429
Class path 252
CLDC 435
Cleanup Stack, using 409, 410
Client interaction scenarios 332
  Accessing banking services 332
  Selecting a game 332
ClientSession class 348
Clipping region, using 266
Closing a program 29
Communications API, JavaPhone 183
  CommPortIdentifier class 183
  CommPort class 184

Communication between packages 208
Confidentiality 304
Constrained devices, optimizing code for 247
Constrained devices, differences between PCs and 243
Contacts application 25
Containers, optimizing use of 259
com.symbian.devnet.quartz.awt package 81
com.symbian.devnet.crystal.awt package 53
Crystal (reference design) 4, 231, 241, 45
  Application control 46
  Command Button Area 46
  Screen capabilities 45
  Soft keys 46
  Status pane 46
  UI features 45
  Virtual cursor 47
  XBombs example 58
Crystal APIs 47
  CBAEvent class 50
  CBAListener class 50
  CKONToolkit class 47
  Command Button Area APIs 48
  EikCommandButtonGroup class 53
  EikStatusPane class 51
  EikVirtualCursor class 52
  Packages and class files 47
  Status pane APIs 51
  Toolkit APIs 47
  Virtual cursor APIs 52
Crystal utility classes 53
  CBAHandler class 53
  CFrame class 55
  CFrame and the CBA 56
  GridPanel class 54
  Setting the status area's width 57
  Sizing the application to the screen 55
CVM 438

Database package, JavaPhone 141
Datagram API, JavaPhone 170
DatagramIntroModel class 36
Debugging and Optimization 282
Deck class 216
Decryption 305
Deployment of services 358
DES 305
Design patterns 283
  Caching 283
  Caching results from a database 285
  Early or lazy instantiation 285
Developer resources and bibliography 449
Development tools 23
  Emulator 24
  Java tools 25
  SDK 24
DiceMachine 253
Digital signature 305

EikCommandButtonGroup class 53
EikStatusPane class 51
EikVirtualCursor class 52
Emulator 24
  Debug 27
  Hardware setup 35
  Unexplained crash on 430
Encryption 305
End-to-end security 304
EPOC 1, 467
EPOC Release 5 2, 13, 455
EPOC Release 5 devices, principal characteristics 230
EPOC Release 5u 458
EPOC thread local storage 422
EPOC window server, accessing from JNI code 429
ERROR LNK2001: Unresolved External Symbol 429
Exception handling 397
  Exceptions in EPOC 398
  Exceptions in Java 397
  Exceptions in JNI functions 400
  Handling exceptions on the native side 401
  Handling exceptions on the Java side 405
Extras application 25, 27, 33

Feedback, providing user with 251
File Exit command 29
File Manager application 27
Fine-grained security 308
Following message route 348

Game engine 211
  Deck class 216
  Game class 212
  Hand class 217
Global references, managing 425
GPS, use via a proxy 358
Graphics, optimization 264
  Clipping region, using 266
Image buffer, using 267
Painting implementations, comparison of 268
Popcorn program 265
Guidelines for developing applications 39
  Coding conventions, Java 41, 44
  Coding standards, Symbian 43
  Directory structures, separation 39
  Packages, use of 39

Hand class 217
Handheld Travel Assistant 351
  Abstract service proxy 362
  Application Architecture 356
  Combining services 359
  DemoService interface 359
  DemoServiceListener interface 360
  Location service 354
  MapOnWeb, use by map service 352
  Object exchange method 358
  Remote call 363
  Service proxy 360
  Servlet, use of 358
  Trafikanten, use by public transport service 354
  Travel scenario 351
Hashtable, optimizing use of 260
Hello program 25
  Building 26
  Running 27
HTTP, object exchange using 358
HTTPS, using to access a secure site 311

iBus//Mobile Messaging Service 336
Image buffer, using 267
In-lining 255
Instantiations, early or lazy 285
Installing programs on target devices 33
Integrity 303

JAR files, using 251
Java
  Benefits of 7
  Custom extensions 371
  Exception handling in 397
  Limitations of 369
  Standard extension packages 371
  WAP and 445
Java 2 Micro Edition (J2ME) 435
  CDC 437
  CLDC 435
  Configurations and VMs 435
  CVM 438
  KVM 437
  Profiles 439
Java Hotspot 289
Java Native Interface (JNI) 369, 262
Java performance, enhancing 290
Java references, management of 425
  Global references 425
  Local references 427
Java security 304
Java Services Provisioning Manager 330
  Client interaction scenarios 332
  JSPM developers 333
Java Web Start 334
java.lang.UnsatisfiedLinkError 429
JAVAC.EXE—Application Error 429
JavaPhone 12, 118, 34
JavaPhone Architecture 119
  APIs 119
  Packages 119
  Unsupported packages 121
JavaPhone on Symbian OS 121
Java-side caching 419
javax.comm 35, 183

javax.net.datagram 35, 170
javax.pim.addressbook 34, 143
javax.pim.calendar 34, 156
javax.pim.database 35, 141
javax.pim.userprofile 34, 169
javax.power.monitor 35, 178
javax.telephony.mobile 34, 129
javax.telephony 34, 123
JDK 1.1.8 25, 23
JDK 1.2, extensions to JNI specification 429
JForge 233
JIT compilers 288
JNI 369, 262
  Accessing fields of Java objects 384
  Build control files 377
  Build environment for JNI projects 375
  Build process 375
  Calling Java code from native code 370
  Calling native code from Java code 370
  Exceptions in EPOC 398
  Invoking methods of Java objects 386
  Further information source 431
  Java variables and methods, using 381
  JLong, use of 383
  Limitations on Symbian OS 373
  Mapping Java primitives to EPOC primitives 382
  on Symbian OS 372
  Primitive types, equivalence relations 382
  Prerequisites for development 374
  Prior knowledge required 375
  Required software 374
  Unicode data, working with 393
JNI functions, exceptions in 400, 403
JNI specification, JDK 1.2 extensions to 429
JPhoneIntro program 36
  Building and running 36
JPhoneIntroFrame class 36
JSPM 330
JSPM developers 333
JTAPI, Symbian's implementation 131
JTAPI APIs, JavaPhone 123
  JtapiPeer class 124
  JtapiPeerFactory class 124
  Listeners 127
  MetaEvents 127
  Observers 127
  Provider 125
JTAPI Mobile APIs, JavaPhone 129
JTAPIIntroModel class 36
Just in Time compilers 288

KahJongg 240
Key 305
Key management 307
Key pair 305
KeyboardInputPreferred interface 103
KVM 437
  SymbianOS and the KVM 438

Launcher screen 27
Local references, managing 427
Location-based services 298, 351
  Organizations 298
  RoutePlanner example 299

makesis 33
Memory management 286, 409
  Defensive coding 288
  Heap failure, invoking 287
  Memory leaks 286
  Mimicking small-memory machines in the emulator 286
Message, Simple Conference 344
Message digest 305
Messaging, Simple Conference 343
Method modifiers 255
MExE 336, 445
  Capability and content negotiation 446
MExE Classmarks 445
MExE Security domains 446
MID Profile recommended practice document 325
  Authorization 328
  Discovery 325

Installation 326
MIDlet attributes 329
Provisioning MIDlets in a WAP environment 328
Removal 328
Update 328
Mobilitec 333
Billing 333
Content aggregation 333
Middleware 334
Provisioning capability 333
Multi-player strategy games 191

Native code 388
Accessing static fields and methods 388
Arrays of primitives 392
Constructing New Java objects 389
Exceptions on the Native Side, handling 401
Java arrays in Native Code, using 391
Java objects in Native Code, using 383
Object arrays in Native Code 391
Throwing Java exceptions from Native Code 407
Native code, calling from Java code 370
Native code, calling Java code from 370
Native security 312
Certificate management module 312
Cryptography module 312
NoInputPreferred interface 103
Non-repudiation 304

Obfuscators 290
Object creation 254
Optimization 247, 13
Optimization, general guidelines 250
Optimization, how not to do it 261
Optimization, tools for 277
Analyzing CPU usage by caller 278
Analyzing times by callee 279
Analyzing times by line number 279
Debugging and Optimization 282
Heap analysis 281
Performance analysis tool 278
Profiling 277
Optimizing synchronized methods 280
OSGi 335
OSGi service framework 335
OSGi services gateway 335
Out-of-Memory, handling 288

Painting implementations, comparison of 268
Performance, enhancing 290
Personal Finance Manager 239
PersonalJava, history in Symbian OS 14
PersonalJava 99
Compiling with PersonalJava 116
Component input behaviour 102
Double buffering 102
Handling unsupported features 104
Symbian's implementation 99
Timer classes 104
Timer tester example 106
PersonalJava, component interfaces 102
ActionInputPreferred interface 102
KeyboardInputPreferred interface 103
NoInputPreferred interface 103
PositionalInputPreferred interface 103
PersonalJava, Timer classes 104
PTimer class 105
PTimerSpec class 104
PTimerWentOffEvent interface 106
PTimerWentOffListener interface 106
PersonalJava, Timer tester example 106
Architecture 107
CBA handler 112
Constructor 109
Experimenting with Timer tester 115
Miscellaneous methods 111
Setup 108
TimerHandler class 113
PhoneCall example 133
PhoneCall class 133
PhoneListener class 138
PKI (public key infrastructure) 307
Polling, avoiding 264
Popcorn program 265
POSIX 12
PositionalInputPreferred interface 103

Power Monitoring API, JavaPhone 178
PowerMonitor class 179
PowerMonitorListener class 179
PowerWarningType class 179
Profiles 439
Foundation Profile 439
MID Profile 439
Other Profiles 441
PDA Profile 441
Personal Profile 439
Profiles into building blocks 442
Symbian OS and the MID Profile 440
Provisioning Java services 323
Applets 324
MID Profile recommended practice, provisioning 325
Proxy 356, 357
PTimer class 105
PTimerSpec class 104
PTimerWentOffEvent interface 106
PTimerWentOffListener interface 106
Publish-Subscribe messaging 343
Publish–server side 346
Subscriber–client side 346

QCalendar 158
Architecture 160
CalendarModel class 167
Class diagram 162
Displaying CalendarEntry's repeat details 162
Modifying CalendarEntry's repeat details 164
QCalendar, CalendarModel class 167
close() 168
dateToString 168
findEntry() 168
open() 168
QCalendar, CentryDetail class 165
Methods and variables 165
Principal methods 166
setEntryRepeat() 167
setRepeatDetails() 166
QFrame class 81
QIkon 81
Quartz (reference design) 69, 5, 241
Quartz application, anatomy of 71
Application areas 73
Dialogs 77
Event handling 75
Hardware buttons 75
Menus 76
Messages 77
Screen layout 71
System areas 72
Text input mechanism 75
User interaction mechanisms 74
Views 73
Quartz user paradigm 69
Accessing data 71
Navigation between tasks 69
Quartz utilities 78
Beam 80
Find 78
Folders 78
Send as 80
Undo 79
Zoom 79
Quartz, limitations of Java programming on 94
Closing menus and choice lists 95
Quartz utilities 95
System clipboard, using the 95
Quartz, tuning Java applications for 80
Background/foreground event handling 89
Dialogs 91
EPOC window server, interacting with 86
Infoprints 84
Native utilities, access to 83
QFrame class 81
Saving application state 87
Shutting down applications 87
System-modal dialogs 84
TaskSwitch class 84
What you get for free 81
Queue, optimizing 269
QueueTest programme 270
Arithmetic operations 276
Arrays, increasing size of 276
AWT, avoiding 272
First results 270
Notify(), reducing calls to 274

References and further reading
JavaPhone further reading 189
Optimization references 291
Simple Conference Service references 350
RemoteFiler 238
Responsiveness of applications 251
Reusable classes, developing 405
Robustness 8
RoutePlanner 299

SDKs 24
C++ SDK 24
Documentation 24
Java SDK 24
JNI development 24
Secure environment, components 305
Certificates (digital) 306
Decryption 305
DES 305
Digital signature 305
Encryption 305
Fine-grained security 308
Key 305
Key management 305
Key pair 305
Message digest 305
PKI (public key infrastructure) 307
Use of PKI 308
Secure environment, developing 303
Authentication 303
Authorization 304
Confidentiality 304
End-to-end security 304
Integrity 303
Non-repudiation 304
Security 7
Security management, future 320, 446
ServerSession class 348
Services for multiple devices, developing 349
Bundles and bundling 350
Package names 349
Services, deployment of 358
Servlet 357
Servlet-proxy pairs 358
Sessions 348
Shell program 29
Simple Conference service 339
Bundling 350
Chat service 340
Connecting to Simple Conference server 339
Simple Conference, Crystal version 341
Following message route 348
Message class 344
Messaging 343
Model-View-Controller pattern 347
Publish–server side 346
Publish-Subscribe messaging 343
Simple Conference, Quartz version 343
Subscriber–client side 346
Whiteboard service 341, 299
Smart arrays 10
Stack, optimizing use of 260
Standard output stream, redirecting 33
Standardization 8
Strategy games, developing 191
Multi-player strategy games 191
Wireless Whist 191
Strings 257
As keys in hash tables 258
Comparing strings 257
Concatenating strings 257
Clean up 410
cp1252 data, handling 412
Symbian's JavaPhone Calendar implementation 157
Symbian's JTAPI implementation 131
Unsupported JTAPI methods 131
Symbian licensee devices 455, 229
EPOC Release 5 455
EPOC Release 5u 458
Symbian OS v6.0 458
Symbian OS 1, 4
Future of Java on 19, 433
History of Symbian OS1
History of Java on 12
Java security on 303
JavaPhone on 121
JNI limitations on 373
JNI on 372
Reasons for choosing Symbian OS 11

Symbian OS v6.0, security capabilities 309
  Extending 309
  Identity 309
  IdentityScope 310
  Java security 309
  Signer class 310
Symbian OS, next release of 433
  MIDP compatibility 433
  Performance improvements 434
Symbian reference designs 240
Symbian's PJAE implementation 99
  Optionality 99
  Performance enhancements 102
Symbian's road map and J2ME 442
Symbian's smartphone platform 242

Tailored news services 300
Target devices, characteristics of 246
TaskSwitch class 84
Threads 251
Thumbprint example 313
  Constructor 314
  Handling user input 317
  Measuring performance 320
  Thumbprint.openfile() 316
  Setup 313
  Thumbprint.updateSignature() 316
  Utilities 320
Ticketing, mobile 300
Ticketing, paperless 300
TokenRing engine 193
  DatagramCreator class 194, 193
  Receiver class 197, 193
  Ring class 200, 193
  TokenRing class 198, 202, 193
Traffic/travel congestion information 300
Troubleshooting 428
  CL.EXE — Unable to locate DLL 429
  ERROR LNK2001: Unresolved External Symbol 429
  java.lang.UnsatisfiedLinkError 429
  JAVAC.EXE — Application Error 429
User Interface package, Whist 220
User interfaces, creating 237
  AgendaWriter UI 237
  KahJongg UI 240
  Personal Finance Manager UI 239
  RemoteFiler UI 238
User interfaces, general guidelines 245
User Profile API, JavaPhone 169
  UserProfile class 169

Vectors, optimizing use of 260

WAP 443
  Java and 445
  Security in 444
Whist class 220
Whiteboarding 299, 341
WIDs, differences between PCs and 243
  Assumptions, don't make 243
  Missing features 244
WIDs, building user interfaces for 229
WIDs, running desktop applications on 231
  Installing and running Adobe Acrobat Viewer 235
  Installing and running JForge 233
  Larger Java applications 233
  Simpler applications and applets 231
  Write once, run anywhere 231
Wireless commerce 300, 15
Wireless games 17, 191
Wireless Java games 191
Wireless Java services 295
Wireless marketplace 15
  Cashless payment 300, 15
  Wireless commerce 300, 15
  Congestion, travel and traffic 300, 16
  Information, rail and flight 300, 354, 16
  Location-based services 298, 354, 16
  Wireless Services 297, 15
  Ticketing, mobile 300, 16
  Ticketing, paperless 300, 16
Wireless market, exploiting
  Business case 18

Device or platform 18
Write or port 18
Wireless services 297, 15
Network operator's perspective 295
Service provider's view 296
Case studies 301, 339, 351
Characteristics of good services 297
Chat service 340
Congestion information 300, 16
Location-based 298, 354, 16
Provisioning 297
Tailored news services 300
Ticketing, mobile 300, 16
Ticketing, paperless 300, 16
Whiteboarding 299, 341
Wireless commerce 300, 15
Wireless Whist 191
Communication between packages 208
Design issues 222
Future improvements 223
Game engine 211
Packages 193
Platform constraints 223
TokenRing engine 193
User Interface package 220
Writable static data 417
EPOC thread local storage 422
Java-side caching 419
problem of 417
towards a solution 418

XBombs: a Crystal example 58
ActionListener interface 60
Board class 60
Cursor class 60
CursorImage class 60
Game architecture 59
Graphics drawing 62
Handling keyboard input 66
Handling mouse input 66
Option class 60
Square class 60
Using the CBA 61
XBombs class 59

Wireless Java™ for Symbian Devices

By Jonathan Allin

"The combination of Java and Symbian OS enables you, the developer, to deploy services and applications in a multi-platform, mass-market, wireless environment. Using Java technology enables rapid development times, with easier coding and fewer bugs, in a secure and reliable environment."

From the foreword by Greg Papadopoulos, CTO, Sun Microsystems, Inc.

Visit the Wiley Communications Technology website on-line at:

www.wiley.co.uk/commstech

and Symbian Press at:

www.symbian.com/books

ISBN 0-471-48684-1



Java™ is the preferred language for developing applications and services on Symbian OS. It is a powerful programming environment that enables fast, secure deployment of applications and services onto a wide range of wireless devices. Symbian's Java implementation gives developers access to key wireless technologies, including telephony, contacts management, calendaring, Datagram messaging and power monitoring.

Complete with numerous illustrative and real-world examples, this 'from-the-source' guide provides developers with authoritative, practical and timely information on:

- using the Symbian Software Development Kits
- the JavaPhone and Personal Java APIs, and how they are implemented
- writing efficient and optimized code for constrained devices
- understanding and working with Java on Symbian reference designs
- creating games
- accessing native functionality through JNI (Java Native Interface)
- creating wireless services
- operator and service provider issues, including security, provisioning, rapid and reliable development

Jonathan Allin is Symbian's Java Technology Manager. He is one of a number of people in Symbian whose role is to ensure that Symbian provides a first class Java platform for wireless information devices. He is responsible for Symbian's Wireless Java technology training program and wrote the Java section in the Professional Symbian Programming book.

The book also contains contributions from Colin Turfus, Lucy Sweet, Alan Robinson and John Bown of Symbian and wireless case studies from Digia Oy (a Symbian Competence Center) and Telenor R&D.

Case study source code available for download from accompanying website

symbian PRESS

JOHN WILEY & SONS, LTD Chichester · New York · Weinheim · Brisbane · Singapore · Toronto