Advanced Programming in Java™

Rex Jaeschke

Preface

This text covers a number of more advanced Java topics most of which were introduced in JDK1.1. The material is not hardware or operating system-specific.

Reader Assumptions

To fully understand and exploit the material, you should be conversant with the following concepts and the syntax required to express them in Java:

- Basic Language Elements
- Looping and Testing
- Methods
- References, Strings, and Arrays
- Classes
- Inheritance
- Exception handling
- Input and Output
- Packages
- Interfaces

Source Code, Exercises, and Solutions

The programs shown in the text are provided on disk in a directory tree named source, where each chapter has its own subdirectory.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided on disk in a directory tree named labs, in which each chapter has its own subdirectory.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See lab file *xx*.java.)". This indicates the corresponding solution or test file in the labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Java Development Kit

The initial production release of Java was the Java Development Kit (JDK) version 1.0. Versions 1.1 through 1.5 contained numerous bug fixes, and language and library enhancements. Over the years, the numbering system changed, with 1.6 being known as Java 6. Then came editions 7, 8, 9, 10, and 11. The latest version can be downloaded from Oracle's website.

While the language has remained very stable, along the way, several features were added along with numerous new packages and classes and new methods to existing classes. Also, some existing method names have been

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

changed. In these latter cases, the old names continue to be acceptable, but are flagged by the compiler as *deprecated*, meaning that support for them might well be removed in future versions. If your compiler issues such a warning, consult the JDK on-line documentation to find the recommended replacement.

From an internationalization (I18N) viewpoint, one of the most significant additions made by V1.1 was the completion of support for dealing with non-US, non-English environments, including those involving very large alphabets and non-Latin writing systems.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my Java seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, March 2019

1. Threads

Java supports the ability to create multiple threads of execution within a single program. In this chapter, we'll see how threads are created and synchronized.[1] We'll also see how shared variables can be guarded against compromise during concurrent operations.

1.1 Introduction

A *thread* is an individual stream of execution as seen by the processor, and each thread has its own register and stack context. The run-time environment executes only one thread at a time. The execution of a thread is interrupted when it needs resources that are not available, it is waiting for an operation such as an I/O to complete, or if it uses up its processor time slice. When the processor changes from executing one thread to another, this is called *context switching*. By executing another thread when one thread becomes blocked, the system allows processor idle time to be reduced. This is called *multitasking*.

When a program is executed, the system is told where on disk to get instructions and static data. A set of virtual memory locations, collectively called an *address space* is allocated to that program, as are various system resources. This runtime context is called a *process*. However, before a process can do any work, it must have at least one thread. When each process is created, it is automatically given one thread, called the *primary thread*. However, this thread has no more capability than other threads created for that process; it just happened to the first thread created for that process. The number of threads in a process can vary at runtime, under program control. Any thread can create other threads; however, a creating thread does not in any sense own the threads it creates; all threads in a process belong to the process as a whole.

The work done by a process can be broken into subtasks with each being executed by a different thread. This is called *multithreading*. Each thread in a process shares the same address space and process resources. When the last remaining thread in a process terminates, the parent process terminates.

Why have more than one thread in a process? If a process has only one thread, it executes serially. When the thread is blocked, the system is idle if no other process has an active thread waiting. This may be unavoidable if the subtasks of the process must be performed serially; however, this is not the case with many processes. Consider a process that has multiple options. A user selects some option, which results in lots of computations using data in memory or a file and the generation of a report. By spawning off a new thread to perform this work, a process can continue accepting new requests for work without waiting for the previous option to complete. And by specifying thread priorities, a process can allow less-critical threads to run only when more-critical threads are blocked.

Once a thread has been dispatched, another thread can be used to service keyboard or mouse input. For example, the user might decide that a previous request is not the way to go after all, and wishes to abort the first thread. This can be done by selecting the appropriate option on a pull-down menu and having one thread stop the other.

Another example involves a print spooler. Its job is to keep a printer busy as much as possible and to service print requests from users. The users would be very unhappy if the spooler waits until a job had completed printing before it started accepting new requests. Of course, it could periodically stop printing to see if any new requests

[1] It is important to note that Java does not support synchronization of threads in different programs.

were pending (this is called *polling*), but that wastes time if there are no requests. And if the time interval between polls is too long, there is a delay in servicing requests. If it is too short, the thread spends too much time polling. Why not have the spooler have two threads: one to send work to the printer, the other to deal with requests from users. Each runs independent of the other, and when a thread runs out of work, it either terminates itself or goes into an efficient state of hibernation.

When dealing with concurrently executing threads, we must understand two important aspects: atomicity and reentrancy.

An *atomic* variable or object is one that can be accessed as a whole even in the presence of asynchronous operations that access the same variable or object. For example, if one thread is updating an atomic variable or object while another thread reads its contents, the logical integrity of those contents cannot be compromised—the read will get either the old or the new value, never part of each. Normally, the only things that can be accessed atomically are those having types supported atomically in hardware, such as bytes and words. All the primitive types in Java except `long` and `double` are guaranteed to be atomic. (These two might also be atomic for a given implementation, however, that's not guaranteed across implementations.) Clearly a Point object is not atomic; it has two parts, an x- and a y-coordinate, and a writer of a Point's value could be interrupted by a reader to that Point, resulting in the reader getting the new x and old y, or vice versa. Similarly, arrays cannot be accessed atomically. Since most objects cannot be accessed atomically, we must use some form of synchronization to ensure that only one thread at a time can operate on certain objects. For this reason, Java assigns each object, array, and class a synchronization lock.

A *reentrant* method is one that can be executed in parallel safely by multiple threads of execution. When a thread begins executing a method, all data allocated in that method comes either from the stack or from the heap. In any event, it's unique to that invocation. If another thread begins executing that same method while the first thread is still working there, each thread's data will be kept separate. However, if that method accesses variables or files that are shared between threads, it must use some form of synchronization.

1.2 Creating Threads

In the following example (see directory Th01a), the primary thread creates two other threads, and the three threads run in parallel without synchronization. No data is shared between the threads and the process terminates when the last thread terminates:

```
public class Th01a extends Thread
{
/*1*/   public Th01a(String threadName)
        {
/*2*/           super(threadName);
        }

/*3*/   public Th01a()
        {
        }
/*4*/   public void run()
        {
                int i, j;

                for (i = 0, j = 0; i <= 50000; ++i, ++j)
                {
                        if (i % 10000 == 0)
                        {
                                System.out.println(getName() + ": i = " +
                                        i + ", j = " + j);
                        }
                }
                System.out.println(getName() + " thread terminating");
        }

        public static void main(String[] args)
        {
/*5*/           Th01a t1 = new Th01a("t1");
/*6*/           Th01a t2 = new Th01a();

/*7*/           t1.start();
/*8*/           t2.start();
/*9*/           System.out.println("Primary thread terminating");
        }
}
```

Let's begin by looking at the first executable statement in the program, that in case 5. Here we create a new thread object of type `Th01a`, a user-defined class that extends the library class `Thread`. Apart from the static method `main`, that class has two constructors, one instance method, `run`, and no fields. We call the constructor taking a string argument and pass it our name for this thread. The spelling of this name is our choice and has no utility outside of our ability to set, retrieve, and display it. The constructor defined in case 1 is called and it simply passes this string onto its superclass constructor in case 2, where it is stored in some field hidden in the base object.

In case 6, we call the constructor taking no arguments, the one defined in case 3. Even though this constructor does nothing, we must define it; the compiler only creates a default constructor for a class that has no constructors explicitly defined. The default name given to the resulting thread created by the JDK execution environment has the form `Thread-`*n*, where *n* is an integer.[1]

At this stage, two thread objects have been constructed; however, no new threads have yet been created. At this stage, these threads are *inactive*. To make a thread *active*, we must call `start`, as shown in cases 7 and 8. This method starts a new thread executing by calling its `run` method. (Calling `start` on a thread that is already active results in an exception of type `IllegalThreadStateException`.) Since we have not defined a `start` method in `Th01a`, the one defined in its superclass, `Thread`, is used. The version of `run` defined in class `Thread` does nothing so we must override it in class `Th01a` giving it the signature as shown in case 4. This method is the main

[1] Some implementations start thread numbers at 0, others with 1.

method for the new thread, and since every thread of type Th01a starts execution with the same method, it and all the methods it calls, had better be re-entrant.

The run method loops, and every 10,000 iterations, it displays the values of its local variables. Since each thread that executes run gets its own set of these variables, the two threads don't adversely affect each other.

All three threads write to standard output and as we can see from the following example, the output from the threads is intertwined:

```
Primary thread terminating
t1: i = 0, j = 0
t1: i = 10000, j = 10000
Thread-0: i = 0, j = 0
Thread-0: i = 10000, j = 10000
Thread-0: i = 20000, j = 20000
t1: i = 20000, j = 20000
t1: i = 30000, j = 30000
t1: i = 40000, j = 40000
Thread-0: i = 30000, j = 30000
Thread-0: i = 40000, j = 40000
Thread-0: i = 50000, j = 50000
Thread-0 thread terminating
t1: i = 50000, j = 50000
t1 thread terminating
```

Of course, the output might be ordered differently on subsequent executions. Here's the output from another run:

```
Primary thread terminating
t1: i = 0, j = 0
t1: i = 10000, j = 10000
t1: i = 20000, j = 20000
Thread-0: i = 0, j = 0
Thread-0: i = 10000, j = 10000
Thread-0: i = 20000, j = 20000
t1: i = 30000, j = 30000
t1: i = 40000, j = 40000
t1: i = 50000, j = 50000
t1 thread terminating
Thread-0: i = 30000, j = 30000
Thread-0: i = 40000, j = 40000
Thread-0: i = 50000, j = 50000
Thread-0 thread terminating
```

In both sets of output, we see that the primary thread terminated before either of the other two threads started running. This demonstrates that although the primary thread was the parent of the other threads, the lifetimes of all three threads are unrelated.[1]

Although our version of run in this example is trivial, that method can call any other method to which it has access.

If we want different threads to start execution with different run methods, we must define them to have different classes. Although the following program (see directory Th01b, which is a version of Th01a) has only one run method, we can see how the thread support can be moved to a class separate from the application, thus allowing us to have multiple classes, each with their own run method:

```
class MyThread extends Thread
{
        public MyThread(String threadName)
        {
                super(threadName);
        }

        public MyThread()
        {
        }

        public void run()
        {
                int i, j;

                for (i = 0, j = 0; i <= 50000; ++i, ++j)
                {
                        if (i % 10000 == 0)
                        {
                                System.out.println(getName() + ": i = " +
                                        i + ", j = " + j);
                        }
                }
                System.out.println(getName() + " thread terminating");
        }
}
```

[1] We can make the life of a child thread be dependent on that of its parent by making the child a daemon thread, as we shall see later.

```
public class Th01b
{
        public static void main(String[] args)
        {
                MyThread t1 = new MyThread("t1");
                MyThread t2 = new MyThread();

                t1.start();
                t2.start();
                System.out.println("Primary thread terminating");
        }
}
```

1.3 Synchronized Methods

In the following example (see directory Th02), we have two threads accessing the same Point. One of them continually sets its x- and y-coordinates to some new value while the other retrieves these values and displays them:

```
public class Point
{
        private int x;
        private int y;

        public Point()
        {
                x = 0;
                y = 0;
        }

/*1*/   public synchronized void move(int xor, int yor)
        {
                x = xor;
                y = yor;
        }

/*2*/   public synchronized String toString()
        {
                return "(" + x + "," + y + ")";
        }
}

public class Th02 extends Thread
{
        private Point pnt;
        private boolean mover;
        public Th02(boolean isMover, Point p)
        {
                mover = isMover;
                pnt = p;
        }

        public void run()
        {
                if (mover)
                {
                        for (int i = 1; i <= 10000000; ++i)
                        {
/*3*/                           pnt.move(i, i);
                        }
                }
                else
                {
                        for (int i = 1; i <= 10000000; ++i)
                        {
                                if (i % 2000000 == 0)
                                {
/*4*/                                   System.out.println(pnt); // calls toString
                                }
                        }
                }
        }

        public static void main(String[] args)
        {
                Point p = new Point();

                Th02 t1 = new Th02(true, p);
                Th02 t2 = new Th02(false, p);

                t1.start();
                t2.start();
        }
}
```

Even though both threads start executing the same run method, by passing a value to their constructors, we can make each thread behave differently.

The potential for conflict arises out of the fact that one thread can be calling `move` in case 3 while the other is (implicitly) calling `toString` in case 4. Since both access the same Point, without synchronization, `move` might update the x-coordinate, but before it can update the corresponding y-coordinate, `toString` runs and displays a mismatched coordinate pair. When these methods are not synchronized, if we run the program often enough, eventually we'll likely see mismatched output; for example:

```
(2479893,2479893)
(4131691,4131690)       // mismatch
(5774640,5774640)
(7363071,7363071)
(10000000,10000000)

(2334280,2334280)
(3987228,3987228)
(5589142,5589142)
(7235212,8842648)       // mismatch
(10000000,10000000)
```

When the methods are synchronized by declaring them `synchronized`, as shown in cases 1 and 2, the coordinate pairs always match.

With synchronization, if `move` is called to operate on the same Point as `toString`, `move` is blocked until `toString` is completed, and vice versa. As a result, the methods spend time waiting on each other whereas without synchronization, they both run as fast as possible.

By declaring a method `synchronized`, we ensure it enables one such method from that class to operate on a given object of that class's type at any one time. Of course, an unsynchronized method in that class pays no mind to what any of its synchronized siblings are doing, so we must be careful to make all methods synchronized as appropriate. Synchronized methods that are operating on different objects do not wait on each other. The lock placed on all the instance methods when a synchronized method gets control stays in place until that method returns normally or as the result of an exception's being thrown. Therefore, the lock is in place while that method calls any and all other methods.

Constructors don't need to be synchronized; as their objects are just being created, no other thread is yet able to access them.

If a class method (rather than an instance method) is declared `synchronized`, that attribute applies to the lock on the class as a whole. Specifically, only one synchronized class method for a given class can execute at a time. Synchronization of class methods and instance methods is quite separate.

The `synchronized` attribute is not part of the signature of a method; that is, when overriding a synchronized method, we need not make the new method synchronized, and vice versa.

A synchronized method can call another synchronized method for the same object since it already has a lock on that object. In this case, the lock count is simply increased; it must decrease to zero before that object can be operated on by another synchronized method in another thread.

It is the programmer's responsibility to avoid a *deadlock*, that situation when thread A is waiting on thread B, and vice versa.

Exercise 1-1*: Modify Th02.java such that it contains four classes: `Point`, `MoveThread`, `PrintThread`, and a main application class, where `MoveThread` and `PrintThread` each have their own `run` methods. (See lab directory Lbth01.)

1.4 Synchronized Statements

Consider a method that contains 25 statements, only three consecutive ones of which really need synchronization. If we make the whole method synchronized, we'll be locking out resources longer than we really

2. Object Serialization

Most useful programs depend on information of a more permanent nature than that generated during a single execution. For example, programs that access an inventory typically query (and possibly update) one of more related data files. The lives of such "master files" transcend that of the execution of any of the application programs that use them. Other applications involve the communication of messages between separate programs, often referred to as *client* and *server*. While the life of a message is often much shorter than that of a database record, both cases involve the use of some data format external to the programs that manipulate them.

In this chapter, we'll see how Java objects and primitives can be converted into some external form suitable for use in file storage or for transmission during inter-program communication. The process of converting to some external form is known as *serialization* while that of converting back again is known as *deserialization*. Support for serialization was introduced with JDK1.1 and was expanded significantly in JDK1.2/Java 2.

2.1 Introduction

Consider the following example (see directory Sr01) which writes a number of objects and primitives to a disk file, closes that file, and then opens that file and reads them back into memory again:

```
import java.io.*;
import java.util.Date;

public class Sr01
{
        public static void main(String[] args)
        {

        // Serialize data to a file.

                int[] intArray = {10, 20, 30};
                float[][] floatArray = {
                        {1.2F, 2.4F},
                        {3.5F, 6.8F, 8.4F},
                        {9.7F}
                };
                String nullString = null;

                try
                {
/*1*/                   FileOutputStream fos = new FileOutputStream("Sr01.ser");
/*2*/                   ObjectOutputStream oos = new ObjectOutputStream(fos);

/*3*/                   oos.writeObject("Hello");
/*4*/                   oos.writeObject(new Date());
/*5a*/                  oos.writeObject(intArray);
/*5b*/                  oos.writeObject(floatArray);
/*6*/                  oos.writeObject(nullString);
/*7*/                  oos.writeBoolean(true);
/*8*/                  oos.writeInt(1000);
/*9*/                  oos.writeInt(2000);
/*10*/                 oos.writeFloat(1.23456789F);

/*11a*/                oos.close();
/*11b*/                fos.close();
                }
                catch (IOException e)
                {
                       System.out.println("Output stream error");
                       e.printStackTrace();
                       System.exit(1);
                }
```

In cases 1 and 2, we simply create a new file and connect an output stream to it; however, the stream is of a special kind, namely `ObjectOutputStream`. A stream of this kind writes primitive values and objects to an `OutputStream`. Later on, these primitives and objects can be read using an `ObjectInputStream`, as we shall see.

Class `ObjectOutputStream` implements the interface `OutputObject`, which, in turn, extends the interface `DataOutput`. The latter declares a family of methods for performing output of primitive types, while the former extends that interface to include objects. The methods they declare are as follows:

Table 2-1: Interface ObjectOutput's Methods

Name	Purpose
`close`	Closes the stream
`flush`	Flushes the stream
`write`	Writes out one or more bytes
`writeObject`	Write out an object

Table 2-2: Interface DataOutput's Methods

Name	Purpose
`write`	Writes out one or more bytes
`writeBoolean`	Writes out a boolean
`writeByte`	Writes out a byte
`writeBytes`	Writes out a `String`

Name	Purpose
`writeChar`	Writes out a `char`
`writeChars`	Writes out a `String`
`writeDouble`	Writes out a `double`
`writeFloat`	Writes out a `float`
`writeInt`	Writes out an `int`
`writeLong`	Writes out a `long`
`writeShort`	Writes out a `short`
`writeUTF`	Writes out a `String` in UTF format

`writeObject` can be used to output any object or array type, as can be seen in cases 3–6. The format of the output is of no importance to the programmer (although it is documented by Sun/Oracle[1]); all he cares about is that the reverse process will get those same values back. Primitive values are written out using their corresponding write methods as shown in cases 7–10. These values are written in binary just as they are represented in memory.

The calls to `close` in case 11 involve an implicit call to `flush`.

The class `ObjectInputStream` is used to read primitive values and objects from an `InputStream`. Class `ObjectInputStream` implements the interface `InputObject`, which, in turn, extends the interface `DataInput`. The latter declares a family of methods for performing input of primitive types, while the former extends that interface to include objects. The methods they declare are as follows:

Table 2-3: Interface ObjectInput's Methods

Name	Purpose
`available`	Number of bytes that can be read without blocking
`close`	Closes the stream
`read`	Reads in one or more bytes
`readObject`	Read out an object
`skip`	Skips one or more bytes

[1] Refer to the JDK documentation set for the web location of the serialization specification.

Table 2-4: Interface DataInput's Methods

Name	Purpose
`readBoolean`	Reads in a `boolean`
`readByte`	Reads in a `byte`
`readChar`	Reads in a `char`
`readDouble`	Reads in a `double`
`readFloat`	Reads in a `float`
`readFully`	Reads one or more bytes
`readInt`	Reads in an `int`
`readLong`	Reads in a `long`
`readShort`	Reads in a `short`
`readUnsignedByte`	Reads in a `byte` as an unsigned value
`readUnsignedShort`	Reads in a `short` as an unsigned value
`readUTF`	Reads in a `String` in UTF format
`skipBytes`	Skips a given number of bytes

Here then is the code that deserializes the data from the file created above:

```
        // Deserialize data from a file.

                try
                {
                        FileInputStream fis = new FileInputStream("Sr01.ser");
                        ObjectInputStream ois = new ObjectInputStream(fis);
```

The source continues, with each fragment being followed by the output it produced:

```
/*12*/                  String text = (String)ois.readObject();
                        System.out.println("String  >" + text + "<");
```

The output produced is:

```
String  >Hello<
```

`readObject` reads and returns the next object in the input stream, as a reference to type `Object`. Since we assign the returned value to a reference to `String`, an explicit cast is needed. If the type of the actual object read is not compatible with that to which it is being assigned, an exception of type `ClassCastException` is thrown.

It is worth noting that deserialization creates new objects; it cannot overwrite an object that already exists in a program. Note, however, that while `readObject` behaves somewhat like a constructor, no constructor is actually called.

```
/*13*/                  Object date = ois.readObject();
                        System.out.println("Date    >" + date + "<");
```

The output produced is:

```
Date    >Mon Feb 18 16:19:50 EST 2019<
```

In case 13, we do not need to cast to type `Date` explicitly, since the destination is of type `Object`. In any event, the correct version of `toString` will be called because `date`'s runtime type is really `Date`, not `Object`.

```
/*14a*/                 int[] newIntArray = (int[])ois.readObject();
                        for (int j = 0; j < newIntArray.length; ++j)
                        {
                                System.out.println("newIntArray[" + j + "] = "
                                        + newIntArray[j]);
                        }
```

The output produced is:

```
newIntArray[0] = 10
newIntArray[1] = 20
newIntArray[2] = 30
```

The process for deserializing an array is as we should expect.

```
/*14b*/                 float[][] newFloatArray = (float[][])ois.readObject();
                        for (int j = 0; j < newFloatArray.length; ++j)
                        {
                                System.out.print("newFloatArray["
                                        + j + "][0] -> [" + j + "]["
                                        + newFloatArray[j].length + "] = ");
                                for (int k = 0; k < newFloatArray[j].length; ++k)
                                {
                                        System.out.print(" " +
                                                newFloatArray[j][k]);
                                }
                                System.out.println();
                        }
```

The output produced is:

```
newFloatArray[0][0] -> [0][2] =  1.2 2.4
newFloatArray[1][0] -> [1][3] =  3.5 6.8 8.4
newFloatArray[2][0] -> [2][1] =  9.7
```

The code continues:

```
/*15*/                  System.out.println("Bytes available = "
                                + ois.available());
                        String str = (String)ois.readObject();
                        System.out.println("String  >" + str + "<");
```

The output produced is:

```
Bytes available = 0
String  >null<
```

`available` returns the number of bytes that can be read without blocking; however, zero is returned if the next thing in the input stream is an object. A return value of non-zero indicates that many bytes of contiguous primitive data exist in the input stream at this point.

```
/*16*/                  System.out.println("Bytes available = "
                                + ois.available());
                        boolean b = ois.readBoolean();
                        System.out.println("boolean >" + b + "<");
```

The output produced is:

```
Bytes available = 13
boolean >true<
```

Once all the objects have been read, 13 bytes of primitives remain, in the following order: 1 1-byte `boolean`, 2 4-byte `int`s, and 1 4-byte `float`. (Of course, objects and primitives can be interspersed as the programmer wishes; there is no need for primitives to be grouped together.)

```
                        System.out.println("Bytes available = "
                                + ois.available());
                        System.out.println("Skipping 4 bytes");
/*17*/                  ois.skipBytes(4);
```

The output produced is:

```
Bytes available = 12
Skipping 4 bytes
```

Once the 1-byte `boolean` has been read, only 12 bytes remain, and we skip 4 of those (that correspond exactly to the first `int`) by calling `skipBytes`. When skipping, we must be careful; for example, if we skip to a location that is not the start of a primitive or object value, we can easily get a bogus value returned. If we call a primitive-type read method and there are insufficient contiguous primitive bytes remaining, an `EOFException` is thrown. If we call `readObject` and the next byte belongs instead to a primitive value, an `OptionalDataException` is thrown. (If the format of control information in the stream violates certain internal consistency checks, a `StreamCorruptedException` exception is thrown.)

```
                        System.out.println("Bytes available = "
                                + ois.available());
/*18*/                  int i = ois.readInt();
                        System.out.println("int     >" + i + "<");
```

The output produced is:

```
Bytes available = 8
int     >2000<
```

The code continues:

```
                        System.out.println("Bytes available = "
                                + ois.available());
/*19*/                  float f = ois.readFloat();
                        System.out.println("float   >" + f + "<");
```

The output produced is:

```
Bytes available = 4
float   >1.2345679<
```

The code continues:

```
                        ois.close();
                        fis.close();
                }
                catch (IOException e)
                {
                        System.out.println("Input stream error");
                        e.printStackTrace();
                        System.exit(2);
                }
                catch (ClassNotFoundException e)
                {
                        e.printStackTrace();
                        System.exit(3);
                }
        }
}
```

Although this program performs both the serialization and deserialization of the same data set, these tasks are often done in separate programs. Therefore, when data is being deserialized, the class file for each class whose objects are being deserialized must be available at runtime. If it is not, a `ClassNotFoundException` is thrown.

2.2 Serializing Objects that Contain References

In the previous example, we wrote and read relatively simple object types. What about an object that contains numerous references to other objects; how is that handled? Consider a dictionary of more than 20,000 words, stored in a collection such that entries can be retrieved by key. JDK1.2/Java 2 provides such a class called `HashSet`, which is used in the following example (see directory Sr02a):

```
import java.io.*;
import java.util.*;

public class Sr02a
{
        public static void main(String[] args)
        {
/*1*/           HashSet set = new HashSet(21000);

                try
                {
                        InputStream is = new FileInputStream("dictionary.txt");
                        BufferedReader br =
                                new BufferedReader(new InputStreamReader(is));
                        String str;

                        while ((str = br.readLine()) != null)
                        {
/*2*/                           set.add(str);
                        }
                        br.close();
                        is.close();
                }
                catch (IOException e)
                {
                        System.out.println("Input stream error");
                        e.printStackTrace();
                        System.exit(1);
                }
```

In case 1, we pre-allocate the HashSet to handle 21,000 entries. (This simply speeds up the process, as it does not require reallocation during addition of a large number of entries.) Then in case 2, we read in a word, one per line, from a text file, and add that word to the HashSet.

Once all of the words have been read in and added to the HashSet, it can be written out as one big object with one simple call to `writeObject`, as shown in case 3 below:

```
                System.out.println("Dictionary contains " + set.size()
                        + " entries");

                try
                {
                        FileOutputStream fos =
                                new FileOutputStream("dictionary.ser");
                        ObjectOutputStream oos = new ObjectOutputStream(fos);

/*3*/                   oos.writeObject(set);
                        oos.close();
                        fos.close();
                }
                catch (IOException e)
                {
                        System.out.println("Output stream error");
                        e.printStackTrace();
                        System.exit(1);
                }
        }
}
```

In a separate program (see directory Sr02b), we wish to read in this dictionary and perform lookups on it for each word provided by the user, as follows:

```
import java.io.*;
import java.util.*;

public class Sr02b
{
        public static void main(String[] args)
        {
                HashSet set = null;

                try
                {
                        FileInputStream fis
                                = new FileInputStream("dictionary.ser");
                        ObjectInputStream ois = new ObjectInputStream(fis);
/*1*/                   set = (HashSet)ois.readObject();
                        System.out.println("Dictionary contains " + set.size()
                                + " entries");
                        ois.close();
                        fis.close();
                }
                catch (IOException e)
                {
                        System.out.println("Input stream error");
                        e.printStackTrace();
                        System.exit(1);
                }
                catch (ClassNotFoundException e)
                {
                        e.printStackTrace();
                        System.exit(2);
                }

                BufferedReader br = new BufferedReader(
                        new InputStreamReader(System.in));
                String word;

                try
                {
                        while (true)
                        {
                                System.out.print("Enter a word: ");
                                word = br.readLine();
                                if (word == null)
                                {
                                        break;
                                }
                                System.out.println(word
                                        + (set.contains(word) ? "" : " not")
                                        + " found");
                        }
                        br.close();
                }
                catch (IOException e)
                {
                        System.out.println("Input stream error");
                        e.printStackTrace();
                        System.exit(3);
                }
        }
}
```

Here's an example of some input and the corresponding output from this program:

```
Dictionary contains 20159 entries
Enter a word: house
house found
Enter a word: houses
houses not found
```

```
Enter a word: brick
brick found
Enter a word: manly
manly not found
```

The important lesson here is that we can serialize and deserialize an object of arbitrary size and complexity, in a single method call.

2.3 Handling Multiple References

It seems obvious that when we pass to `writeObject` a reference to an object, that a copy of the underlying object is written; however, is that what is really happening? What if we write out an object that contains multiple references to some other object, or we call `writeObject` twice, each time giving it a reference to the same object? Do we really want multiple copies of the same object to be written? It is quite likely that we don't.

What really happens then is that the serialization process keeps track of each distinct object written out, assigning it a unique handle. When the first reference to some object is passed to `writeObject`, the object is serialized and written to the output stream. Then when subsequent calls are made with a reference to that same object, only handle reference information is written out, allowing `readObject` to correctly return multiple references to the same object during deserialization. (This process can be circumvented by calling `ObjectOutputStream:reset` on the output stream; however, that is outside of the scope of this text.)

The following program (see directory Sr03) demonstrates this process:

```
import java.io.*;

/*1*/
class Employee implements Serializable { /* ... */ }

public class Sr03
{
        public static void main(String[] args)
                throws IOException, ClassNotFoundException
        {

        // Serialize data to a file.

                Employee emp1a = new Employee();
                Employee emp2a = new Employee();
                Employee emp3a = emp2a;

/*2a*/          System.out.println("emp1a == emp2a is " + (emp1a == emp2a));
/*2b*/          System.out.println("emp2a == emp3a is " + (emp2a == emp3a));
/*2c*/          System.out.println("emp1a == emp3a is " + (emp1a == emp3a));
```

3. Sockets

It has become increasingly common for large applications to be broken down into a set of cooperating programs that communicate with each other using some sort of communications protocol. These programs might run on different machines, under different operating systems, and be written in different languages. They might also be running on the same machine. In fact, these programs might even really be different threads in the same program.

In this chapter, we will learn about sockets, the means by which Java programs can communicate with each other.[1] For the most part, we will be discussing communication between programs running on the same machine. While communication across a network is indeed an important topic, this chapter is about inter-process communication, not networking.

3.1 Introduction

Consider an application involving a database query facility. One program, called the *server*, waits for requests from another program, called a *client*. When a request is received, the server executes the query and returns the results (or perhaps an error message) to the client. In many cases, there will be numerous clients, all sending queries to the same server at the same time, requiring the server to be a more sophisticated program than are the clients.

While in some environments the server runs on a machine dedicated to that task, a server could simply be a program running alongside numerous others, some of which are servers and/or clients in other applications. In fact, if our database server needs access to files not resident on its own system, it may well be a client of a file server on some other system. In fact, a single program might have a server thread and one or more client threads. Therefore, we must be careful when using the terms "client" and "server". While they might convey a familiar meaning in the abstract sense, they might be used quite differently in different physical implementations.

From a generic viewpoint then, a client is a consumer of services provided by a server, and a server can be a client of some other service.

3.2 Server Sockets

Let's start by looking at a simple yet representative server program (see Sk01Server.java in directory Sk01). This program waits for a client to send it a pair of integers. The server then adds these values together and sends the result back to the client:

[1] You might also be able to communicate with programs written in other languages too; however, that is outside the scope of this text.

```
import java.io.*;
import java.net.*;

class Sk01Server
{
        public static void main(String[] args)
        {
                int port;

                // port gets initialized somehow

                try
                {
/*1*/                   ServerSocket ser = new ServerSocket(port);
/*2*/                   Socket soc = ser.accept();
                        System.out.println("New connection accepted");
```

Socket-related support is provided by the package `java.net`, so we import that.

In case 1, we define a variable of type `ServerSocket` and bind it to a specific port number. A port number is an integer in the range 0–65535, where port numbers 0–1023 are reserved for special purposes. So, unless a system is documented otherwise, port numbers 1024–65535 are available for use by application program servers. This program gets its port number from the command line; however, the conversion code is irrelevant and is not shown here.

Objects of type `ServerSocket` are only used by servers. Basically, the creation of an object of this type declares that this server is open for communication on the given port number.

A port number of 0 tells the system to assign any available port. The server can then find out this port number by calling `getLocalPort`. It can then make this available to clients via a file or some other mechanism, so they will know which port number to use for connection requests.

If a security manager is present, the port number used is checked to see if we have permission for this operation.

Once a ServerSocket has been created, the server can listen using it, to see if any client wants to establish communication. This is done in case 2, via the method `accept`. Once this method is called, the calling thread is placed in an efficient wait state until a client connection request is detected for the given port. When that happens, an object of type `Socket` is created and it is returned to the server. This Socket is the channel by which the server and its new client can communicate.

A server can be connected to multiple clients at the same time. By default, a ServerSocket supports up to 50 concurrent client connections; however, that number can be specified when the ServerSocket is constructed. (This capability is not shown here.)

By default, `accept` waits indefinitely; however, we can establish a wait limit in milliseconds by calling `setSoTimeout`. A limit of zero means "wait indefinitely". If the time limit expires before a connect request is seen, an exception of type `InterruptedIOException` is thrown. The wait limit can be obtained by calling `getSoTimeout`.

```
/*3a*/                  InputStream is = soc.getInputStream();
/*3b*/                  DataInputStream dis = new DataInputStream(is);
/*3c*/                  OutputStream os = soc.getOutputStream();
/*3d*/                  DataOutputStream dos = new DataOutputStream(os);
```

The methods `getInputStream` and `getOutputStream` called in cases 3a and 3c, associate an input stream and an output stream, respectively, to this Socket. We need both in this server, because we will both read from and write to the client via this Socket. Of course, one-way communication is also possible. These low-level streams are then associated with higher-level ones in cases 3b and 3d, allowing us to use the `readInt` and `writeInt` methods in cases 4a, 4b, and 4c.

```
                        try
                        {
                                int value1, value2;
                                int result;

                                while (true)
                                {
/*4a*/                                  value1 = dis.readInt();
/*4b*/                                  value2 = dis.readInt();
                                        System.out.println("Received values "
                                                + value1 + " and " + value2);

                                        result = value1 + value2;
/*4c*/                                  dos.writeInt(result);
                                        System.out.println("Sent result " +
result);
                                }
                        }
```

The server loops indefinitely, reading pairs of integers, computing their sum, and returning the result back to the client. When end-of-file is detected on the input stream (by the client having closed its end of the socket), an exception of type `EOFException` is thrown, resulting in the closing of the Socket's I/O streams and the server's sockets, in case 5. Then the server terminates.

```
/*5*/                   catch (EOFException e)
                        {
                                dis.close();
                                dos.close();
                                soc.close();
                                ser.close();
                        }
                }
```