

Advanced Programming in Java™

Rex Jaeschke

Advanced Programming in Java

© 1999–2000, 2002, 2005, 2009 Rex Jaeschke. All rights reserved.

Edition: 2.0 (matches JDK1.6/Java 2)

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means whatsoever, except in the case of brief quotations embodied in critical reviews and articles.

The information in this book is subject to change without notice, and should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and the publisher assume no responsibility for errors or omissions.

Java is a trademark of Sun Microsystems.

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface	v
Reader Assumptions	v
Exercises and Solutions	v
The Java Development Kit	vi
Acknowledgments	vi
1. Threads	1
1.1 Introduction	1
1.2 Creating Threads	2
1.3 Synchronized Methods	6
1.4 Synchronized Statements	9
1.5 Other Forms of Synchronization	12
1.6 Managing Threads	16
1.7 Thread Groups	18
1.8 The Runnable Interface	20
1.9 volatile Fields	21
1.10 Thread-Local Storage	22
1.11 Applets with Multiple Threads	26
2. Object Serialization	35
2.1 Introduction	35
2.2 Serializing Objects that Contain References	42
2.3 Handling Multiple References	45
2.4 Customized Serialization	47
2.5 Identifying the Fields to be Serialized	50
2.6 javadoc and Serialization	52
2.7 Class Evolution and Versioning	55
2.8 Miscellaneous Features	58
2.8.1 The Externalizable Interface	58
2.8.2 Serializing an Applet	58
3. Sockets	61
3.1 Introduction	61
3.2 Server Sockets	61
3.3 Client Sockets	64
3.4 Serialization over Sockets	67
3.5 Networking	69
3.6 Miscellaneous Issues	69
4. Cloning Objects	71
4.1 Copying by Constructor	71
4.2 Class Cloning	72
4.3 The Method clone	73
4.4 Using Object.clone	74
4.5 Cloning Arrays	77
4.6 Creation without Construction	79
4.7 Miscellaneous Issues	81
5. Documentation Comments	83
5.1 Introduction	83
5.2 A Detailed Example	83

Advanced Programming in Java

5.3	javadoc Tags	95
5.4	javadoc Tool Reference	95
5.4.1	Doclets	95
5.4.2	Miscellaneous Issues	96
6.	Java Archives	97
6.1	Introduction	97
6.2	Manifest Files.....	99
6.3	JAR Files vs. ZIP Files	100
6.4	Package <code>java.util.zip</code>	100
6.5	Package <code>java.util.jar</code>	100
6.6	jar Tool Reference	100
Annex A.	Operator Precedence	103
Annex B.	Java Language Keywords	105
Index	107

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

Experience in writing applets will be useful but is not necessary.

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.

¹ The solutions are only available to licensees of these materials when they are used in formal training scenarios.
© 1999–2000, 2002, 2005, 2009 Rex Jaeschke.

The Java Development Kit

Sun's initial production release of Java was the Java Development Kit, version 1.0. Versions 1.1 through 1.5 contained numerous bug fixes and language and library enhancement. The latest version can be downloaded from Sun's website (www.sun.com).

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 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 Sun's on-line documentation to find the recommended replacement.

From an internationalization 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, September 2009

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 be 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.

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

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 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.) 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`. That class has two constructors, one method, 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 has the form `Thread-n`, where n is an integer.¹

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

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 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
<code>close</code>	Closes the stream
<code>flush</code>	Flushes the stream
<code>write</code>	Writes out one or more bytes
<code>writeObject</code>	Write out an object

Table 2-2: Interface `DataOutput`'s Methods

Name	Purpose
<code>write</code>	Writes out one or more bytes
<code>writeBoolean</code>	Writes out a boolean
<code>writeByte</code>	Writes out a byte
<code>writeBytes</code>	Writes out a String
<code>writeChar</code>	Writes out a char
<code>writeChars</code>	Writes out a String
<code>writeDouble</code>	Writes out a double
<code>writeFloat</code>	Writes out a float
<code>writeInt</code>	Writes out an int
<code>writeLong</code>	Writes out a long
<code>writeShort</code>	Writes out a short
<code>writeUTF</code>	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 Sun¹); 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
<code>available</code>	Number of bytes that can be read without blocking
<code>close</code>	Closes the stream
<code>read</code>	Reads in one or more bytes
<code>readObject</code>	Read out an object
<code>skip</code>	Skips one or more bytes

¹ Refer to the JDK documentation set for the web location of the serialization specification.
© 1999–2000, 2002, 2005, 2009 Rex Jaeschke.

4. Cloning Objects

Unlike some other languages, Java provides no way to have an expression of some object type; the best we can do is to have an expression whose value is really a reference to that type. As a result, Java provides no linguistic way to copy an object. Assuming we can justify it, how then can we make a complete copy—that is, a *clone*—of an object?

4.1 Copying by Constructor

Consider the following code fragment (see directory Cn01):

```
public class Point
{
    private int xor;
    private int yor;

    // ...

    public Point(Point p)
    {
        xor = p.xor;
        yor = p.yor;
    }
}
```

The constructor creates a new `Point` using the contents of an existing one. (In C++, this kind of constructor is so special, it has the name *copy constructor*; however, this term is not used in the context of Java.)

```
public class Cn01
{
    public static void main(String[] args)
    {
        /*1*/      Point p1 = new Point(3, 5);
        /*2*/      System.out.println("p1: " + p1);

        /*3*/      Point p2 = new Point(p1);

        /*4*/      p1.move(9, 11);

        /*5*/      System.out.println("p1: " + p1);
        /*6*/      System.out.println("p2: " + p2);
    }
}
```

Point p2 is a copy of p1, so when the coordinate values of p1 are changed in case 4, we see from the following output that those of p2 do not:

```
p1: (3,5)
p1: (9,11)
p2: (3,5)
```

While this approach works fine, we won't find many standard library classes that define this form of constructor, however.

4.2 Class Cloning

Those few standard library classes that do support some sort of copy mechanism achieve it by what is referred to as *cloning*. Consider the following example (see directory Cn02) that uses the library class `Vector`:

```
import java.util.Vector;

public class Cn02
{
    public static void main(String[] args)
    {
        Vector v1 = new Vector();

/*1*/        v1.addElement("Red");
              v1.addElement("Blue");
              v1.addElement("Green");
              v1.addElement("Yellow");
/*2*/        System.out.println("v1: " + v1);

/*3*/        Vector v2 = (Vector)v1.clone();
/*4*/        System.out.println("v2: " + v2);

/*5*/        v1.removeElement("Blue");
              v1.addElement("Black");
              v1.setElementAt("Brown", 0);

/*6*/        System.out.println("v1: " + v1);
/*7*/        System.out.println("v2: " + v2);
    }
}
```

The output produced is:

```
v1: [Red, Blue, Green, Yellow]
v2: [Red, Blue, Green, Yellow]
v1: [Brown, Green, Yellow, Black]
v2: [Red, Blue, Green, Yellow]
```

Vector `v1` consists of a set of 4 strings specifying different colors. We then make a complete copy of that object in case 3 by calling the method `Vector:clone`, so the output produced by cases 2 and 4 is the same. Then we modify `v1` by removing the second element, by adding a new element to the end, and by changing the first element's value. When we compare the output produced by cases 6 and 7, we see that the changes applied to `v1` have no effect on `v2`. Specifically, the internal reference inside of `v2` points to its own private copy of elements, not to the same set as `v1`. This is referred to as a *deep copy*, whereas simply making both Vectors' internal reference point to the same set of values is called a *shallow copy*.

If we wish to make copies of our own objects, we would do well to model the copy process on the library cloning machinery.

4.3 The Method `clone`

The key to cloning is to define a method called `clone`. For example, consider the following program fragment (see directory `Cn04`):

```
public class Point implements Cloneable
{
    private int xor;
    private int yor;

    // ...

    public Object clone()
    {
        return new Point(xor, yor);
    }
}
```

By convention, `clone` takes no arguments and returns a reference to type `Object`. The reason we use this signature is that we are really overriding the method `Object:clone`. In this example, `clone` simply constructs a new `Point` whose coordinates are identical to those of the current one, and returns a reference to that new `Point`.