

PEARSON

JAVATM 核心技术

Core Java Volume I — Fundamentals, Ninth Edition

卷 I : 基础知识 下

(第9版 · 英文版)



[美] Cay S. Horstmann Gary Cornell 著

人民邮电出版社
POSTS & TELECOM PRESS

JAVA 核心技术

Core Java Volume I — Fundamentals, Ninth Edition

卷 I：基础知识 下

(第9版·英文版)

[美] Cay S. Horstmann Gary Cornell 著

人民邮电出版社
北京

Contents

Chapter 10: Deploying Applications and Applets	565
10.1 JAR Files	566
10.1.1 The Manifest	567
10.1.2 Executable JAR Files	568
10.1.3 Resources	569
10.1.4 Sealing	573
10.2 Java Web Start	574
10.2.1 The Sandbox	578
10.2.2 Signed Code	579
10.2.3 The JNLP API	582
10.3 Applets	591
10.3.1 A Simple Applet	591
10.3.1.1 Converting Applications to Applets	595
10.3.2 The applet HTML Tag and Its Attributes	596
10.3.3 The object Tag	599
10.3.4 Use of Parameters to Pass Information to Applets	600
10.3.5 Accessing Image and Audio Files	606
10.3.6 The Applet Context	607
10.3.6.1 Inter-Applet Communication	608
10.3.6.2 Displaying Items in the Browser	609
10.4 Storage of Application Preferences	610
10.4.1 Property Maps	611
10.4.2 The Preferences API	616
Chapter 11: Exceptions, Assertions, Logging, and Debugging	625
11.1 Dealing with Errors	626
11.1.1 The Classification of Exceptions	628
11.1.2 Declaring Checked Exceptions	630
11.1.3 How to Throw an Exception	632
11.1.4 Creating Exception Classes	634

11.2	Catching Exceptions	635
11.2.1	Catching Multiple Exceptions	637
11.2.2	Rethrowing and Chaining Exceptions	639
11.2.3	The finally Clause	640
11.2.4	The Try-with-Resources Statement	644
11.2.5	Analyzing Stack Trace Elements	646
11.3	Tips for Using Exceptions	649
11.4	Using Assertions	653
11.4.1	Assertion Enabling and Disabling	654
11.4.2	Using Assertions for Parameter Checking	655
11.4.3	Using Assertions for Documenting Assumptions	656
11.5	Logging	657
11.5.1	Basic Logging	658
11.5.2	Advanced Logging	658
11.5.3	Changing the Log Manager Configuration	661
11.5.4	Localization	662
11.5.5	Handlers	663
11.5.6	Filters	667
11.5.7	Formatters	667
11.5.8	A Logging Recipe	668
11.6	Debugging Tips	677
11.7	Tips for Troubleshooting GUI Programs	682
11.7.1	Letting the AWT Robot Do the Work	686
11.8	Using a Debugger	690
Chapter 12:	Generic Programming	697
12.1	Why Generic Programming?	698
12.1.1	Who Wants to Be a Generic Programmer?	699
12.2	Defining a Simple Generic Class	700
12.3	Generic Methods	702
12.4	Bounds for Type Variables	704
12.5	Generic Code and the Virtual Machine	706
12.5.1	Translating Generic Expressions	708
12.5.2	Translating Generic Methods	708
12.5.3	Calling Legacy Code	711
12.6	Restrictions and Limitations	712

12.6.1	Type Parameters Cannot Be Instantiated with Primitive Types	712
12.6.2	Runtime Type Inquiry Only Works with Raw Types	712
12.6.3	You Cannot Create Arrays of Parameterized Types	713
12.6.4	Varargs Warnings	713
12.6.5	You Cannot Instantiate Type Variables	715
12.6.6	Type Variables Are Not Valid in Static Contexts of Generic Classes	717
12.6.7	You Cannot Throw or Catch Instances of a Generic Class ...	717
	12.6.7.1 You Can Defeat Checked Exception Checking	718
12.6.8	Beware of Clashes after Erasure	720
12.7	Inheritance Rules for Generic Types	721
12.8	Wildcard Types	723
12.8.1	Supertype Bounds for Wildcards	725
12.8.2	Unbounded Wildcards	728
12.8.3	Wildcard Capture	728
12.9	Reflection and Generics	731
12.9.1	Using <code>Class<T></code> Parameters for Type Matching	732
12.9.2	Generic Type Information in the Virtual Machine	733
Chapter 13:	Collections	741
13.1	Collection Interfaces	741
13.1.1	Separating Collection Interfaces and Implementation	742
13.1.2	Collection and Iterator Interfaces in the Java Library	745
	13.1.2.1 Iterators	745
	13.1.2.2 Removing Elements	748
	13.1.2.3 Generic Utility Methods	748
13.2	Concrete Collections	751
13.2.1	Linked Lists	752
13.2.2	Array Lists	762
13.2.3	Hash Sets	763
13.2.4	Tree Sets	767
13.2.5	Object Comparison	768
13.2.6	Queues and Deques	774
13.2.7	Priority Queues	776
13.2.8	Maps	777

13.2.9	Specialized Set and Map Classes	782
13.2.9.1	Weak Hash Maps	782
13.2.9.2	Linked Hash Sets and Maps	783
13.2.9.3	Enumeration Sets and Maps	785
13.2.9.4	Identity Hash Maps	785
13.3	The Collections Framework	787
13.3.1	Views and Wrappers	792
13.3.1.1	Lightweight Collection Wrappers	793
13.3.1.2	Subranges	794
13.3.1.3	Unmodifiable Views	794
13.3.1.4	Synchronized Views	796
13.3.1.5	Checked Views	796
13.3.1.6	A Note on Optional Operations	797
13.3.2	Bulk Operations	799
13.3.3	Converting between Collections and Arrays	800
13.4	Algorithms	801
13.4.1	Sorting and Shuffling	802
13.4.2	Binary Search	805
13.4.3	Simple Algorithms	806
13.4.4	Writing Your Own Algorithms	808
13.5	Legacy Collections	810
13.5.1	The Hashtable Class	810
13.5.2	Enumerations	810
13.5.3	Property Maps	811
13.5.4	Stacks	812
13.5.5	Bit Sets	813
13.5.5.1	The “Sieve of Eratosthenes” Benchmark	814
Chapter 14:	Multithreading	819
14.1	What Are Threads?	820
14.1.1	Using Threads to Give Other Tasks a Chance	827
14.2	Interrupting Threads	833
14.3	Thread States	836
14.3.1	New Threads	836
14.3.2	Runnable Threads	836
14.3.3	Blocked and Waiting Threads	837

14.3.4	Terminated Threads	839
14.4	Thread Properties	839
14.4.1	Thread Priorities	840
14.4.2	Daemon Threads	841
14.4.3	Handlers for Uncaught Exceptions	841
14.5	Synchronization	843
14.5.1	An Example of a Race Condition	843
14.5.2	The Race Condition Explained	848
14.5.3	Lock Objects	850
14.5.4	Condition Objects	854
14.5.5	The synchronized Keyword	859
14.5.6	Synchronized Blocks	864
14.5.7	The Monitor Concept	865
14.5.8	Volatile Fields	866
14.5.9	Final Variables	867
14.5.10	Atomics	868
14.5.11	Deadlocks	868
14.5.12	Thread-Local Variables	871
14.5.13	Lock Testing and Timeouts	873
14.5.14	Read/Write Locks	874
14.5.15	Why the stop and suspend Methods Are Deprecated	875
14.6	Blocking Queues	877
14.7	Thread-Safe Collections	886
14.7.1	Efficient Maps, Sets, and Queues	886
14.7.2	Copy on Write Arrays	888
14.7.3	Older Thread-Safe Collections	888
14.8	Callables and Futures	890
14.9	Executors	895
14.9.1	Thread Pools	896
14.9.2	Scheduled Execution	900
14.9.3	Controlling Groups of Tasks	901
14.9.4	The Fork-Join Framework	902
14.10	Synchronizers	905
14.10.1	Semaphores	906
14.10.2	Countdown Latches	907

14.10.3 Barriers	907
14.10.4 Exchangers	908
14.10.5 Synchronous Queues	908
14.11 Threads and Swing	909
14.11.1 Running Time-Consuming Tasks	910
14.11.2 Using the Swing Worker	915
14.11.3 The Single-Thread Rule	923
Appendix: Java Keywords	925
Index	929

Deploying Applications and Applets

In this chapter:

- JAR Files, page 566
- Java Web Start, page 574
- Applets, page 591
- Storage of Application Preferences, page 610

At this point, you should be comfortable with using most of the features of the Java programming language, and you've had a pretty thorough introduction to basic graphics programming in Java. Now that you are ready to create applications for your users, you will want to know how to package them for deployment on your users' computers. The traditional deployment choice—which was responsible for the unbelievable hype during the first few years of Java's life—is to use *applets*. An applet is a special kind of Java program that a Java-enabled browser can download from the Internet and run. The hopes were that users would be freed from the hassles of installing software and that they could access their software from any Java-enabled computer or device with an Internet connection.

For a number of reasons, applets never quite lived up to these expectations. Therefore, we will start this chapter with instructions for packaging applications. We then turn to the *Java Web Start* mechanism—an alternative approach for Internet-based application delivery which fixes some of the problems of applets.

Finally, we will cover applets and show you in which circumstances you still want to use them.

We will also discuss how your applications can store configuration information and user preferences.

10.1 JAR Files

When you package your application, you want to give your users a single file, not a directory structure filled with class files. Java Archive (JAR) files were designed for this purpose. A JAR file can contain both class files and other file types such as image and sound files. Moreover, JAR files are compressed, using the familiar ZIP compression format.



TIP: An alternative to the ZIP format is the “pack200” compression scheme that is specifically tuned to compress class files more efficiently. Oracle claims a compression rate of close to 90% for class files. See <http://docs.oracle.com/javase/1.5.0/docs/guide/deployment/deployment-guide/pack200.html> for more information.

Use the `jar` tool to make JAR files. (In the default JDK installation, it’s in the `jdk/bin` directory.) The most common command to make a new JAR file uses the following syntax:

```
jar cvf JARFileName File1 File2 . . .
```

For example:

```
jar cvf CalculatorClasses.jar *.class icon.gif
```

In general, the `jar` command has the following format:

```
jar options File1 File2 . . .
```

Table 10.1 lists all the options for the `jar` program. They are similar to the options of the UNIX `tar` command.

You can package application programs, program components (sometimes called “beans”—see Chapter 8 of Volume II), and code libraries into JAR files. For example, the runtime library of the JDK is contained in a very large file `rt.jar`.

Table 10.1 jar Program Options

Option	Description
c	Creates a new or empty archive and adds files to it. If any of the specified file names are directories, the jar program processes them recursively.
C	Temporarily changes the directory. For example, <pre>jar cvf JARFileName.jar -C classes *.class</pre> changes to the classes subdirectory to add class files.
e	Creates an entry point in the manifest (see Section 10.1.2, “Executable JAR Files,” on p. 568).
f	Specifies the JAR file name as the second command-line argument. If this parameter is missing, jar will write the result to standard output (when creating a JAR file) or read it from standard input (when extracting or tabulating a JAR file).
i	Creates an index file (for speeding up lookups in a large archive).
m	Adds a <i>manifest</i> to the JAR file. A manifest is a description of the archive contents and origin. Every archive has a default manifest, but you can supply your own if you want to authenticate the contents of the archive.
M	Does not create a manifest file for the entries.
t	Displays the table of contents.
u	Updates an existing JAR file.
v	Generates verbose output.
x	Extracts files. If you supply one or more file names, only those files are extracted. Otherwise, all files are extracted.
0	Stores without ZIP compression.

10.1.1 The Manifest

In addition to class files, images, and other resources, each JAR file contains a *manifest* file that describes special features of the archive.

The manifest file is called `MANIFEST.MF` and is located in a special `META-INF` subdirectory of the JAR file. The minimum legal manifest is quite boring—just

```
Manifest-Version: 1.0
```

Complex manifests can have many more entries. The manifest entries are grouped into sections. The first section in the manifest is called the *main section*. It applies to the whole JAR file. Subsequent entries can specify properties of named entities such as individual files, packages, or URLs. Those entries must begin with a *Name* entry. Sections are separated by blank lines. For example:

```
Manifest-Version: 1.0  
lines describing this archive
```

```
Name: Woozle.class  
lines describing this file  
Name: com/mycompany/mypkg/  
lines describing this package
```

To edit the manifest, place the lines that you want to add to the manifest into a text file. Then run

```
jar cfm JARFileName ManifestFileName . . .
```

For example, to make a new JAR file with a manifest, run

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

To update the manifest of an existing JAR file, place the additions into a text file and use a command such as

```
jar ufm MyArchive.jar manifest-additions.mf
```



NOTE: See <http://docs.oracle.com/javase/7/docs/technotes/guides/jar> for more information on the JAR and manifest file formats.

10.1.2 Executable JAR Files

You can use the *e* option of the *jar* command to specify the *entry point* of your program—the class that you would normally specify when invoking the *java* program launcher:

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass files to add
```

Alternatively, you can specify the *main class* of your program in the manifest, including a statement of the form

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

Do not add a *.class* extension to the main class name.



CAUTION: The last line in the manifest must end with a newline character. Otherwise, the manifest will not be read correctly. It is a common error to produce a text file containing just the `Main-Class` line without a line terminator.

With either method, users can simply start the program as

```
java -jar MyProgram.jar
```

Depending on the operating system configuration, users may even be able to launch the application by double-clicking the JAR file icon. Here are behaviors for various operating systems:

- On Windows, the Java runtime installer creates a file association for the “.jar” extension that launches the file with the `javaw -jar` command. (Unlike the `java` command, the `javaw` command doesn’t open a shell window.)
- On Solaris, the operating system recognizes the “magic number” of a JAR file and starts it with the `java -jar` command.
- On Mac OS X, the operating system recognizes the “.jar” file extension and executes the Java program when you double-click a JAR file.

However, a Java program in a JAR file does not have the same feel as a native application. On Windows, you can use third-party wrapper utilities that turn JAR files into Windows executables. A wrapper is a Windows program with the familiar .exe extension that locates and launches the Java virtual machine (JVM) or tells the user what to do when no JVM is found. There are a number of commercial and open source products, such as JSmooth (<http://jsmooth.sourceforge.net>) and Launch4J (<http://launch4j.sourceforge.net>). The open source installer generator IzPack (<http://izpack.org>) also contains a native launcher. For more information on this topic, see www.javalobby.org/articles/java2exe.

On the Macintosh, the situation is a bit easier. The Jar Bundler utility that is a part of XCode lets you turn a JAR file into a first-class Mac application.

10.1.3 Resources

Classes used in both applets and applications often have associated data files, such as

- Image and sound files;
- Text files with message strings and button labels; or
- Files with binary data, for example, to describe the layout of a map.

In Java, such an associated file is called a *resource*.



NOTE: In Windows, the term “resource” has a more specialized meaning. Windows resources also consist of images, button labels, and so on, but they are attached to the executable file and accessed by a standard programming interface. In contrast, Java resources are stored as separate files, not as part of class files. It is up to each program to access and interpret the resource data.

For example, consider a class, `AboutPanel`, that displays a message such as the one in Figure 10.1.

Of course, the book title and copyright year in the panel will change for the next edition of the book. To make it easy to track this change, we will put the text inside a file and not hardcode it as a string.

But where should you put a file such as `about.txt`? Of course, it would be convenient to simply place it with the rest of the program files inside the JAR file.

The class loader knows how to search for class files until it has located them somewhere on the class path, or in an archive, or on a web server. The resource mechanism gives you the same convenience for files that aren’t class files. Here are the necessary steps:

1. Get the `Class` object of the class that has a resource—for example, `AboutPanel.class`.
2. If the resource is an image or audio file, call `getResource(filename)` to get the resource location as a URL. Then read it with the `getImage` or `getAudioClip` method.

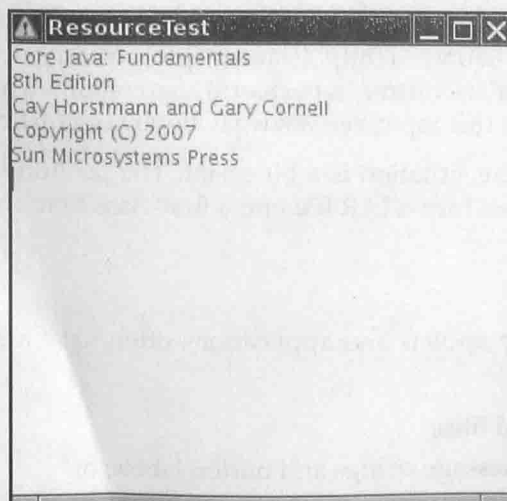


Figure 10.1 Displaying a resource from a JAR file

3. For resources other than images or audio files, use the `getResourceAsStream` method to read the data in the file.

The point is that the class loader remembers how to locate the class, so it can then search for the associated resource in the same location.

For example, to make an icon with the image file `about.gif`, do the following:

```
URL url = ResourceTest.class.getResource("about.gif");  
Image img = new ImageIcon(url).getImage();
```

That means “locate the `about.gif` file in the same place where you found the `ResourceTest` class.”

To read in the file `about.txt`, use these commands:

```
InputStream stream = ResourceTest.class.getResourceAsStream("about.txt");  
Scanner in = new Scanner(stream);
```

Instead of placing a resource file inside the same directory as the class file, you can place it in a subdirectory. You can then use a hierarchical resource name such as

```
data/text/about.txt
```

This is a relative resource name, and it is interpreted relative to the package of the class that loads the resource. Note that you must always use the `/` separator, regardless of the directory separator on the system that actually stores the resource files. For example, on the Windows file system, the resource loader automatically translates `/` to `\` separators.

A resource name starting with a `/` is called an absolute resource name. It is located in the same way a class inside a package would be located. For example, a resource

```
/corejava/title.txt
```

is located in the `corejava` directory which may be a subdirectory of the class path, inside a JAR file, or, for applets, on a web server.

Automating the loading of files is all the resource loading feature does. There are no standard methods for interpreting the contents of a resource file. Each program must have its own way of interpreting its resource files.

Another common application of resources is the internationalization of programs. Language-dependent strings, such as messages and user interface labels, are stored in resource files, with one file per language. The *internationalization API*, which is discussed in Chapter 5 of Volume II, supports a standard method for organizing and accessing these localization files.

Listing 10.1 is a program that demonstrates resource loading. Compile, build a JAR file, and execute it:

```
javac ResourceTest.java
jar cvfm ResourceTest.jar ResourceTest.mf *.class *.gif *.txt
java -jar ResourceTest.jar
```

Move the JAR file to a different directory and run it again to check that the program reads the resource files from the JAR file, not from the current directory.

Listing 10.1 resource/ResourceTest.java

```
1 package resource;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.net.*;
6 import java.util.*;
7 import javax.swing.*;
8
9 /**
10  * @version 1.4 2007-04-30
11  * @author Cay Horstmann
12  */
13 public class ResourceTest
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(new Runnable()
18         {
19             public void run()
20             {
21                 JFrame frame = new ResourceTestFrame();
22                 frame.setTitle("ResourceTest");
23                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24                 frame.setVisible(true);
25             }
26         });
27     }
28 }
29
30 /**
31  * A frame that loads image and text resources.
32  */
```



```

33 class ResourceTestFrame extends JFrame
34 {
35     private static final int DEFAULT_WIDTH = 300;
36     private static final int DEFAULT_HEIGHT = 300;
37
38     public ResourceTestFrame()
39     {
40         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41         URL aboutURL = getClass().getResource("about.gif");
42         Image img = new ImageIcon(aboutURL).getImage();
43         setIconImage(img);
44
45         JTextArea textArea = new JTextArea();
46         InputStream stream = getClass().getResourceAsStream("about.txt");
47         Scanner in = new Scanner(stream);
48         while (in.hasNext())
49             textArea.append(in.nextLine() + "\n");
50         add(textArea);
51     }
52 }

```

java.lang.Class 1.0

- URL `getResource(String name)` **1.1**
- InputStream `getResourceAsStream(String name)` **1.1**
finds the resource in the same place as the class and then returns a URL or input stream that you can use for loading the resource. Returns null if the resource isn't found, so does not throw an exception for an I/O error.

10.1.4 Sealing

We mentioned in Chapter 4 that you can *seal* a Java language package to ensure that no further classes can add themselves to it. You would want to seal a package if you use package-visible classes, methods, and fields in your code. Without sealing, other classes can place themselves into the same package and thereby gain access to its package-visible features.

For example, if you seal the package `com.mycompany.util`, then no class outside the sealed archive can be defined with the statement

```
package com.mycompany.util;
```

To achieve this, put all classes of the package into a JAR file. By default, packages in a JAR file are not sealed. You can change that global default by placing the line

```
Sealed: true
```