JAVA

Fundamental Classes Reference

O'REILLY™

Mark Grand & Jonathan Knudsen

Java Fundamental Classes Reference

By Mark Grand and Jonathan Knudsen; 1-56592-241-7, 1152 pages
1st Edition May 1997

Table of Contents

Part III: Appendixes

This part provides information about the Unicode 2.0 standard and the UTF-8 encoding used by Java.

Search the text of *Java Fundamental Classes Reference*.

Library Home | Java in a Nutshell | Java Language Reference | Java AWT Reference | Java Fundamental Classes Reference | Exploring Java

Preface

Contents:

This book is a reference manual for the *fundamental classes* in the Java programming environment; it covers version 1.1 of the Java API. We've defined fundamental classes to mean those classes in the Java Development Kit (JDK) that every Java programmer is likely to need, minus the classes that comprise the Abstract Window Toolkit (AWT). (The classes in the AWT are covered by a companion volume, the *Java AWT Reference*, from O'Reilly & Associates.) Thus, this book covers the classes in the `java.lang` and `java.io` packages, among others, and is essential for the practicing Java programmer.

This is an exciting time in the development of Java. Version 1.1 introduces a massive amount of infrastructure that more than doubles the size of the core Java APIs. This new infrastructure provides many new facilities, such as:

- Java is now more dynamic. An expanded `Class` class, in conjunction with the new `java.lang.reflect` package, allows objects to access methods and variables of objects that they were not compiled with.
- There are classes in `java.io` that build on the new dynamic capabilities to provide the ability to read and write objects as streams of bytes.
- There is increased support for internationalization. The support includes a `Locale` class and classes to format and parse data in locale-specific ways. There is also support for loading external locale-specific resources, such as textual strings.

- The `java.util.zip` package provides the ability to read and write compressed files.
- The `java.math` package provides the ability to perform arithmetic operations to any degree of precision that is necessary.

There are also more ways to package and distribute Java programs. In addition to being able to build command-line based applications and applets that are hosted by browsers, we now have the Java Servelet API that allows Java programs to function as part of a web server. Furthermore, the nature of applets may be changing. Instead of waiting for large applet to be downloaded by a browser, we now have push technologies such as Marimba's Castanet that ensure that the most current version of an applet is already on our machine when we want to run it.

Many new uses for Java have appeared or are on the horizon. For example, NASA is using Java applets to monitor telemetry data, instead of building more large, dedicated hardware consoles. Cellular phone manufacturers have committed to making cellular phone models that support Java, so in the future we may see Java programs that run on cellular phones and allow us to check e-mail or view location maps. Many additional APIs are also on the way, from Sun and other companies. These APIs not only supply infrastructure, but also provide frameworks for building domain-specific applications, in such areas as electronic commerce and manufacturing.

This book is about the classes that provide the most fundamental infrastructure for Java. As you use this book, we hope that you will share our enthusiasm for the richness of what is provided and the anticipation of what is yet to come.

What This Book Covers

The *Java Fundamental Classes Reference* is the definitive resource for programmers working with the core, non-AWT classes in Java. It covers all aspects of these fundamental classes as of version 1.1.1 of Java. If there are any changes to these classes after 1.1.1 (at least one more patch release is expected), we will integrate them as soon as possible. Watch the book's web site, http://www.ora.com/catalog/javafund/, for details on changes.

Specifically, this book completely covers the following packages:

- `java.io` (1.0 and 1.1)
- `java.lang` (1.0 and 1.1)
- `java.lang.reflect` (new in 1.1)
- `java.math` (new in 1.1)
- `java.net` (1.0 and 1.1)

- `java.text` (new in 1.1)
- `java.util` (1.0 and 1.1)
- `java.util.zip` (new in 1.1)

As you can see from the list above, this book covers four packages that are completely new in Java 1.1. In addition, it includes material on all of the new features in the four original 1.0 packages. Here are the highlights of what is new in Java 1.1:

`java.lang`

> This package contains the new `Byte`, `Short`, and `Void` classes that are needed for the new Reflection API. The `Class` class also defines a number of new methods for the Reflection API. Chapter 12, *The java.lang Package*, contains reference material on all of the classes in the `java.lang` package.

`java.io`

> This package contains a number of new classes, mostly for object serialization and character streams. Chapter 11, *The java.io Package*, contains reference material on all of the classes in the `java.io` package.

`java.net`

> This package contains a new `MulticastSocket` class that supports multicast sockets and several new exception types for more detailed networking exceptions. Chapter 15, *The java.net Package*, contains reference material on all of the classes in the `java.net` package.

`java.util`

> This package includes a handful of new classes for internationalization, such as `Locale` and `ResourceBundle`. The package also defines the base classes that support the new AWT event model. The new `Calendar` and `TimeZone` classes provide increased support for working with dates and times. Chapter 17, *The java.util Package*, contains reference material on all of the classes in the `java.util` package.

`java.lang.reflect`

> This new package defines classes that implement the bulk of the new Reflection API. The classes in the package represent the fields, methods, and constructors of a class. Chapter 13, *The java.lang.reflect Package*, contains reference material on all of the classes in the `java.lang.reflect` package.

`java.math`

This new package includes two classes that support arithmetic: one with arbitrarily large integers and another with arbitrary-precision floating-point numbers. Chapter 14, *The java.math Package*, contains reference material on all of the classes in the `java.math` package.

`java.text`

This new package contains the majority of the classes that implement the internationalization capabilities of Java 1.1. It includes classes for formatting dates, times, numbers, and textual messages for any specified locale. Chapter 16, *The java.text Package*, contains reference material on all of the classes in the `java.text` package.

`java.util.zip`

This new package defines classes that support general-purpose data compression and decompression using the ZLIB compression algorithms, as well as classes that work with the popular GZIP and ZIP formats. Chapter 18, *The java.util.zip Package*, contains reference material on all of the classes in the `java.util.zip` package.

1. Introduction

Contents:

The phenomenon that is Java continues to capture new supporters every day. What began as a programming environment for writing fancy animation applets that could be embedded in web browsers is growing up to be a sophisticated platform for delivering all kinds of portable, distributed applications. If you are already an experienced Java programmer, you know just how powerful the portability of Java is. If you are just now discovering Java, you'll be happy to know that the days of porting applications are over. Once you write a Java application, it can run on UNIX workstations, PCs, and Macintosh computers, as well as on many other supported platforms.

This book is a complete programmer's reference to the "fundamental classes" in the Java programming environment. The fundamental classes in the Java Development Kit (JDK) provide a powerful set of tools for creating portable applications; they are an important component of the toolbox used by every Java programmer. This reference covers the classes in the `java.lang`, `java.io`, `java.net`, `java.util`, `java.lang.reflect`, `java.math`, `java.text`, and `java.util.zip` packages. This chapter offers an overview of the fundamental classes in each of these packages.

This reference assumes you are already familiar with the Java language and class libraries. If you aren't, *Exploring Java*, by Pat Niemeyer and Josh Peck, provides a general introduction, and other books in the O'Reilly Java series provide detailed references and tutorials on specific topics. Note that the material herein does not cover the classes that comprise the Abstract Window Toolkit (AWT): the AWT is covered by a companion volume, the *Java AWT Reference*, by John Zukowski. In addition, this book does not cover any of the new "enterprise" APIs in the core 1.1 JDK, such as the classes in the `java.rmi`, `java.sql`, and `java.security` packages. These packages will be covered by forthcoming books on distributed computing and database programming. See the Preface for a complete list of titles in the O'Reilly Java series.

You should be aware that this book covers two versions of Java: 1.0.2 and 1.1. Version 1.1 of the Java Development Kit (JDK) was released in February 1997. This release includes many improvements and additions to the fundamental Java classes; it represents a major step forward in the evolution of Java. Although Java 1.1 has a number of great new features, you may not want to switch to the new version right away, especially if you are writing mostly Java applets. You'll need to keep an eye on the state of Java support in browsers to help you decide when to switch to Java 1.1. Of course, if you are writing Java applications, you can take the plunge today.

This chapter points out new features of Java 1.1 as they come up. However, there is one "feature" that deserves mention that doesn't fit naturally into an overview. As of Java 1.1, classes, methods, and constructors available in Java 1.0.2 can be deprecated in favor of new classes, methods, and constructors in Java 1.1. The Java 1.1 compiler issues a warning whenever you use a deprecated entity.

1.1 The java.lang Package

The `java.lang` package contains classes and interfaces essential to the Java language. For example, the `Object` class is the ultimate superclass of all other classes in Java. `Object` defines some basic methods for thread synchronization that are inherited by all Java classes. In addition, `Object` defines basic methods for equality testing, hashcode generation, and string conversion that can be overridden by subclasses when appropriate.

The `java.lang` package also contains the `Thread` class, which controls the operation of each thread in a multithreaded application. A `Thread` object can be used to start, stop, and suspend a thread. A `Thread` must be associated with an object that implements the `Runnable` interface; the `run()` method of this interface specifies what the thread actually does. See Chapter 3, *Threads*, for a more detailed explanation of how threads work in Java.

The `Throwable` class is the superclass of all error and exception classes in Java, so it defines the basic functionality of all such classes. The `java.lang` package also defines the standard error and exception classes in Java. The error and exception hierarchies are rooted at the `Error` and `Exception` subclasses of `Throwable`. See Chapter 4, *Exception Handling*, for more information about the exception-handling mechanism.

The `Boolean`, `Character`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short` classes encapsulate the Java primitive data types. `Byte` and `Short` are new in Java 1.1, as is the `Void` class. All of these classes are necessary to support the new Reflection API and class literals in Java 1.1 The `Class` class also has a number of new methods in Java 1.1 to support reflection.

All strings in Java are represented by `String` objects. These objects are immutable. The

`StringBuffer` class in `java.lang` can be used to work with mutable text strings. Chapter 2, *Strings and Related Classes*, offers a more detailed description of working with strings in Java.

See Chapter 12, *The java.lang Package*, for complete reference material on all of the classes in the `java.lang` package.

PREVIOUS HOME NEXT
Acknowledgments BOOK INDEX The java.lang.reflect Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2. Strings and Related Classes

Contents:

As with most programming languages, strings are used extensively throughout Java, so the Java API has quite a bit of functionality to help you manipulate strings. This chapter describes the following classes:

- The `java.lang.String` class represents all textual strings in Java. A `String` object is immutable; once you create a `String` object, there is no way to change the sequence of characters it represents or the length of the string.
- The `java.lang.StringBuffer` class represents a variable-length, mutable sequence of characters. With a `StringBuffer` object, you can insert characters anywhere in the sequence and add characters to the end of the sequence.
- The `java.util.StringTokenizer` class provides support for parsing a string into a sequence of words, or tokens.

2.1 String

You can create a `String` object in Java simply by assigning a string literal to a `String` variable:

```
String quote = "To be or not to be";
```

All string literals are compiled into `String` objects. Although the Java compiler does not generally treat expressions involving object references as compile-time constants, references to `String` objects created from string literals are treated as compile-time constants.

Of course, there are many other ways to create a `String` object. The `String` class has a number of constructors that let you create a `String` from an array of bytes, an array of characters, another `String` object, or a `StringBuffer` object.

If you are a C or C++ programmer, you may be wondering if `String` objects are null-terminated. The answer is no, and, in fact, the question is irrelevant. The `String` class actually uses a character array internally. Since arrays in Java are actual objects that know their own length, a `String` object also knows its length and does not require a special terminator. Use the `length()` method to get the length of a `String` object.

Although `String` objects are immutable, the `String` class does provide a number of useful methods for working with strings. Any operation that would otherwise change the characters or the length of the string returns a new `String` object that copies the necessary portions of the original `String`.

The following methods access the contents of a `String` object:

- `substring()` creates a new `String` object that contains a sub-sequence of the sequence of characters represented by a `String` object.
- `charAt()` returns the character at a given position in a `String` object.
- `getChars()` and `getBytes()` return a range of characters in a `char` array or a `byte` array.
- `toCharArray()` returns the entire contents of a `String` object as a `char` array.

You can compare the contents of `String` objects with the following methods:

- `equals()` returns `true` if two `String` objects have the exact same contents, while `equalsIgnoreCase()` returns `true` if two objects have the same contents ignoring differences between upper- and lowercase versions of the same character.
- `regionMatches()` determines if two sub-strings contain the same sequence of characters.
- `startsWith()` and `endsWith()` determine if a `String` object begins or ends with a particular sequence of characters.
- `compareTo()` determines if the contents of one `String` object are less than, equal to, or greater than the contents of another `String` object.

Use the following methods to search for characters in a string:

- `indexOf()` searches forward through a string for a given character or string.

- `lastIndexOf()` searches backwards through a string for a given character or string.

The following methods manipulate the contents of a string and return a new, related string:

- `concat()` returns a new `String` object that is the concatenation of two `String` objects.
- `replace()` returns a new `String` object that contains the same sequence of characters as the original string, but with a given character replaced by another given character.
- `toLowerCase()` and `toUpperCase()` return new `String` objects that contain the same sequence of characters as the original string, but converted to lower- or uppercase.
- `trim()` returns a new `String` object that contains the same character sequence as the original string, but with leading and trailing white space and control characters removed.

The `String` class also defines a number of `static` methods named `valueOf()` that return string representations of primitive Java data types and objects. The `Object` class defines a `toString()` method, and, since `Object` is the ultimate superclass of every other class, every class inherits a basic `toString()` method. Any class that has a string representation should override the `toString()` method to produce the appropriate string.

3. Threads

Contents:

Threads provide a way for a Java program to do multiple tasks concurrently. A thread is essentially a flow of control in a program and is similar to the more familiar concept of a process. An operating system that can run more than one program at the same time uses processes to keep track of the various programs that it is running. However, processes generally do not share any state, while multiple threads within the same application share much of the same state. In particular, all of the threads in an application run in the same address space, sharing all resources except the stack. In concrete terms, this means that threads share field variables, but not local variables.

When multiple processes share a single processor, there are times when the operating system must stop the processor from running one process and start it running another process. The operating system must execute a sequence of events called a *context switch* to transfer control from one process to another. When a context switch occurs, the operating system has to save a lot of information for the process that is being paused and load the comparable information for the process being resumed. A context switch between two processes can require the execution of thousands of machine instructions. The Java virtual machine is responsible for handling context switches between threads in a Java program. Because threads share much of the same state, a context switch between two threads typically requires the execution of less than 100 machine instructions.

There are a number of situations where it makes sense to use threads in a Java program. Some programs must be able to engage in multiple activities and still be able to respond to additional input from the user. For example, a web browser should be able to respond to user input while fetching an image or playing a sound. Because threads can be suspended and resumed, they can make it easier to control multiple activities, even if the activities do not need to be concurrent. If a program models real world objects that display independent, autonomous behavior, it makes sense to use a separate thread for each object. Threads can also implement asynchronous methods, so that a calling method does not have to wait for the method it calls to complete before continuing with its own activity.

Java applets make considerable use of threads. For example, an animation is generally implemented with a separate thread. If an applet has to download extensive information, such as an image or a sound, to initialize itself, the initialization can take a long time. This initialization can be done in a separate thread to prevent the initialization from interfering with the display of the applet. If an applet needs to process messages from the network, that work generally is done in a separate thread so that the applet can continue painting itself on the screen and responding to mouse and keyboard events. In addition, if each message is processed separately, the applet uses a separate thread for each message.

For all of the reasons there are to use threads, there are also some compelling reasons not to use them. If a program uses inherently sequential logic, where one operation starts another operation and then must wait for the other operation to complete before continuing, one thread can implement the entire sequence. Using multiple threads in such a case results in a more complex program with no accompanying benefits. There is considerable overhead in creating and starting a thread, so if an operation involves only a few primitive statements, it is faster to handle it with a single thread. This can even be true when the operation is conceptually asynchronous. When multiple threads share objects, the objects must use synchronization mechanisms to coordinate thread access and maintain consistent state. Synchronization mechanisms add complexity to a program, can be difficult to tune for optimal performance, and can be a source of bugs.

3.1 Using Thread Objects

The `Thread` class in the `java.lang` package creates and controls threads in Java programs. The execution of Java code is always under the control of a `Thread` object. The `Thread` class provides a `static` method called `currentThread()` that provides a reference to the `Thread` object that controls the current thread of execution.

Associating a Method with a Thread

The first thing you need to do to make a `Thread` object useful is to associate it with a method you want it to run. Java provides two ways of associating a method with a `Thread`:

- Declare a subclass of `Thread` that defines a `run()` method.
- Pass a reference to an object that implements the `Runnable` interface to a `Thread` constructor.

For example, if you need to load the contents of a URL as part of an applet's initialization, but the applet can provide other functionality before the content is loaded, you might want to load the content in a separate thread. Here is a class that does just that:

```
import java.net.URL;
class UrlData extends Thread   {
    private Object data;
    private URL url
    public UrlData(String urlName) throws MalformedURLException {
        url = new URL(urlName);
        start();
    }
    public void run(){
        try {
            data = url.getContent();
        } catch (java.io.IOException  e) {
        }
    }
    public Object getUrlData(){
        return data;
    }
}
```

The `UrlData` class is declared as a subclass of `Thread` so that it can get the contents of the URL in a separate thread. The constructor creates a `java.net.URL` object to fetch the contents of the URL, and then calls the `start()` method to start the thread. Once the thread is started, the constructor returns; it does not wait for the contents of the URL to be fetched. The `run()` method is executed after the thread is started; it does the real work of fetching the data. The `getUrlData()` method is an access method that returns the value of the `data` variable. The value of this variable is `null` until the contents of the URL have been fetched, at which time it contains a reference to the actual data.

Subclassing the `Thread` class is convenient when the method you want to run in a separate thread does not need to belong to a particular class. Sometimes, however, you need the method to be part of a particular class that is a subclass of a class other than `Thread`. Say, for example, you want a graphical object that is displayed in a window to alternate its background color between red and blue once a second. The object that implements this behavior needs to be a subclass of the `java.awt.Canvas` class. However, at the same time, you need a separate thread to alternate the color of the object once a second.

In this situation, you want to tell a `Thread` object to run code in another object that is not a subclass of the `Thread` class. You can accomplish this by passing a reference to an object that implements the `Runnable` interface to the constructor of the `Thread` class. The `Runnable` interface requires that an object has a `public` method called `run()` that takes no arguments. When a `Runnable` object is passed to the constructor of the `Thread` class, it creates a `Thread` object that calls the `Runnable` object's `run()` method when the thread is started. The following example shows part of the code that implements an object that alternates its background color between red and blue once a second:

```
class AutoColorChange extends java.awt.Canvas implements Runnable {
    private Thread myThread;
    AutoColorChange () {
        myThread = new Thread(this);
        myThread.start();
        ...
    }
    public void run() {
        while (true) {
            setBackground(java.awt.Color.red);
            repaint();
            try {
                myThread.sleep(1000);
            } catch (InterruptedException e) {}
            setBackground(java.awt.Color.blue);
            repaint();
            try {
                myThread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

The `AutoChangeColor` class extends `java.awt.Canvas`, alternating the background color between red and blue once a second. The constructor creates a new `Thread` by passing the current object to the `Thread` constructor, which tells the `Thread` to call the `run()` method in the `AutoChangeColor` class. The constructor then starts the new thread by calling its `start()` method, so that the color change happens asynchronously of whatever else is going on. The class has an instance variable called `myThread` that contains a reference to the `Thread` object, so that can control the thread. The `run()` method takes care of changing the background color, using the `sleep()` method of the `Thread` class to temporarily suspend the thread and calling `repaint()` to redisplay the object after each color change.

Controlling a Thread

As shown in the previous section, you start a `Thread` by calling its `start()` method. Before the `start()` method is called, the `isAlive()` method of the `Thread` object always returns `false`. When the `start()` method is called, the `Thread` object becomes associated with a scheduled thread in the underlying environment. After the `start()` method has returned, the `isAlive()` method always returns `true`. The `Thread` is now scheduled to run until it dies, unless it is suspended or in another unrunnable state.

It is actually possible for `isAlive()` to return `true` before `start()` returns, but not before `start()` is called. This can happen because the `start()` method can return either before the started `Thread` begins to run or after it begins to run. In other words, the method that called `start()` and the new thread are now running concurrently. On a multiprocessor system, the `start()` method can even return at the same time the started `Thread` begins to run.

`Thread` objects have a parent-child relationship. The first thread created in a Java environment does not have a parent `Thread`. However, after the first `Thread` object is created, the `Thread` object that controls the thread used to create another `Thread` object is considered to be the parent of the newly created `Thread`. This parent-child relationship is used to supply some default values when a `Thread` object is created, but it has no further significance after a `Thread` has been created.

Stopping a thread

A thread dies when one of the following things happens:

- The `run()` method called by the `Thread` returns.
- An exception is thrown that causes the `run()` method to be exited.
- The `stop()` method of the `Thread` is called.

The `stop()` method of the `Thread` class works by throwing a `ThreadDeath` object in the `run()` method of the thread. Normally, you should not catch `ThreadDeath` objects in a `try` statement. If you need to catch `ThreadDeath` objects to detect that a `Thread` is about to die, the `try` statement that catches `ThreadDeath` objects should rethrow them.

When an object (`ThreadDeath` or otherwise) is thrown out of the `run()` method for the `Thread`, the `uncaughtException()` method of the `ThreadGroup` for that `Thread` is called. If the thrown object is an instance of the `ThreadDeath` class, the thread dies, and the thrown object is ignored. Otherwise, if the thrown object is of any other class, `uncaughtException()` calls the thrown object's `printStackTrace()` method, the thread dies, and the thrown object is ignored. In either case, if there are other nondaemon threads running in the system, the current program continues to run.

Interrupting a thread

There are a number of methods in the Java API, such as `wait()` and `join()`, that are declared as throwing an `InterruptedException`. What these methods have in common is that they temporarily suspend the execution of a thread. In Java 1.1, if a thread is waiting for one of these methods to return and another thread calls `interrupt()` on the waiting thread, the method that is waiting throws an `InterruptedException`.

The `interrupt()` method sets an internal flag in a `Thread` object. Before the `interrupt()` method is called, the `isInterrupted()` method of the `Thread` object always returns `false`. After the `interrupt()` method is called, `isInterrupted()` returns `true`.

Prior to version 1.1, the methods in the Java API that are declared as throwing an `InterruptedException` do not actually do so. However, the `isInterrupted()` method does function as described above. Thus, if the code in the `run()` method for a thread periodically calls `isInterrupted()`, the thread can respond to a call to `interrupt()` by shutting down in an orderly fashion.

Thread priority

One of the attributes that controls the behavior of a thread is its priority. Although Java does not guarantee much about how threads are scheduled, it does guarantee that a thread with a priority that is higher than that of another thread will be scheduled to run at least as often, and possibly more often, than the thread with the lower priority. The priority of a thread is set when the `Thread` object is created, by passing an argument to the constructor that creates the `Thread` object. If an explicit priority is not specified, the `Thread` inherits the priority of its parent `Thread` object.

You can query the priority of a `Thread` object by calling its `getPriority()` method. Similarly, you can set the priority of a `Thread` using its `setPriority()` method. The priority you specify must be greater than or equal to `Thread.MIN_PRIORITY` and less than or equal to `Thread.MAX_PRIORITY`.

Before actually setting the priority of a `Thread` object, the `setPriority()` method checks the maximum allowable priority for the `ThreadGroup` that contains the `Thread` by calling `getMaxPriority()` on the `ThreadGroup`. If the call to `setPriority()` tries to set the priority to a value that is higher than the maximum allowable priority for the `ThreadGroup`, the priority is instead set to the maximum priority. It is possible for the current priority of a `Thread` to be greater than the maximum allowable priority for the `ThreadGroup`. In this case, an attempt to raise the priority of the `Thread` results in its priority being lowered to the maximum priority.

Daemon threads

A daemon thread is a thread that runs continuously to perform a service, without having any connection with the overall state of the program. For example, the thread that runs the garbage collector in Java is a daemon thread. The thread that processes mouse events for a Java program is also a daemon thread. In general, threads that run application code are not daemon threads, and threads that run system code are daemon threads. If a thread dies and there are no other threads except daemon threads alive, the Java virtual machine stops.

A `Thread` object has a `boolean` attribute that specifies whether or not a thread is a daemon thread. The daemon attribute of a thread is set when the `Thread` object is created, by passing an argument to the constructor that creates the `Thread` object. If the daemon attribute is not explicitly specified, the `Thread` inherits the daemon attribute of its parent `Thread` object.

The daemon attribute is queried using the `isDaemon()` method; it is set using the `setDaemon()` method.

Yielding

When a thread has nothing to do, it can call the `yield()` method of its `Thread` object. This method tells the scheduler to run a different thread. The value of calling `yield()` depends largely on whether the scheduling mechanism for the platform on which the program is running is preemptive or nonpreemptive.

By choosing a maximum length of time a thread can continuously, a *preemptive* scheduling mechanism guarantees that no single thread uses more than its fair share of the processor. If a thread runs for that amount of time without yielding control to another thread, the scheduler preempts the thread and causes it to stop running so that another thread can run.

A *nonpreemptive* scheduling mechanism cannot preempt threads. A nonpreemptive scheduler relies on the individual threads to yield control of the processor frequently, so that it can provide reasonable performance. A thread explicitly yields control by calling the `Thread` object's `yield()` method. More often, however, a thread implicitly yields control when it is forced to wait for something to happen elsewhere.

Calling a `Thread` object's `yield()` method during a lengthy computation can be quite valuable on a platform that uses a nonpreemptive scheduling mechanism, as it allows other threads to run. Otherwise, the lengthy computation can prevent other threads from running. On a platform that uses a preemptive scheduling mechanism, calling `yield()` does not usually make any noticeable difference in the responsiveness of threads.

Regardless of the scheduling algorithm that is being used, you should not make any assumptions about when a thread will be scheduled to run again after it has called `yield()`. If you want to prevent a thread from being scheduled to run until a specified amount of time has elapsed, you should call the `sleep()` method of the `Thread` object. The `sleep()` method takes an argument that specifies a minimum number of milliseconds that must elapse before the thread can be scheduled to run again.

Controlling groups of threads

Sometimes is it necessary to control multiple threads at the same time. Java provides the `ThreadGroup` class for this purpose. Every `Thread` object belongs to a `ThreadGroup` object. By passing an argument to the constructor that creates the `Thread` object, the `ThreadGroup` of a thread can be set when the `Thread` object is created. If an explicit `ThreadGroup` is not specified, the `Thread` belongs to the same `ThreadGroup` as its parent `Thread` object.

JAVA Fundamental Classes Reference

PREVIOUS Chapter 4 NEXT

4. Exception Handling

Contents:

Exception handling is a mechanism that allows Java programs to handle various exceptional conditions, such as semantic violations of the language and program-defined errors, in a robust way. When an exceptional condition occurs, an *exception* is thrown. If the Java virtual machine or run-time environment detects a semantic violation, the virtual machine or run-time environment implicitly throws an exception. Alternately, a program can throw an exception explicitly using the `throw` statement. After an exception is thrown, control is transferred from the current point of execution to an appropriate `catch` clause of an enclosing `try` statement. The `catch` clause is called an exception handler because it handles the exception by taking whatever actions are necessary to recover from it.

4.1 Handling Exceptions

The `try` statement provides Java's exception-handling mechanism. A `try` statement contains a block of code to be executed. Putting a block in a `try` statement indicates that any exceptions or other abnormal exits in the block are going to be handled appropriately. A `try` statement can have any number of optional `catch` clauses that act as exception handlers for the `try` block. A `try` statement can also have a `finally` clause. The `finally` block is always executed before control leaves the `try` statement; it cleans up after the `try` block. Note that a `try` statement must have either a `catch` clause or a `finally` clause.

Here is an example of a `try` statement that includes a `catch` clause and a `finally` clause:

```
try {
    out.write(b);
} catch (IOException e) {
    System.out.println("Output Error");
} finally {
    out.close();
}
```

If `out.write()` throws an `IOException`, the exception is caught by the `catch` clause. Regardless of whether `out.write()` returns normally or throws an exception, the `finally` block is executed, which ensures that `out.close()` is always called.

A `try` statement executes the block that follows the keyword `try`. If an exception is thrown from within the `try` block and the `try` statement has any `catch` clauses, those clauses are searched, in order, for one that can handle the exception. If a `catch` clause handles an exception, that `catch` block is executed.

However, if the `try` statement does not have any `catch` clauses that can handle the exception (or does not have any `catch` clauses at all), the exception propagates up through enclosing statements in the current method. If the current method does not contain a `try` statement that can handle the exception, the exception propagates up to the invoking method. If this method does not contain an appropriate `try` statement, the exception propagates up again, and so on. Finally, if no `try` statement is found to handle the exception, the currently running thread terminates.

A `catch` clause is declared with a parameter that specifies the type of exception it can handle. The parameter in a `catch` clause must be of type `Throwable` or one of its subclasses. When an exception occurs, the `catch` clauses are searched for the first one with a parameter that matches the type of the exception thrown or is a superclass of the thrown exception. When the appropriate `catch` block is executed, the actual exception object is passed as an argument to the `catch` block. The code within a `catch` block should do whatever is necessary to handle the exceptional condition.

The `finally` clause of a `try` statement is always executed, no matter how control leaves the `try` statement. Thus it is a good place to handle clean-up operations, such as closing files, freeing resources, and closing network connections.

PREVIOUS
Synchronizing Multiple Threads

HOME
BOOK INDEX

NEXT
Declaring Exceptions

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5. Collections

Contents:

Java provides a number of utility classes that help you to manage a collection of objects. These collection classes allow you to work with objects without regard to their types, so they can be extremely useful for managing objects at a high level of abstraction. This chapter describes the following collection classes:

- The `java.util.Vector` class, which represents a dynamic array of objects.
- The `java.util.Stack` class, which represents a dynamic stack of objects.
- The `java.util.Dictionary` class, which is an `abstract` class that manages a collection of objects by associating a key with each object.
- The `java.util.Hashtable` class, which is a subclass of `java.util.Dictionary` that implements a specific algorithm to associate keys with objects. Given a key, a `Hashtable` can retrieve the associated object with little or no searching.
- The `java.util.Enumeration` interface, which supports sequential access to a set of elements.

5.1 Enumerations

The `Enumeration` interface is implemented by classes that provide serial access to a set of elements, or objects, in a collection. An object that implements the `Enumeration` interface provides two methods for dealing with the set: `nextElement()` and `hasMoreElements()`. The `nextElement()` method returns a value of type `Object`, so it can be used with any kind of collection. When you remove an object from an `Enumeration`, you may need to cast the object to the appropriate type before using it. You can iterate through the elements in an `Enumeration` only once; there is no way to reset it to the beginning or move backwards through the elements.

Here is an example that prints the contents of an object the implements the `Enumeration` interface:

```
static void printEnumeration(Enumeration e) {
    while (e.hasMoreElements()) {
        System.out.println(e.nextElement());
    }
```

Note that the above method is able to print all of the objects in the `Enumeration` without knowing their class types because the `println()` method handles objects of any type.

A number of classes in the Java API provide a method that returns a reference to an `Enumeration` object, rather than implementing the `Enumeration` interface directly. For example, as you'll see shortly, the `Vector` class provides an `elements()` method that returns an `Enumeration` of the objects in a `Vector` object.

PREVIOUS Generating Exceptions | HOME BOOK INDEX | NEXT Vectors

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

JAVA Fundamental Classes Reference

6. I/O

Contents:

The `java.io` package contains the fundamental classes for performing input and output operations in Java. These I/O classes can be divided into four basic groups:

- Classes for reading input from a stream.
- Classes for writing output to a stream.
- Classes for manipulating files.
- Classes for serializing objects.

All fundamental I/O in Java is based on *streams*. A stream represents a flow of data, or a channel of communication. Conceptually, there is a reading process at one end of the stream and a writing process at the other end. Java 1.0 supported only byte streams, which meant that Unicode characters were not always handled correctly. As of Java 1.1, there are classes in `java.io` for both byte streams and character streams. The character stream classes, which are called *readers* and *writers*, handle Unicode characters appropriately.

The rest of this chapter describes the classes in `java.io` that read from and write to streams, as well as the classes that manipulate files. The classes for serializing objects are described in Chapter 7, *Object Serialization*.

6.1 Input Streams and Readers

The `InputStream` class is an `abstract` class that defines methods to read sequentially from a stream of bytes. Java provides subclasses of the `InputStream` class for reading from files, `StringBuffer` objects, and byte arrays, among other things. Other subclasses of `InputStream` can be chained together to provide additional logic, such as keeping track of the current line number or combining multiple input sources into one logical input stream. It is also easy to define a subclass of `InputStream` that reads from any other kind of data source.

In Java 1.1, the `Reader` class is an `abstract` class that defines methods to read sequentially from a stream of characters. Many of the byte-oriented `InputStream` subclasses have character-based `Reader` counterparts. Thus, there are subclasses of `Reader` for reading from files, character arrays, and `String` objects.

InputStream

The `InputStream` class is the `abstract` superclass of all other byte input stream classes. It defines three `read()` methods for reading from a raw stream of bytes:

```
read()
read(byte[] b)
read(byte[] b, int off, int len)
```

If there is no data available to read, these methods block until input is available. The class also defines an `available()` method that returns the number of bytes that can be read without blocking and a `skip()` method that skips ahead a specified number of bytes. The `InputStream` class defines a mechanism for marking a position in the stream and returning to it later, via the `mark()` and `reset()` methods. The `markSupported()` method returns `true` in subclasses that support these methods.

Because the `InputStream` class is `abstract`, you cannot create a "pure" `InputStream`. However, the various subclasses of `InputStream` can be used interchangeably. For example, methods often take an `InputStream` as a parameter. Such a method accepts any subclass of `InputStream` as an argument.

`InputStream` is designed so that `read(byte[])` and `read(byte[], int, int)` both call `read()`. Thus, when you subclass `InputStream`, you only need to define the `read()` method. However, for efficiency's sake, you should also override `read(byte[], int, int)` with a method that can read a block of data more efficiently than reading each byte separately.

Reader

The `Reader` class is the `abstract` superclass of all other character input stream classes. It defines nearly the same methods as `InputStream`, except that the `read()` methods deal with characters instead of bytes:

```
read()
read(char[] cbuf)
read(char[] cbuf, int off, int len)
```

The `available()` method of `InputStream` has been replaced by the `ready()` method of `Reader`, which simply returns a flag that indicates whether or not the stream must block to read the next character.

`Reader` is designed so that `read()` and `read(char[])` both call `read(char[], int, int)`. Thus, when you subclass `Reader`, you only need to define the `read(char[], int, int)` method. Note that this design is different from, and more efficient than, that of `InputStream`.

InputStreamReader

The `InputStreamReader` class serves as a bridge between `InputStream` objects and `Reader` objects. Although an `InputStreamReader` acts like a character stream, it gets its input from an underlying byte stream and uses a character encoding scheme to translate bytes into characters. When you create an `InputStreamReader`, specify the underlying `InputStream` and, optionally, the name of an encoding scheme. For example, the following code fragment creates an `InputStreamReader` that reads characters from a file that is encoded using the ISO 8859-5 encoding:

```
String fileName = "encodedfile.txt"; String encodingName = "8859_5";
```

```
InputStreamReader in;
try {
   x FileInputStream fileIn = new FileInputStream(fileName);
    in = new InputStreamReader(fileIn, encodingName);
} catch (UnsupportedEncodingException e1) {
    System.out.println(encodingName + " is not a supported encoding scheme.");
} catch (IOException e2) {
    System.out.println("The file " + fileName + " could not be opened.");
}
```

FileInputStream and FileReader

The `FileInputStream` class is a subclass of `InputStream` that allows a stream of bytes to be read from a file. The `FileInputStream` class has no explicit open method. Instead, the file is implicitly opened, if appropriate, when the `FileInputStream` is created. There are three ways to create a `FileInputStream`:

- You can create a `FileInputStream` by passing the name of a file to be read:

  ```
  FileInputStream f1 = new FileInputStream("foo.txt");
  ```

- You can create a `FileInputStream` with a `File` object:

  ```
  File f = new File("foo.txt");
  FileInputStream f2 = new FileInputStream(f);
  ```

- You can create a `FileInputStream` with a `FileDescriptor` object. A `FileDescriptor` object encapsulates the native operating system's representation of an open file. You can get a `FileDescriptor` from a `RandomAccessFile` by calling its `getFD()` method. You create a `FileInputStream` that reads from the open file associated with a `RandomAccessFile` as follows:

  ```
  RandomAccessFile raf;
  raf = new RandomAccessFile("z.txt","r");
  FileInputStream f3 = new FileInputStream(raf.getFD());
  ```

  The `FileReader` class is a subclass of `Reader` that reads a stream of characters from a file. The bytes in the file are converted to characters using the default character encoding scheme. If you do not want to use the default encoding scheme, you need to wrap an `InputStreamReader` around a `FileInputStream`, as shown above. You can create a `FileReader` from a filename, a `File` object, or a `FileDescriptor` object, as described above for `FileInputStream`.

StringReader and StringBufferInputStream

The `StringReader` class is a subclass of `Reader` that gets its input from a `String` object. The `StringReader` class supports mark-and-reset functionality via the `mark()` and `reset()` methods. The following example shows the use of `StringReader`:

```
StringReader sr = new StringReader("abcdefg");
try {
    char[] buffer = new char[3];
    sr.read(buffer);
```

```
    System.out.println(buffer);
} catch (IOException e) {
    System.out.println("There was an error while reading.");
}
```

This code fragment produces the following output:

```
abc
```

The `StringBufferInputStream` class is the byte-based relative of `StringReader`. The entire class is deprecated as of Java 1.1 because it does not properly convert the characters of the string to a byte stream; it simply chops off the high eight bits of each character. Although the `markSupported()` method of `StringBufferInputStream` returns `false`, the `reset()` method causes the next read operation to read from the beginning of the `String`.

CharArrayReader and ByteArrayInputStream

The `CharArrayReader` class is a subclass of `Reader` that reads a stream of characters from an array of characters. The `CharArrayReader` class supports mark-and-reset functionality via the `mark()` and `reset()` methods. You can create a `CharArrayReader` by passing a reference to a `char` array to a constructor like this:

```
char[] c;
...
CharArrayReader r;
r = new CharArrayReader(c);
```

You can also create a `CharArrayReader` that only reads from part of an array of characters by passing an offset and a length to the constructor. For example, to create a `CharArrayReader` that reads elements 5 through 24 of a `char` array you would write:

```
r = new CharArrayReader(c, 5, 20);
```

The `ByteArrayInputStream` class is just like `CharArrayReader`, except that it deals with bytes instead of characters. In Java 1.0, `ByteArrayInputStream` did not fully support `mark()` and `reset()`; in Java 1.1 these methods are completely supported.

PipedInputStream and PipedReader

The `PipedInputStream` class is a subclass of `InputStream` that facilitates communication between threads. Because it reads bytes written by a connected `PipedOutputStream`, a `PipedInputStream` must be connected to a `PipedOutputStream` to be useful. There are a few ways to connect a `PipedInputStream` to a `PipedOutputStream`. You can first create the `PipedOutputStream` and pass it to the `PipedInputStream` constructor like this:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream(po);
```

You can also create the `PipedInputStream` first and pass it to the `PipedOutputStream` constructor like this:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

The `PipedInputStream` and `PipedOutputStream` classes each have a `connect()` method you can use to explicitly connect a `PipedInputStream` and a `PipedOutputStream` as follows:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
pi.connect(po);
```

Or you can use `connect()` as follows:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
po.connect(pi);
```

Multiple `PipedOutputStream` objects can be connected to a single `PipedInputStream` at one time, but the results are unpredictable. If you connect a `PipedOutputStream` to an already connected `PipedInputStream`, any unread bytes from the previously connected `PipedOutputStream` are lost. Once the two `PipedOutputStream` objects are connected, the `PipedInputStream` reads bytes written by either `PipedOutputStream` in the order that it receives them. The scheduling of different threads may vary from one execution of the program to the next, so the order in which the `PipedInputStream` receives bytes from multiple `PipedOutputStream` objects can be inconsistent.

The `PipedReader` class is the character-based equivalent of `PipedInputStream`. It works in the same way, except that a `PipedReader` is connected to a `PipedWriter` to complete the pipe, using either the appropriate constructor or the `connect()` method.

FilterInputStream and FilterReader

The `FilterInputStream` class is a wrapper class for `InputStream` objects. Conceptually, an object that belongs to a subclass of `FilterInputStream` is wrapped around another `InputStream` object. The constructor for this class requires an `InputStream`. The constructor sets the object's `in` instance variable to reference the specified `InputStream`, so from that point on, the `FilterInputStream` is associated with the given `InputStream`. All of the methods in `FilterInputStream` work by calling the corresponding methods in the underlying `InputStream`. Because the `close()` method of a `FilterInputStream` calls the `close()` method of the `InputStream` that it wraps, you do not need to explicitly close the underlying `InputStream`.

A `FilterInputStream` does not add any functionality to the object that it wraps, so by itself it is not very useful. However, subclasses of the `FilterInputStream` class do add functionality to the objects that they wrap in two ways:

- Some subclasses add logic to the `InputStream` methods. For example, the `InflaterInputStream` class in the `java.util.zip` package decompresses data automatically in the `read()` methods.

- Some subclasses add new methods. An example is `DataInputStream`, which provides methods for reading primitive Java data types from the stream.

The `FilterReader` class is the character-based equivalent of `FilterInputStream`. A `FilterReader` is wrapped around an underlying `Reader` object; the methods of `FilterReader` call the corresponding methods of the underlying Reader. However, unlike FilterInputStream, FilterReader is an abstract class, so you cannot instantiate it directly.

DataInputStream

The DataInputStream class is a subclass of FilterInputStream that provides methods for reading a variety of data types. The DataInputStream class implements the DataInput interface, so it defines methods for reading all of the primitive Java data types.

You create a DataInputStream by passing a reference to an underlying InputStream to the constructor. Here is an example that creates a DataInputStream and uses it to read an int that represents the length of an array and then to read the array of long values:

```
long[] readLongArray(InputStream in) throws IOException {
    DataInputStream din = new DataInputStream(in);
    int count = din.readInt();
    long[] a = new long[count];
    for (int i = 0; i < count; i++) {
        a[i] = din.readLong();
    }
    return a;
}
```

BufferedReader and BufferedInputStream

The BufferedReader class is a subclass of Reader that buffers input from an underlying Reader. A BufferedReader object reads enough characters from its underlying Reader to fill a relatively large buffer, and then it satisfies read operations by supplying characters that are already in the buffer. If most read operations read just a few characters, using a BufferedReader can improve performance because it reduces the number of read operations that the program asks the operating system to perform. There is generally a measurable overhead associated with each call to the operating system, so reducing the number of calls into the operating system improves performance. The BufferedReader class supports mark-and-reset functionality via the mark() and reset() methods.

Here is an example that shows how to create a BufferedReader to improve the efficiency of reading from a file:

```
try {
    FileReader fileIn = new FileReader("data.dat");
    BufferedReader in = new BufferedReader(fileIn);
    // read from the file
} catch (IOException e) {
    System.out.println(e);
}
```

The BufferedInputStream class is the byte-based counterpart of BufferedReader. It works in the same way as BufferedReader, except that it buffers input from an underlying InputStream.

LineNumberReader and LineNumberInputStream

The LineNumberReader class is a subclass of BufferedReader. Its read() methods contain additional logic to count end-of-line characters and thereby maintain a line number. Since different platforms use different characters to represent the end of a line, `LineNumberReader` takes a flexible approach and recognizes `"\n"`, `"\r"`, or `"\r\n"` as the end of a line. Regardless of the end-of-line character it reads, `LineNumberReader` returns only `"\n"` from its `read()` methods.

You can create a `LineNumberReader` by passing its constructor a `Reader`. The following example prints out the first five lines of a file, with each line prefixed by its number. If you try this example, you'll see that the line numbers begin at `0` by default:

```
try {
    FileReader fileIn = new FileReader("text.txt");
    LineNumberReader in = new LineNumberReader(fileIn);
    for (int i = 0; i < 5; i++)
        System.out.println(in.getLineNumber() + " " + in.readLine());
}catch (IOException e) {
    System.out.println(e);
}
```

The `LineNumberReader` class has two methods pertaining to line numbers. The `getLineNumber()` method returns the current line number. If you want to change the current line number of a `LineNumberReader`, use `setLineNumber()`. This method does not affect the stream position; it merely sets the value of the line number.

The `LineNumberInputStream` is the byte-based equivalent of `LineNumberReader`. The entire class is deprecated in Java 1.1 because it does not convert bytes to characters properly. Apart from the conversion problem, `LineNumberInputStream` works the same as `LineNumberReader`, except that it takes its input from an `InputStream` instead of a `Reader`.

SequenceInputStream

The `SequenceInputStream` class is used to sequence together multiple `InputStream` objects. Consider this example:

```
FileInputStream f1 = new FileInputStream("data1.dat");
FileInputStream f2 = new FileInputStream("data2.dat");
SequenceInputStream s = new SequenceInputStream(f1, f2);
```

This example creates a `SequenceInputStream` that reads all of the bytes from `f1` and then reads all of the bytes from `f2` before reporting that it has encountered the end of the stream. You can also cascade `SequenceInputStream` object themselves, to allow more than two input streams to be read as if they were one. You would write it like this:

```
FileInputStream f3 = new FileInputStream("data3.dat");
SequenceInputStream s2 = new SequenceInputStream(s, f3);
```

The `SequenceInputStream` class has one other constructor that may be more appropriate for wrapping more than two `InputStream` objects together. It takes an `Enumeration` of `InputStream` objects as its argument. The following example shows how to create a `SequenceInputStream` in this manner:

```
Vector v = new Vector();
v.add(new FileInputStream("data1.dat"));
v.add(new FileInputStream("data2.dat"));
```

```
v.add(new FileInputStream("data3.dat"));
Enumeration e = v.elements();
SequenceInputStream s = new SequenceInputStream(e);
```

PushbackInputStream and PushbackReader

The `PushbackInputStream` class is a `FilterInputStream` that allows data to be pushed back into the input stream and reread by the next read operation. This functionality is useful for implementing things like parsers that need to read data and then return it to the input stream. The Java 1.0 version of `PushbackInputStream` supported only a one-byte pushback buffer; in Java 1.1 this class has been enhanced to support a larger pushback buffer.

To create a `PushbackInputStream`, pass an `InputStream` to its constructor like this:

```
FileInputStream ef = new FileInputStream("expr.txt");
PushbackInputStream pb = new PushbackInputStream(ef);
```

This constructor creates a `PushbackInputStream` that uses a default one-byte pushback buffer. When you have data that you want to push back into the input stream to be read by the next read operation, you pass the data to one of the `unread()` methods.

The `PushbackReader` class is the character-based equivalent of `PushbackInputStream`. In the following example, we create a `PushbackReader` with a pushback buffer of 48 characters:

```
FileReader fileIn = new FileReader("expr.txt");
PushbackReader in = new PushbackReader(fileIn, 48);
```

Here is an example that shows the use of a `PushbackReader`:

```
public String readDigits(PushbackReader pb) {
    char c;
    StringBuffer buffer = new StringBuffer();
    try {
        while (true) {
            c = (char)pb.read();
            if (!Character.isDigit(c))
                break;
            buffer.append(c);
        }
        if (c != -1)
            pb.unread(c);
    }catch (IOException e) {}
    return buffer.toString();
}
```

The above example shows a method that reads characters corresponding to digits from a `PushbackReader`. When it reads a character that is not a digit, it calls the `unread()` method so that the nondigit can be read by the next read operation. It then returns a string that contains the digits that were read.

7. Object Serialization

Contents:

The object serialization mechanism in Java 1.1 provides a way for objects to be written as a stream of bytes and then later recreated from that stream of bytes. This facility supports a variety of interesting applications. For example, object serialization provides persistent storage for objects, whereby objects are stored in a file for later use. Also, a copy of an object can be sent through a socket to another Java program. Object serialization forms the basis for the remote method invocation mechanism in Java that facilitates distributed programs. Object serialization is supported by a number of new classes in the `java.io` package in Java 1.1.

7.1 Object Serialization Basics

If a class is designed to work with object serialization, reading and writing instances of that class is quite simple. The process of writing an object to a byte stream is called *serialization*. For example, here is how you can write a `Color` object to a file:

```
FileOutputStream out = new FileOutputStream("tmp");
ObjectOutput objOut = new ObjectOutputStream(out);
objOut.writeObject(Color.red);
```

All you need to do is create an `ObjectOutputStream` around another output stream and then pass the object to be written to the `writeObject()` method. If you are writing objects to a socket or any other destination that is time-sensitive, you should call the `flush()` method after you are finished passing objects to the `ObjectOutputStream`.

The process of reading an object from byte stream is called *deserialization*. Here is how you can read that `Color` object from its file:

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream objIn = new ObjectInputStream(in);
Color c = (Color)objIn.readObject();
```

Here all you need to do is create an `ObjectInputStream` object around another input stream and call its `readObject()` method.

PREVIOUS File Manipulation | HOME BOOK INDEX | NEXT Writing Classes to Work with Serialization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8. Networking

Contents:

The `java.net` package provides two basic mechanisms for accessing data and other resources over a network. The fundamental mechanism is called a socket. A socket allows programs to exchange groups of bytes called packets. There are a number of classes in `java.net` that support sockets, including `Socket`, `ServerSocket`, `DatagramSocket`, `DatagramPacket`, and `MulticastSocket`. The `java.net` package also includes a `URL` class that provides a higher-level mechanism for accessing and processing data over a network.

8.1 Sockets

A socket is a mechanism that allows programs to send packets of bytes to each other. The programs do not need to be running on the same machine, but if they are running on different machines, they do need to be connected to a network that allows the machines to exchange data. Java's socket implementation is based on the socket library that was originally part of BSD UNIX. Programmers who are familiar with UNIX sockets or the Microsoft WinSock library should be able to see the similarities in the Java implementation.

When a program creates a socket, an identifying number called a port number is associated with the socket. Depending on how the socket is used, the port number is either specified by the program or assigned by the operating system. When a socket sends a packet, the packet is accompanied by two pieces of information that specify the destination of the packet:

- A network address that specifies the system that should receive the packet.
- A port number that tells the receiving system to which socket to deliver the data.

Sockets typically work in pairs, where one socket acts as a client and the other functions as a server. A server socket specifies the port number for the network communication and then listens for data that is sent to it by client sockets. The port numbers for server sockets are well-known numbers that are known to client programs. For example, an FTP server uses a socket that listens at port 21. If a client program wants to communicate with an FTP server, it knows to contact a socket that listens at port 21.

The operating system normally specifies port numbers for client sockets because the choice of a port number is not usually important. When a client socket sends a packet to a server socket, the packet is accompanied by the port number of the client socket and the client's network address. The server is then able to use that information to respond to the client.

When using sockets, you have to decide which type of protocol that you want it to use to transport packets over the network: a connection-oriented protocol or a connectionless protocol. With a connection-oriented protocol, a client socket establishes a connection to a server socket when it is created. Once the connection has been established, a connection-oriented protocol ensures that data is delivered reliably, which means:

- For every packet that is sent, the packet is delivered. Every time a socket sends a packet, it expects to receive an acknowledgement that the packet has been received successfully. If the socket does not receive that acknowledgement within the time it expects to receive it, the socket sends the packet again. The socket keeps trying until transmission is successful, or it decides that delivery has become impossible.

- Packets are read from the receiving socket in the same order that they were sent. Because of the way that networks work, packets may arrive at the receiving socket in a different order than they were sent. A reliable, connection-oriented protocol allows the receiving socket to reorder the packets it receives, so that they can be read by the receiving program in the same order that they were sent.

A connectionless protocol allows a best-effort delivery of packets. It does not guarantee that packets are delivered or that packets are read by the receiving program in the same order they were sent. A connectionless protocol trades these deficiencies for performance advantages over connection-oriented protocols. Here are two types of situations in which connectionless protocols are frequently preferred over connection-oriented protocols:

- When only a single packet needs to be sent and guaranteed delivery is not crucial, a connectionless protocol eliminates the overhead involved in creating and destroying a connection. For comparison purposes, the connection-oriented TCP/IP protocol uses seven packets to send a single packet, while the connectionless UDP/IP protocol uses only one. A protocol for getting the current time typically uses a connectionless protocol to request the current time from the server and to return the time to the requester.

- For extremely time-sensitive applications, such as sending audio in real time, the guarantee of reliable transmission is not an advantage and may be a disadvantage. Pausing until a missing piece of data is received can cause noticeable clicks or pauses in the audio. Techniques for sending audio over a network that use a connectionless protocol have been developed and they work noticeably better. For example, RealAudio uses a protocol that runs on top of a connectionless protocol to transmit sound over a network.

Table 8.1 shows the roles of the various socket classes in the `java.net` package.

Table 8.1: Socket Classes in java.net

	Client	**Server**
Connection-oriented Protocol	`Socket`	`ServerSocket`
Connectionless Protocol	`DatagramSocket`	`DatagramSocket`

As of Java 1.1, the `java.net` package also contains a `MulticastSocket` class that supports connectionless, multicast data communication.

Sockets for Connection-Oriented Protocols

When you are writing code that implements the server side of a connection-oriented protocol, your code typically follows this pattern:

- Create a `ServerSocket` object to accept connections.
- When the `ServerSocket` accepts a connection, it creates a `Socket` object that encapsulates the connection.
- The `Socket` is asked to create `InputStream` and `OutputStream` objects that read and write bytes to and from the connection.
- The `ServerSocket` can optionally create a new thread for each connection, so that the server can listen for new connections while it is communicating with clients.

The code that implements the client side of a connection-oriented protocol is quite simple. It creates a `Socket` object that opens a connection with a server, and then it uses that `Socket` object to communicate with the server.

Now let's look at an example. The example consists of a pair of programs that allows a client to get the contents of a file from a server. The client requests the contents of a file by opening a connection to the server and sending it the name of a file followed by a newline character. If the server is able to read the named file, it responds by sending the string `"Good:\n"` followed by the contents of the file. If the server is not able to read the named file, it responds by sending the string `"Bad:"` followed by the name of the file and a newline character. After the server has sent its response, it closes the connection.

Here's the program that implements the server side of this file transfer:

```
public class FileServer extends Thread {
    public static void main(String[] argv) {
        ServerSocket s;
        try {
            s = new ServerSocket(1234, 10);
        }catch (IOException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            return;
        }
        try {
            while (true) {
                new FileServer(s.accept());
            }
        }catch (IOException e) {
        }
    }
    private Socket socket;
    FileServer(Socket s) {
        socket = s;
        start();
    }
    public void run() {
        InputStream in;
        String fileName = "";
        PrintStream out = null;
        FileInputStream f;
        try {
            in = socket.getInputStream();
            out = new PrintStream(socket.getOutputStream());
            fileName = new DataInputStream(in).readLine();
            f = new FileInputStream(fileName);
        }catch (IOException e) {
            if (out != null)
              out.print("Bad:"+fileName+"\n");
            out.close();
            try {
                socket.close();
            }catch (IOException ie) {
            }
            return;
        }
        out.print("Good:\n");
        // send contents of file to client.
        byte[] buffer = new byte[4096];
        try {
            int len;
            while ((len = f.read(buffer)) > 0) {
                out.write(buffer, 0, len);
            }// while
        }catch (IOException e) {
        }finally {
            try {
                in.close();
                out.close();
                socket.close();
            }catch (IOException e) {
            }
        }
    }
}
```

The `FileServer` class implements the server side of the file transfer; it is a subclass of `Thread` to make it easier to write code that can handle multiple connections at the same time. The `main()` method provides the top-level logic for the program. The first thing that `main()` does is to create a `ServerSocket` object to listen for connections. The constructor for `ServerSocket` takes two parameters: the port number for the socket and a value that specifies the maximum length of the pending connections queue. The operating system can accept connections on behalf of the socket when the server program is busy doing something other than accepting connections. If the second parameter is greater than zero, the operating system can accept up to that many connections on behalf of the socket and store them in a queue. If the second parameter is zero, however, the operating system does not accept any connections on behalf of the server program. The remainder of the `main()` method accepts a connection, creates a new instance of the `FileServer` class to process the connection, and then waits for the next connection.

Each `FileServer` object is responsible for handling a connection accepted by its `main()` method. A `FileServer` object uses a `private` variable, `socket`, to refer to the `Socket` object that allows it to communicate with the client program on the other end of the connection. The constructor for `FileServer` sets its `socket` variable to refer to the `Socket` object that is passed to it by the `main()` method and then calls its `start()` method. The `FileServer` class inherits the `start()` method from the `Thread` class; the `start()` method starts a new thread that calls the `run()` method. Because the rest of the connection processing is done asynchronously in a separate thread, the constructor can return immediately. This allows the `main()` method to accept another connection right away, instead of having to wait for this connection to be fully processed before accepting another.

The `run()` method uses the `in` and `out` variables to refer to `InputStream` and `PrintStream` objects that read from and write to the connection associated with the `Socket` object, respectively. These streams are created by calling the `getInputStream()` and `getOutputStream()` methods of the `Socket` object. The `run()` method then reads the name of the file that the client program wants to receive and creates a `FileInputStream` to read that file. If any of the methods called up to this point have detected a problem, they throw some kind of `IOException`. In this case, the server sends a response to the client that consists of the string `"Bad:"` followed by the filename and then closes the socket and returns, which kills the thread.

If everything up to this point has been fine, the server sends the string `"Good:"` and then copies the contents of the file to the socket. The copying is done by repeatedly filling a buffer with bytes from the file and writing the buffer to the socket. When the contents of the file are exhausted, the streams and the socket are closed, the `run()` method returns, and the thread dies.

Now let's take a look at the client part of this program:

```
public class FileClient {
    private static boolean usageOk(String[] argv) {
        if (argv.length != 2) {
            String msg = "usage is: " + "FileClient server-name file-name";
            System.out.println(msg);
            return false;
        }
        return true;
    }
    public static void main(String[] argv) {
        int exitCode = 0;
        if (!usageOk(argv))
          return;
        Socket s = null;
        try {
            s = new Socket(argv[0], 1234);
        }catch (IOException e) {
            String msg = "Unable to connect to server";
            System.err.println(msg);
            e.printStackTrace();
            System.exit(1);
        }
        InputStream in = null;
        try {
            OutputStream out = s.getOutputStream();
            new PrintStream(out).print(argv[1]+"\n");
            in = s.getInputStream();
            DataInputStream din = new DataInputStream(in);
            String serverStatus = din.readLine();
            if (serverStatus.startsWith("Bad")) {
                exitCode = 1;
            int ch;
            while((ch = in.read()) >= 0) {
                System.out.write((char)ch);
            }// while
        }catch (IOException e) {
        }finally {
            try {
                s.close();
            }catch (IOException e) {
            }
        }
    }
}
```

The `usageOk()` method is simply a utility method that verifies that the correct number of arguments have been passed to the client application. It outputs a help message if the number of arguments is not what is expected. It is generally a good idea to include a method like this in a Java application that uses command-line parameters.

The `main()` method does the real work of `FileClient`. After it verifies that it has the correct number of parameters, it attempts to create a socket connected to the server program running on the specified host and listening for connections on port number 1234. The socket that it creates is encapsulated by a `Socket` object. The constructor for the `Socket` object takes two arguments: the name of the machine the server program is running on and the port number. After the socket is successfully opened, the client sends the specified filename, followed by a new line character, to the server. The client then gets an `InputStream` from the socket to read what the server is sending and reads the success/failure code that the server sends back. If the request is a success, the client reads the contents of the requested file.

Note that the `finally` clause at the end closes the socket. If the program did not explicitly close the socket, it would be closed automatically when the program terminates. However, it is a good programming practice to explicitly close a socket when you are done with it.

Sockets for Connectionless Protocols

Communicating with a connectionless protocol is simpler than using a connection-oriented protocol, as both the client and the server use `DatagramSocket` objects. The code for the server-side program has the following pattern:

- Create a `DatagramSocket` object associated with a specified port number.
- Create a `DatagramPacket` object and ask the `DatagramSocket` to put the next piece of data it receives in the `DatagramPacket`.

On the client-side, the order is simply reversed:

- Create a `DatagramPacket` object associated with a piece of data, a destination network address, and a port number.
- Ask a `DatagramSocket` object to send the data associated with the `DatagramPacket` to the destination associated with the `DatagramSocket`.

Let's look at an example that shows how this pattern can be coded into a server that provides the current time and a client that requests the current time. Here's the code for the server class:

```
public class TimeServer {
    static DatagramSocket socket;
    public static void main(String[] argv) {
        try {
            socket = new DatagramSocket(7654);
        }catch (SocketException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            System.exit(1);
        }
        DatagramPacket datagram;
        datagram = new DatagramPacket(new byte[1], 1);
        while (true) {
            try {
                socket.receive(datagram);
                respond(datagram);
            }catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    static void respond(DatagramPacket request) {
        ByteArrayOutputStream bs;
        bs = new ByteArrayOutputStream();
        DataOutputStream ds = new DataOutputStream(bs);
        try {
            ds.writeLong(System.currentTimeMillis());
        }catch (IOException e) {
        }
        DatagramPacket response;
        byte[] data = bs.toByteArray();
        response = new DatagramPacket(data, data.length,
                       request.getAddress(), request.getPort());
        try {
            socket.send(response);
        }catch (IOException e) {
            // Give up, we've done our best.
        }
    }
}
```

The `main()` method of the `TimeServer` class begins by creating a `DatagramSocket` object that uses port number 7654. The `socket` variable refers to this `DatagramSocket`, which is used to communicate with clients. Then the `main()` method creates a `DatagramPacket` object to contain data received by the `DatagramSocket`. The two-argument constructor for `DatagramPacket` creates objects that receive data. The first argument is an array of bytes to contain the data, while the second argument specifies the number of bytes to read. When a `DatagramSocket` is asked to receive a packet into a `DatagramPacket`, only the specified number of bytes are read. Even though the client is not really sending any information to the server, we still create a `DatagramPacket` with a 1-byte buffer. In theory, all that the server needs is an empty packet that specifies the client's network address and port number, but attempting to receive a zero-byte packet does not work. When the `receive()` method of a `DatagramSocket` is called to receive a zero-byte packet, it returns immediately, rather than waiting for a packet to arrive. Finally, the server enters an infinite loop that receives requests from clients using the `receive()` method of the `DatagramSocket`, and sends responses.

The `respond()` method handles sending responses. It starts by writing the current time as a `long` value to an array of bytes. Next, the `respond()` method prepares to send the array of bytes by creating a `DatagramPacket` object that encapsulates the array and the address and port number of the client that requested the time. Notice that the constructor used to create a `DatagramPacket` object for sending a packet takes four arguments: an array of bytes, the number of bytes to send, the client's network address, and the client's port number. The address and port are retrieved from the request `DatagramPacket` with the `getAddress()` and `getPort()` methods. The `respond()` method finishes its work by actually sending the `DatagramPacket` using the `send()` method of the `DatagramSocket`.

Now here's the code for the corresponding client program:

```
public class TimeClient {
    private static boolean usageOk(String[] argv) {
        if (argv.length != 1) {
            String msg = "usage is: " + "TimeClient server-name";
            System.out.println(msg);
            return false;
        }
        return true;
    }
    public static void main(String[] argv) {
        if (!usageOk(argv))
            System.exit(1);
        DatagramSocket socket;
        try {
            socket = new DatagramSocket();
        }catch (SocketException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            System.exit(1);
            return;
        }
        long time;
        try {
            byte[] buf = new byte[1];
            socket.send(new DatagramPacket(buf, 1,
                            InetAddress.getByName(argv[0]), 7654));
            DatagramPacket response = new DatagramPacket(new byte[8],8);
            socket.receive(response);
            ByteArrayInputStream bs;
            bs = new ByteArrayInputStream(response.getData());
            DataInputStream ds = new DataInputStream(bs);
            time = ds.readLong();
        }catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
            return;
        }
        System.out.println(new Date(time));
        socket.close();
    }
}
```

The `main()` method does the real work of `TimeClient`. After it verifies that it has the correct number of parameters with `usageOk()`, it creates a `DatagramSocket` object for communicating with the server. Note that the constructor for this `DatagramSocket` does not specify any parameters; a client `DatagramSocket` is not explicitly connected to a specific port. Then the `main()` method creates a `DatagramPacket` object to contain the request to be sent to the server. Since this `DatagramPacket` is being used to send a packet, the code uses the four-argument constructor that specifies an array of bytes, the number of bytes to send, the specified network address for a time server, and the server's port number. The `DatagramPacket` is then sent to the server with the `send()` method of the `DatagramSocket`.

Now the `main()` method creates another `DatagramPacket` to receive the response from the server. The two-argument constructor is used this time because the object is being created to receive data. After calling the `receive()` method of the `DatagramSocket` to get the response from the server, the `main()` method gets the data from the response `DatagramPacket` by calling `getData()`. The data is wrapped in a `DataInputStream` so that the data can be read as a `long` value. If everything has gone smoothly, the client finishes by printing the current time and closing the socket.

PREVIOUS Versioning of Classes | HOME BOOK INDEX | NEXT URL Objects

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9. Security

Contents:

Java uses a "sandbox" security model to ensure that applets cannot cause security problems. The idea is that an applet can do whatever it wants within the constraints of its sandbox, but that nothing done inside the sandbox has any consequences outside of the sandbox.

9.1 SecurityManager

Java implements the sandbox model using the `java.lang.SecurityManager` class. An instance of `SecurityManager` is passed to the method `System.setSecurityManager()` to establish the security policy for an application. Before `setSecurityManager()` is called, a Java program can access any resources available on the system. After `setSecurityManager()` is called, however, the `SecurityManager` object is responsible for providing a security policy. Once a security policy has been set by calling `setSecurityManager`, the method cannot be called again. Subsequent calls simply throw a `SecurityException`.

All methods in the Java API that can access resources outside of the Java environment call a `SecurityManager` method to ask permission before doing anything. If the `SecurityManager` method throws a `SecurityException`, the exception is thrown out of the calling method, and access to the resource is denied. The `SecurityManager` class defines a number of methods for asking for permission to access specific resources. Each of these methods has a name that begins with the word "check." Table 9.1 shows the names of the `check` methods provided by the `SecurityManager` class.

Table 9.1: The Check Methods of SecurityManager

Method Name	Permission
`checkAccept()`	To accept a network connection

`checkAccess()`	To modify a `Thread` or `ThreadGroup`
`checkAwtEventQueueAccess()`	To access the AWT event queue
`checkConnect()`	To establish a network connection or send a datagram
`checkCreateClassLoader()`	To create a `ClassLoader` object
`checkDelete()`	To delete a file
`checkExec()`	To call an external program
`checkExit()`	To stop the Java virtual machine and exit the Java environment
`checkLink()`	To dynamically link an external library into the Java environment
`checkListen()`	To listen for a network connection
`checkMemberAccess()`	To access the members of a class
`checkMulticast()`	To use a multicast connection
`checkPackageAccess()`	To access the classes in a package
`checkPackageDefinition()`	To define classes in a package
`checkPrintJobAccess()`	To initiate a print job request
`checkPropertiesAccess()`	To get or set the `Properties` object that defines all of the system properties
`checkPropertyAccess()`	To get or set a system property
`checkRead()`	To read from a file or input stream
`checkSecurityAccess()`	To perform a security action
`checkSetFactory()`	To set a factory class that determines classes to be used for managing network connections and their content
`checkSystemClipboardAccess()`	To access the system clipboard
`checkTopLevelWindow()`	To create a top-level window on the screen
`checkWrite()`	To write to a file or output stream

The `SecurityManager` class provides implementations of these methods that always refuse the requested permission. To implement a more permissive security policy, you need to create a subclass of `SecurityManager` that implements that policy.

In Java 1.0, most browsers consider an applet to be trusted or untrusted. An untrusted applet is one that does not come from the local filesystem. An untrusted applet is treated as follows by most popular browsers:

- It can establish network connections to the network address from which it came.
- It can create new windows on the screen. However, a notice is displayed on the bottom of the

window that the window was created by an untrusted applet.

- It cannot access any other external resources. In particular, untrusted applets cannot access local files.

As of Java 1.1, an applet can have a digital signature attached to it. When an applet has been signed by a trusted entity, a browser may consider the applet to be trusted and relax its security policy.

PREVIOUS URL Objects | HOME BOOK INDEX | NEXT ClassLoader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10. Accessing the Environment

Contents:

The `java.lang.System` and `java.lang.Runtime` classes provide a variety of methods that allow a Java program to access information and resources for the environment in which it is running. This environment includes the Java virtual machine and the native operating system.

10.1 I/O

The `System` class defines three `static` variables for the three default I/O stream objects that are used by Java programs:

`in`

This variable refers to an `InputStream` that is associated with the process's standard input.

`out`

This variable refers to a `PrintStream` object that is associated with the process's standard output. In an applet environment, the `PrintStream` is likely to be associated with a separate window or a file, although this is not guaranteed.

This stream is the most commonly used of the three I/O streams provided by the `System` class. Even in GUI-based applications, sending output to this stream can be useful for debugging purposes. The usual idiom for sending output to this stream is:

```
    System.out.println("some string");
```

`err`

This variable refers to a `PrintStream` object that is associated with the process's standard error output. In an applet environment, the `PrintStream` is likely to be associated with a separate window or a file, although this is not guaranteed.

PREVIOUS HOME NEXT
ClassLoader BOOK INDEX System Properties

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11. The java.io Package

Contents:

The package `java.io` contains the classes that handle fundamental input and output operations in Java. The I/O classes can be grouped as follows:

- Classes for reading input from a stream of data.
- Classes for writing output to a stream of data.
- Classes that manipulate files on the local filesystem.
- Classes that handle object serialization.

I/O in Java is based on streams. A stream represents a flow of data or a channel of communication. Java 1.0 supports only byte streams. The `InputStream` class is the superclass of all of the Java 1.0 byte input streams, while `OutputStream` is the superclass of all the byte output streams. The drawback to these byte streams is that they do not always handle Unicode characters correctly.

As of Java 1.1, `java.io` contains classes that represent character streams. These character stream classes handle Unicode characters appropriately by using a character encoding to convert bytes to characters and vice versa. The `Reader` class is the superclass of all the Java 1.1 character input streams, while `Writer` is the superclass of all character output streams.

The `InputStreamReader` and `OutputStreamWriter` classes provide a bridge between byte streams and character streams. If you wrap an `InputStreamReader` around an `InputStream` object, the bytes in the byte stream are read and converted to characters using the character encoding scheme specified by the `InputStreamReader`. Likewise, you can wrap an `OutputStreamWriter` around any `OutputStream` object so that you can write characters and have them converted to bytes.

As of Java 1.1, `java.io` also contains classes to support object serialization. Object serialization is the ability to write the complete state of an object to an output stream, and then later recreate that object by reading in the serialized state from an input stream. The `ObjectOutputStream` and `ObjectInputStream` classes handle serializing and deserializing objects, respectively.

The `RandomAccessFile` class is the only class that does not use a stream for reading or writing data. As its name implies, `RandomAccessFile` provides nonsequential access to a file for both reading and writing purposes.

The `File` class represents a file on the local file system. The class provides methods to identify and retrieve information about a file.

Figure 11.1 shows the class hierarchy for the `java.io` package. The `java.io` package defines a number of standard I/O exception classes. These exception classes are all subclasses of `IOException`, as shown in Figure 11.2.

Figure 11.1: The java.io package

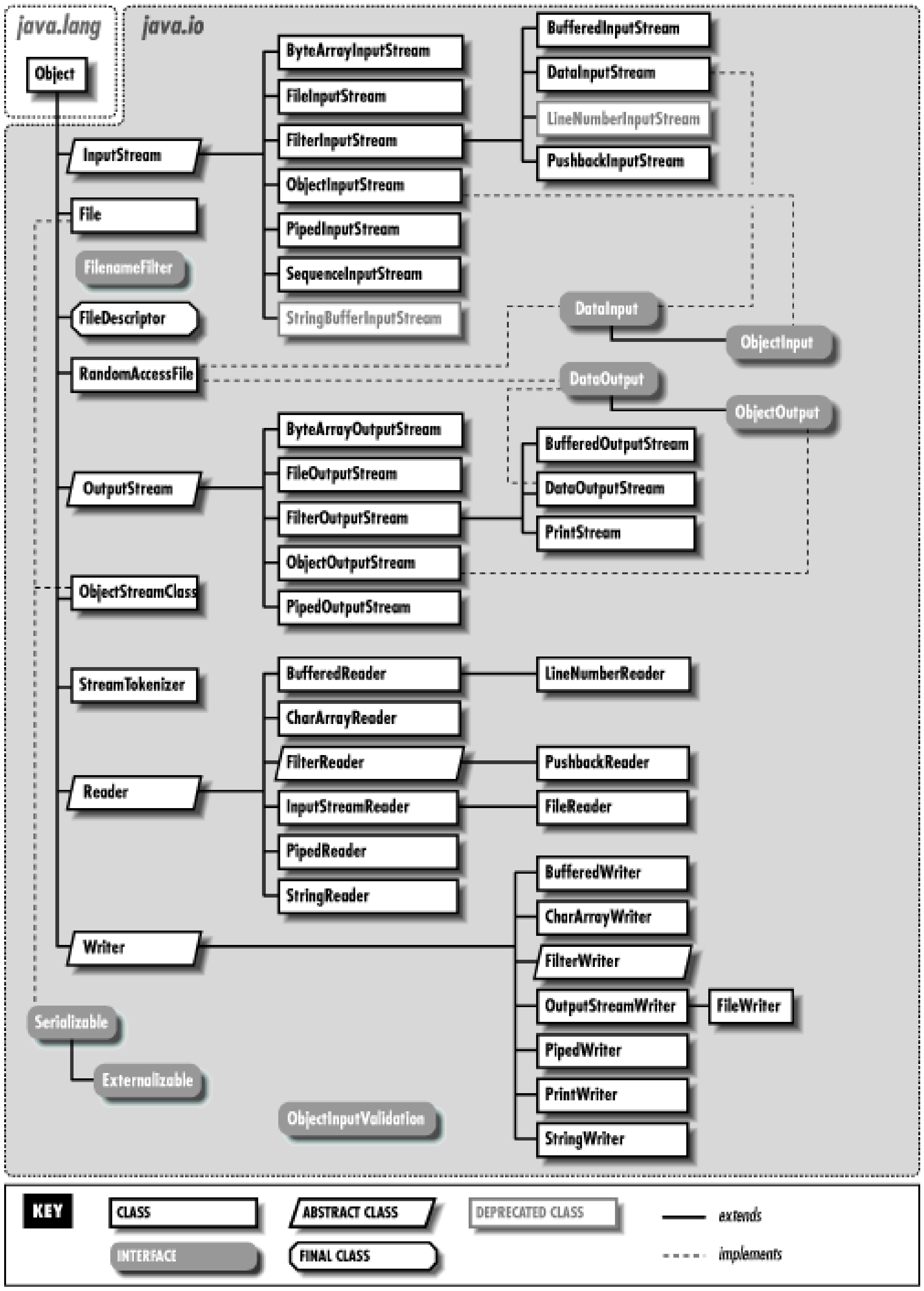
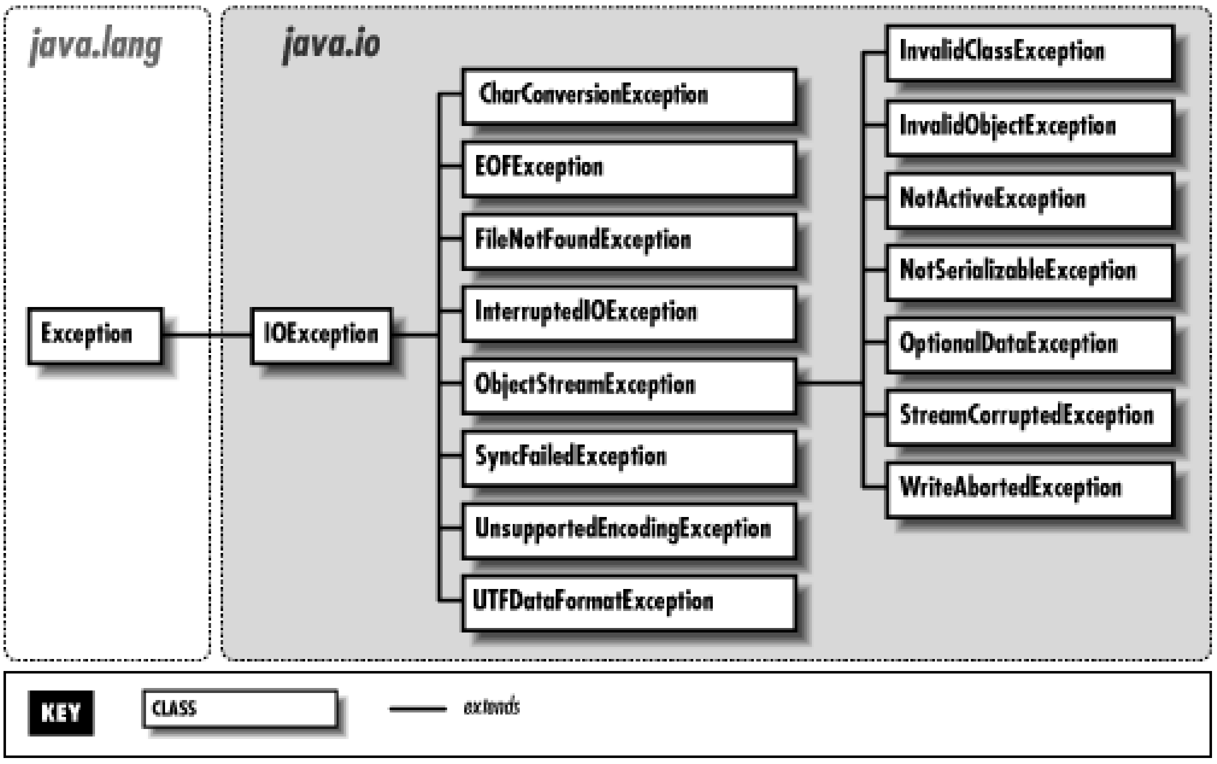


Figure 11.2: The exception classes in the java.io package



BufferedInputStream

Name

BufferedInputStream

Synopsis

Class Name:

    java.io.BufferedInputStream

Superclass:

    java.io.FilterInputStream

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `BufferedInputStream` object provides a more efficient way to read just a few bytes at a time from an `InputStream`. `BufferedInputStream` object use a buffer to store input from an associated `InputStream`. In other words, a large number of bytes are read from the underlying stream and stored in an internal buffer. A `BufferedInputStream` is more efficient than a regular `InputStream` because reading data from memory is faster than reading it from a disk or a network. All reading is done directly from the internal buffer; the disk or network needs to be accessed only occasionally to fill up the buffer.

You should wrap a `BufferedInputStream` around any `InputStream` whose `read()` operations may be time consuming or costly, such as a `FileInputStream`.

`BufferedInputStream` provides a way to mark a position in the stream and subsequently reset the stream to that position, using `mark()` and `reset()`.

Class Summary

```
public class java.io.BufferedInputStream extends java.io.FilterInputStream {
  // Variables
  protected byte[] buf;
  protected int count;
  protected int marklimit;
  protected int markpos;
  protected int pos;
  // Constructors
  public BufferedInputStream(InputStream in);
  public BufferedInputStream(InputStream in, int size);
  // Instance Methods
  public synchronized int available();
  public synchronized void mark(int readlimit);
  public boolean markSupported();
  public synchronized int read();
  public synchronized int read(byte[] b, int off, int len);
  public synchronized void reset();
  public synchronized long skip(long n);
}
```

Variables

buf

protected byte[] buf

Description

The buffer that stores the data from the input stream.

count

protected int count

Description

A placeholder that marks the end of valid data in the buffer.

marklimit

protected int marklimit

Description

The maximum number of bytes that can be read after a call to `mark()` before a call to `reset()` fails.

markpos

protected int markpos

Description

The position of the stream when `mark()` was called. If `mark()` has not been called, this variable is `-1`.

pos

protected int pos

Description

The current position in the buffer, or in other words, the index of the next character to be read.

Constructors

BufferedInputStream

public BufferedInputStream(InputStream in)

Parameters

`in`

The input stream to buffer.

Description

This constructor creates a `BufferedInputStream` that buffers input from the given `InputStream`, using a buffer with the default size of 2048 bytes.

public BufferedInputStream(InputStream in, int size)

Parameters

`in`

The input stream to buffer.

`size`

The size of buffer to use.

Description

This constructor creates a `BufferedInputStream` that buffers input from the given `InputStream`, using a buffer of the given size.

Instance Methods

available

public synchronized int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.available()`

Description

This method returns the number of bytes that can be read without having to wait for more data to become available. The returned value is the sum of the number of bytes remaining in the object's buffer and the number returned as the result of calling the `available()` method of the underlying `InputStream` object.

mark

public synchronized void mark(int readlimit)

Parameters

`readlimit`

The maximum number of bytes that can be read before the saved position becomes invalid.

Overrides

`FilterInputStream.mark()`

Description

This method causes the `BufferedInputStream` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus reread a portion of the buffer.

markSupported

public synchronized boolean markSupported()

Returns

The `boolean` value `true`.

Overrides

`FilterInputStream.markSupported()`

Description

This method returns `true` to indicate that this class supports `mark()` and `reset()`.

read

public synchronized int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read()`

Description

This method returns the next byte from the buffer. If all the bytes in the buffer have been read, the buffer is filled from the underlying `InputStream` and the next byte is returned. If the buffer does not need to be filled, this method returns immediately. If the buffer needs to be filled, this method blocks until data is available from the underlying `InputStream`, the end of the stream is reached, or an exception is thrown.

`public synchronized int read(byte b[], int off, int len) throws IOException`

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read(byte[], int, int)`

Description

This method copies bytes from the internal buffer into the given array `b`, starting at index `off` and continuing for up to `len` bytes. If there are any bytes in the buffer, this method returns immediately.

Otherwise the buffer needs to be filled; this method blocks until the data is available from the underlying `InputStream`, the end of the stream is reached, or an exception is thrown.

reset

public synchronized void reset() throws IOException

Throws

`IOException`

If there was no previous call to this `BufferedInputStream`'s mark method, or the saved position has been invalidated.

Overrides

`FilterInputStream.reset()`

Description

This method sets the position of the `BufferedInputStream` to a position that was saved by a previous call to `mark()`. Subsequent bytes read from this `BufferedInputStream` will begin from the saved position and continue normally.

skip

public synchronized long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.skip()`

Description

This method skips `n` bytes of input. If the new position of the stream is still within the data contained in the buffer, the method returns immediately. Otherwise the `skip()` method of the underlying stream is called. A subsequent call to `read()` forces the buffer to be filled.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`FilterInputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`read(byte[])`	`FilterInputStream`	`toString()`	`Object`
`void wait()`	`Object`	`void wait(long)`	`Object`
`void wait(long, int)`	`Object`		

See Also

`FilterInputStream`, `InputStream`, `IOException`

12. The java.lang Package

Contents:

The package `java.lang` contains classes and interfaces that are essential to the Java language. These include:

- `Object`, the ultimate superclass of all classes in Java.
- `Thread`, the class that controls each thread in a multithreaded program.
- `Throwable`, the superclass of all error and exception classes in Java.
- Classes that encapsulate the primitive data types in Java.

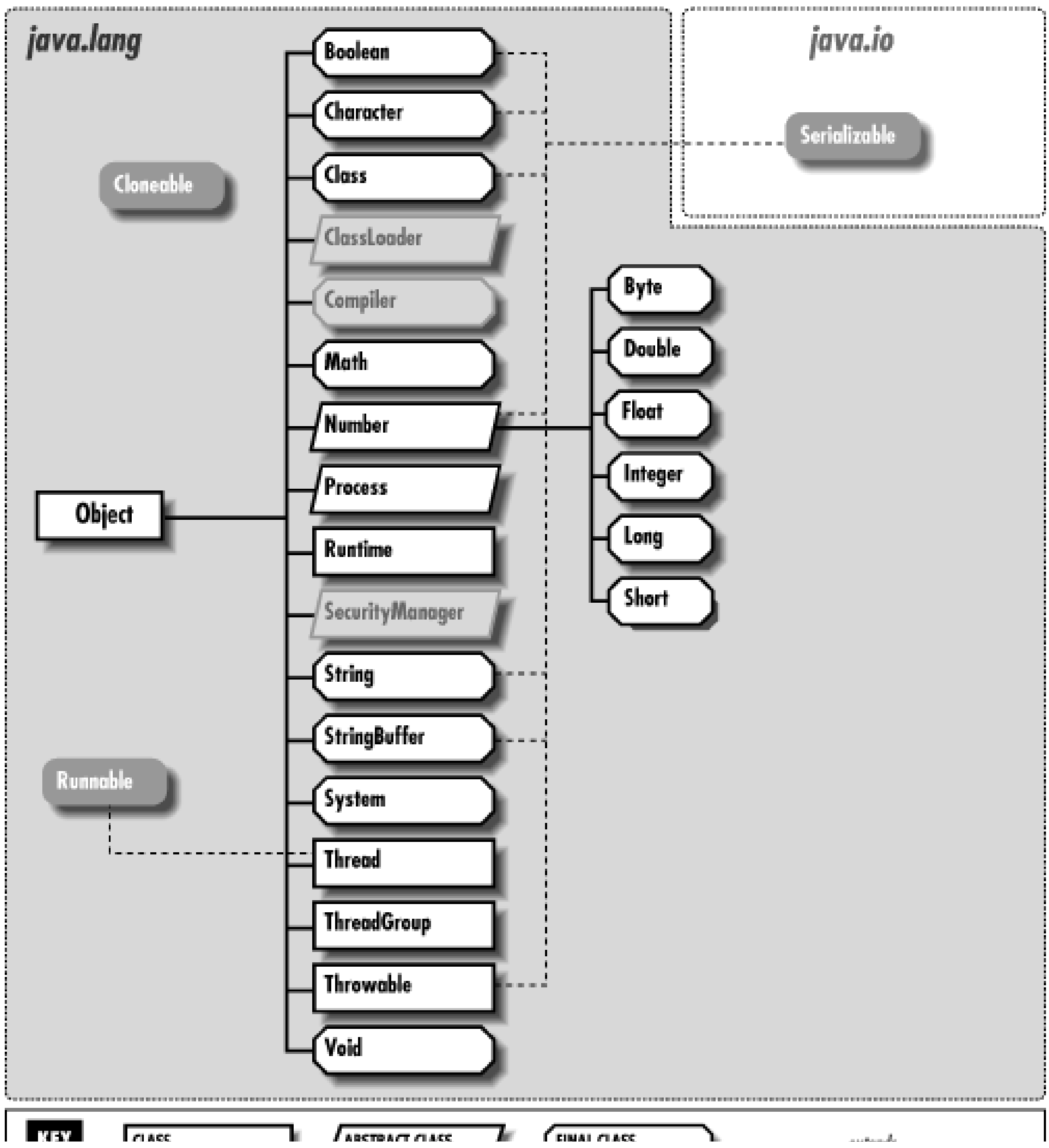
- Classes for accessing system resources and other low-level entities.
- `Math`, a class that provides standard mathematical methods.
- `String`, the class that represents strings.

Because the classes in the `java.lang` package are so essential, the `java.lang` package is implicitly imported by every Java source file. In other words, you can refer to all of the classes and interfaces in `java.lang` using their simple names.

Figure 12.1 shows the class hierarchy for the `java.lang` package.

The possible exceptions in a Java program are organized in a hierarchy of exception classes. The `Throwable` class is at the root of the exception hierarchy. `Throwable` has two immediate subclasses: `Exception` and `Error`. Figure 12.2 shows the standard exception classes defined in the `java.lang` package, while Figure 12.3 shows the standard error classes defined in `java.lang`.

Figure 12.1: The java.lang package



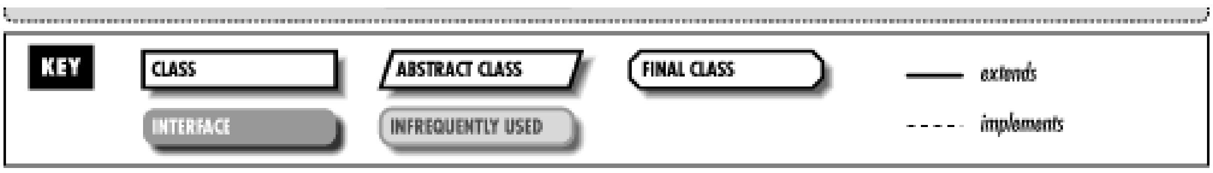


Figure 12.2: The exception classes in the java.lang package

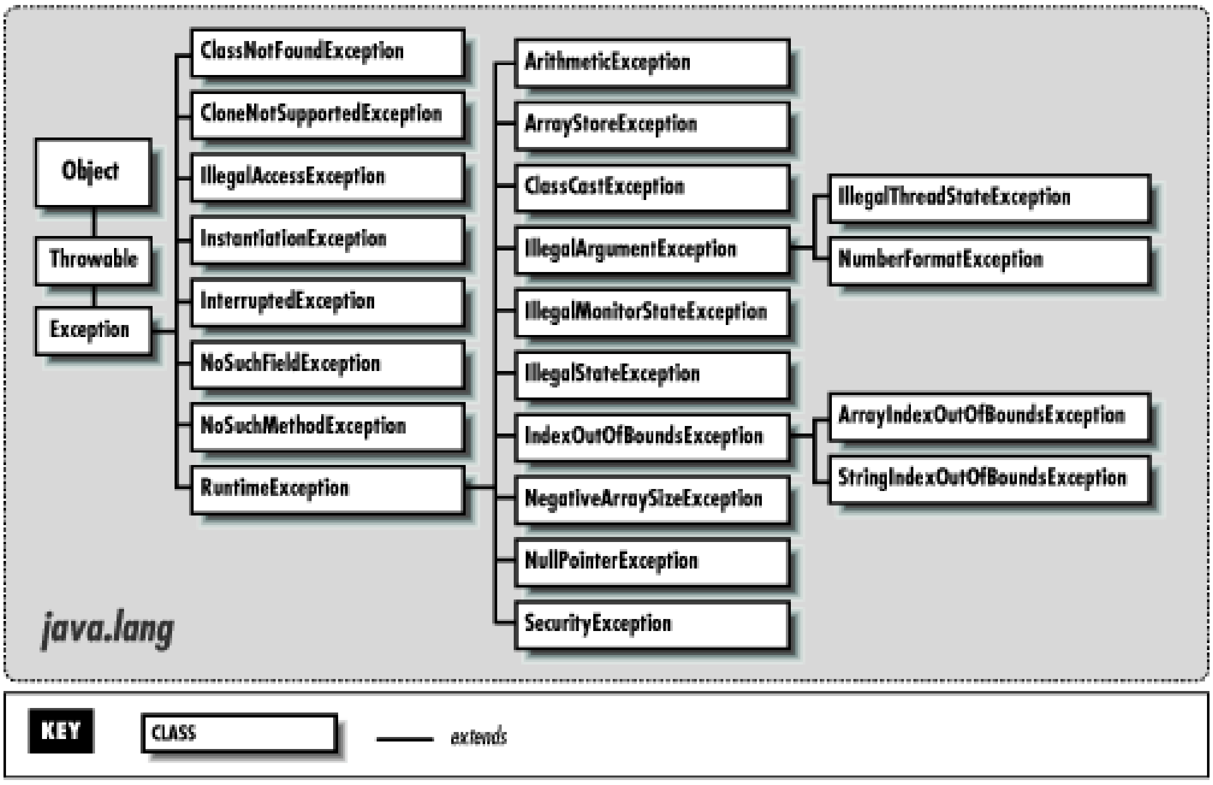
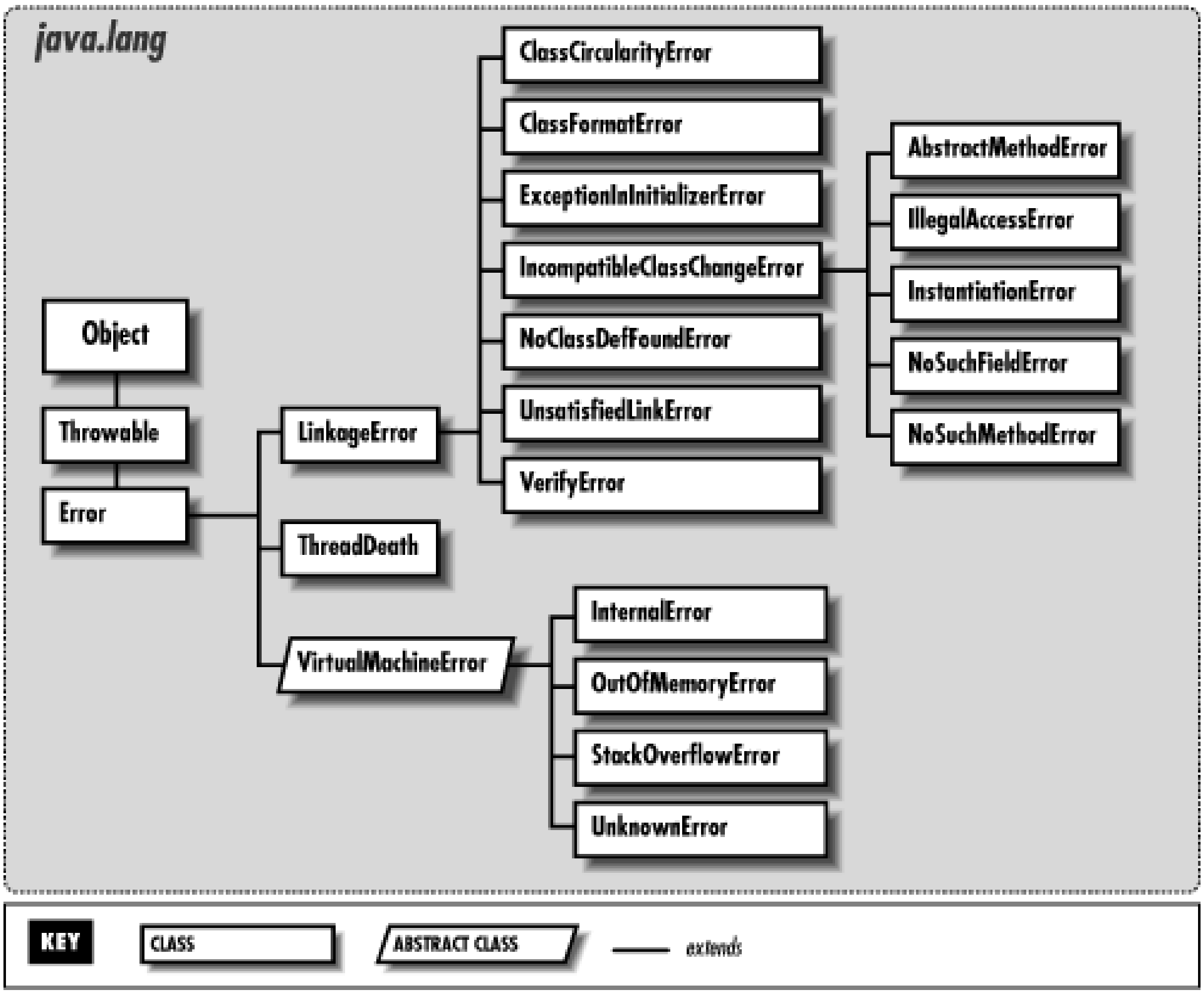


Figure 12.3: The error classes in the java.lang package



AbstractMethodError

Name

AbstractMethodError

Synopsis

Class Name:

```
java.lang.AbstractMethodError
```

Superclass:

```
java.lang.IncompatibleClassChangeError
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `AbstractMethodError` is thrown when there is an attempt to invoke an `abstract` method.

Class Summary

```
public class java.lang.AbstractMethodError
             extends java.lang.IncompatibleClassChangeError {
  // Constructors
  public AbstractMethodError();
  public AbstractMethodError(String s);
}
```

Constructors

AbstractMethodError

public AbstractMethodError()

Description

This constructor creates an `AbstractMethodError` with no associated detail message.

public AbstractMethodError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `AbstractMethodError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `IncompatibleClassChangeError`, `Throwable`

PREVIOUS Writer | HOME BOOK INDEX | NEXT ArithmeticException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13. The java.lang.reflect Package

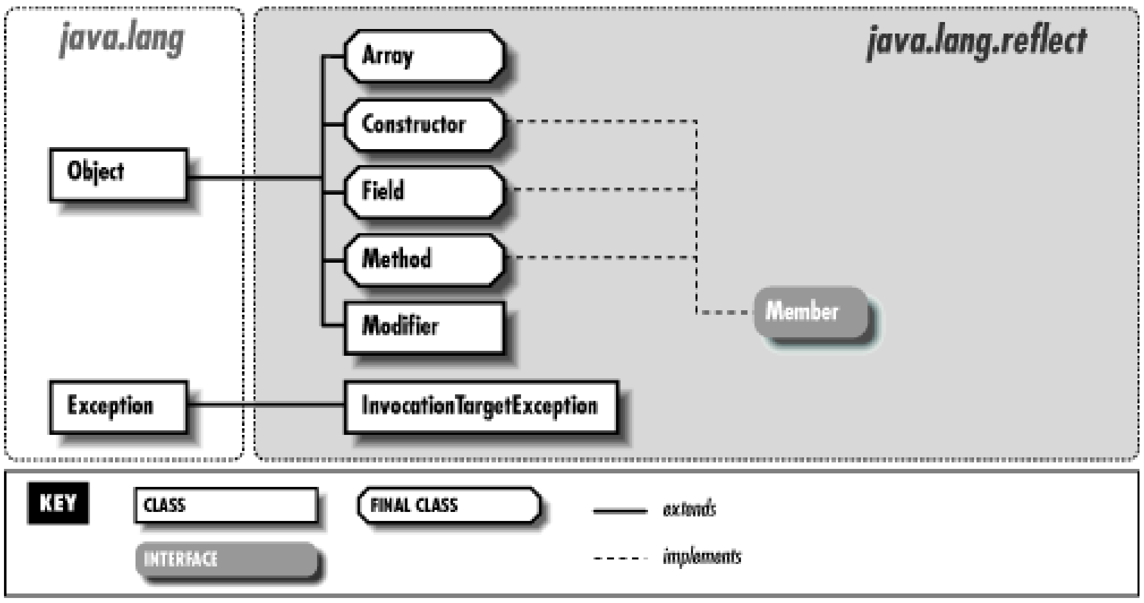
Contents:

The package `java.lang.reflect` is new as of Java 1.1. It contains classes and interfaces that support the Reflection API. Reflection refers to the ability of a class to reflect upon itself, or look inside of itself, to see what it can do. The Reflection API makes it possible to:

- Discover the variables, methods, and constructors of any class.
- Create an instance of any class using any available constructor of that class, even if the class initiating the creation was not compiled with any information about the class to be instantiated.
- Access the variables of any object, even if the accessing class was not compiled with any information about the class to be accessed.
- Call the methods of any object, even if the calling class was not compiled with any information about the class that contains the methods.
- Create an array of objects that are instances of any class, even if the creating class was not compiled with any information about the class.

These capabilities are implemented by the `java.lang.Class` class and the classes in the `java.lang.reflect` package. Figure 13.1 shows the class hierarchy for the `java.lang.reflect` package.

Figure 13.1: The java.lang.reflect package



Java 1.1 currently uses the Reflection API for two purposes:

- The JavaBeans API supports a mechanism for customizing objects that is based on being able to discover their public variables, methods, and constructors. See the forthcoming *Developing Java Beans* from O'Reilly & Associates for more information about the JavaBeans API.
- The object serialization functionality in `java.io` is built on top of the Reflection API. Object serialization allows arbitrary objects to be written to a stream of bytes and then read back later as objects.

Array

Name

Array

Synopsis

Class Name:

`java.lang.reflect.Array`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `Array` class provides static methods to manipulate arbitrary arrays in Java. There are methods to set and retrieve elements in an array, determine the size of an array, and create a new instance of an array.

The `Array` class is used to create array objects and manipulate their elements. The `Array` class is not used to represent array types. Because arrays in Java are objects, array types are represented by `Class` objects.

Class Summary

```
public final class java.lang.reflect.Array extends java.lang.Object {
  // Class Methods
  public static native Object get(Object array, int index);
  public static native boolean getBoolean(Object array, int index);
  public static native byte getByte(Object array, int index);
  public static native char getChar(Object array, int index);
  public static native double getDouble(Object array, int index);
  public static native float getFloat(Object array, int index);
  public static native int getInt(Object array, int index);
  public static native int getLength(Object array);
  public static native long getLong(Object array, int index);
  public static native short getShort(Object array, int index);
  public static Object newInstance(Class componentType, int length);
  public static Object newInstance(Class componentType, int[] dimensions);
  public static native void set(Object array, int index, Object value);
  public static native void setBoolean(Object array, int index, boolean z);
  public static native void setByte(Object array, int index, byte b);
  public static native void setChar(Object array, int index, char c);
  public static native void setDouble(Object array, int index, double d);
  public static native void setFloat(Object array, int index, float f);
  public static native void setInt(Object array, int index, int i);
  public static native void setLong(Object array, int index, long l);
  public static native void setShort(Object array, int index, short s);
}
```

Class Methods

get

`public static native Object get(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The object at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array. If the array contains values of a primitive type, the value at the given index is wrapped in an appropriate object, and the object is returned.

getBoolean

`public static native boolean getBoolean(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `boolean` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `boolean`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `boolean` value.

getByte

`public static native byte getByte(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `byte` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `byte`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `byte` value.

getChar

`public static native char getChar(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `char` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `char`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `char` value.

getDouble

`public static native double getDouble(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `double` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `double`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `double` value.

getFloat

`public static native float getFloat(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `float` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `float`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `float` value.

getInt

`public static native int getInt(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `int` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `int`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `int` value.

getLength

`public static native int getLength(Object array) throws IllegalArgumentException`

Parameters

`array`

An array object.

Returns

The length of the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array.

Description

This method returns the length of the array.

getLong

`public static native long getLong(Object array, long index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `long` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `long`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `long` value.

getShort

`public static native short getShort(Object array, short index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `short` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `short`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `short` value.

newInstance

`public static Object newInstance(Class componentType, int length) throws NegativeArraySizeException`

Parameters

`componentType`

The type of each element in the array.

`length`

The length of the array.

Returns

An array object that contains elements of the given component type and has the specified length.

Throws

`NegativeArraySizeException`

If `length` is negative.

`NullPointerException`

If `componentType` is `null`.

Description

This method creates a single-dimension array with the given length and component type.

`public static Object newInstance(Class componentType, int[] dimensions) throws IllegalArgumentException, NegativeArraySizeException`

Parameters

`componentType`

The type of each element in the array.

`dimensions`

An array that specifies the dimensions of the array to be created.

Returns

An array object that contains elements of the given component type and has the specified number of dimensions.

Throws

`IllegalArgumentException`

If `dimensions` has zero dimensions itself, or if it has too many dimensions (typically 255 array dimensions are supported).

`NegativeArraySizeException`

If `length` is negative.

`NullPointerException`

> If `componentType` is `null`.

Description

> This method creates a multidimensional array with the given dimensions and component type.

set

`public static native void set(Object array, int index, Object value) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

> An array object.

`index`

> An index into the array.

`value`

> The new value.

Throws

`IllegalArgumentException`

> If the given object is not an array, or if it represents an array of primitive values, and the given value cannot be unwrapped and converted to that primitive type.

`ArrayIndexOutOfBoundsException`

> If the given index is invalid.

`NullPointerException`

> If `array` is `null`.

Description

> This method sets the object at the given index in the array to the specified value. If the array contains values of a primitive type, the given value is automatically unwrapped before it is put in the array.

setBoolean

`public static native void setBoolean(Object array, int index, boolean z) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`z`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `boolean` value.

setByte

`public static native void setByte(Object array, int index, byte b) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`b`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `byte` value.

setChar

`public static native void setChar(Object array, int index, char c) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`c`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `char` value.

setDouble

`public static native void setDouble(Object array, int index, double d) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`d`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `double` value.

setFloat

`public static native void setFloat(Object array, int index, float f) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`f`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `float` value.

setInt

`public static native void setInt(Object array, int index, int i) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`i`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `int` value.

setLong

`public static native void setLong(Object array, int index, long l) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`l`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `long` value.

setShort

`public static native void setShort(Object array, int index, short s) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`s`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `short` value.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`

`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ArrayIndexOutOfBoundsException`, `Class`, `IllegalArgumentException`, `NegativeArraySizeException`, `NullPointerException`, `Object`

PREVIOUS Void | HOME BOOK INDEX | NEXT Constructor

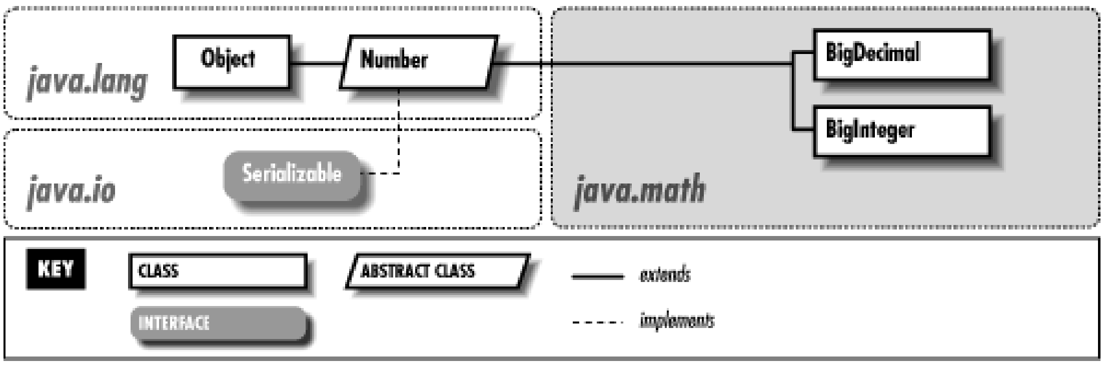
JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14. The java.math Package

Contents:

The package `java.math` is new as of Java 1.1. It contains two classes that support arithmetic on arbitrarily large integers and floating-point numbers. Figure 14.1 shows the class hierarchy for the `java.math` package.

Figure 14.1: The java.math package



BigDecimal

Name

BigDecimal

Synopsis

Class Name:

```
java.math.BigDecimal
```

Superclass:

```
java.lang.Number
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `BigDecimal` class represents arbitrary-precision rational numbers. A `BigDecimal` object provides a good way to represent a real number that exceeds the range or precision that can be represented by a `double` value or the rounding that is done on a `double` value is unacceptable.

The representation for a `BigDecimal` consists of an unlimited precision integer value and an integer scale factor. The scale factor indicates a power of 10 that the integer value is implicitly divided by. For example, a `BigDecimal` would represent the value 123.456 with an integer value of 123456 and the scale factor of 3. Note that the scale factor cannot be negative and a `BigDecimal` cannot overflow.

Most of the methods in `BigDecimal` perform mathematical operations or make comparisons with other `BigDecimal` objects. Operations that result in some loss of precision, such as division, require a rounding method to be specified. The `BigDecimal` class defines constants to represent the different rounding methods. The rounding method determines if the digit before a discarded fraction is rounded up or left unchanged.

Class Summary

```
public class java.math.BigDecimal extends java.lang.Number {
  // Constants
  public static final int ROUND_CEILING;
  public static final int ROUND_DOWN;
  public static final int ROUND_FLOOR;
  public static final int ROUND_HALF_DOWN;
  public static final int ROUND_HALF_EVEN;
  public static final int ROUND_HALF_UP;
  public static final int ROUND_UNNECESSARY;
  public static final int ROUND_UP;
  // Constructors
  public BigDecimal(double val);
  public BigDecimal(String val);
  public BigDecimal(BigInteger val);
  public BigDecimal(BigInteger val, int scale);
  // Class Methods
  public static BigDecimal valueOf(long val);
  public static BigDecimal valueOf(long val, int scale);
  // Instance Methods
  public BigDecimal abs();
  public BigDecimal add(BigDecimal val);
  public int compareTo(BigDecimal val);
  public BigDecimal divide(BigDecimal val, int roundingMode);
  public BigDecimal divide(BigDecimal val, int scale, int roundingMode);
  public double doubleValue();
  public boolean equals(Object x);
  public float floatValue();
  public int hashCode();
  public int intValue();
  public long longValue();
  public BigDecimal max(BigDecimal val);
  public BigDecimal min(BigDecimal val);
  public BigDecimal movePointLeft(int n);
  public BigDecimal movePointRight(int n);
  public BigDecimal multiply(BigDecimal val);
  public BigDecimal negate();
  public int scale();
  public BigDecimal setScale(int scale);
  public BigDecimal setScale(int scale, int roundingMode);
  public int signum();
  public BigDecimal subtract(BigDecimal val);
  public BigInteger toBigInteger();
  public String toString();
}
```

Constants

ROUND_CEILING

public static final int ROUND_CEILING

Description

> A rounding method that rounds towards positive infinity. Under this method, the value is rounded to the least integer greater than or equal to its value. For example, 2.5 rounds to 3 and -2.5 rounds to -2.

ROUND_DOWN

public static final int ROUND_DOWN

Description

> A rounding method that rounds towards zero by truncating. For example, 2.5 rounds to 2 and -2.5 rounds to -2.

ROUND_FLOOR

public static final int ROUND_FLOOR

Description

> A rounding method that rounds towards negative infinity. Under this method, the value is rounded to the greatest integer less than or equal to its value. For example, 2.5 rounds to 2 and -2.5 rounds to -3.

ROUND_HALF_DOWN

public static final int ROUND_HALF_DOWN

Description

> A rounding method that increments the digit prior to a discarded fraction if the fraction is greater than 0.5; otherwise, the digit is left unchanged. For example, 2.5 rounds to 2, 2.51 rounds to 3, -2.5 rounds to -2, and -2.51 rounds to -3.

ROUND_HALF_EVEN

public static final int ROUND_HALF_EVEN

Description

> A rounding method that behaves like `ROUND_HALF_UP` if the digit prior to the discarded fraction is odd; otherwise it behaves like `ROUND_HALF_DOWN`. For example, 2.5 rounds to 2, 3.5 rounds to 4, -2.5 rounds to -2, and -3.5 rounds to -4.

ROUND_HALF_UP

public static final int ROUND_HALF_UP

Description

> A rounding method that increments the digit prior to a discarded fraction if the fraction is greater than or equal to 0.5; otherwise, the digit is left unchanged. For example, 2.5 rounds to 3, 2.49 rounds to 2, -2.5 rounds to -3, and -2.49 rounds to -2.

ROUND_UNNECESSARY

public static final int ROUND_UNNECESSARY

Description

> A constant that specifies that rounding is not necessary. If the result really does require rounding, an `ArithmeticException` is thrown.

ROUND_UP

public static final int ROUND_UP

Description

> A rounding method that rounds away from zero by truncating. For example, 2.5 rounds to 3 and -2.5 rounds to -3.

Constructors

BigDecimal

public BigDecimal(double val) throws NumberFormatException

Parameters

> `val`
>
> > The initial value.

Throws

`NumberFormatException`

If the `double` has any of the special values: `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY`, or `Double.NaN`.

Description

This constructor creates a `BigDecimal` with the given initial value. The scale of the `BigDecimal` that is created is the smallest value such that (`10^scale x val`) is an integer.

public BigDecimal(String val) throws NumberFormatException

Parameters

`val`

The initial value.

Throws

`NumberFormatException`

If the string cannot be parsed into a valid `BigDecimal`.

Description

This constructor creates a `BigDecimal` with the initial value specified by the `String`. The string can contain an optional minus sign, followed by zero or more decimal digits, followed by an optional fraction. The fraction must contain a decimal point and zero or more decimal digits. The string must contain as least one digit in the integer or fractional part. The scale of the `BigDecimal` that is created is equal to the number of digits to the right of the decimal point or 0 if there is no decimal point. The mapping from characters to digits is provided by the `Character.digit()` method.

public BigDecimal(BigInteger val)

Parameters

`val`

The initial value.

Description

This constructor creates a `BigDecimal` whose initial value comes from the given `BigInteger`. The scale of the `BigDecimal` that is created is 0.

`public BigDecimal(BigInteger val, int scale) throws NumberFormatException`

Parameters

`val`

The initial value.

`scale`

The initial scale.

Throws

`NumberFormatException`

If `scale` is negative.

Description

This constructor creates a `BigDecimal` from the given parameters. The `scale` parameter specifies how many digits of the supplied `BigInteger` fall to the right of the decimal point.

Class Methods

valueOf

public static BigDecimal valueOf(long val)

Parameters

`val`

The initial value.

Returns

A `BigDecimal` that represents the given value.

Description

This method creates a `BigDecimal` from the given `long` value. The scale of the `BigDecimal` that is created is 0.

`public static BigDecimal valueOf(long val, int scale) throws NumberFormatException`

Parameters

`val`

The initial value.

`scale`

The initial scale.

Returns

A `BigDecimal` that represents the given value and scale.

Throws

`NumberFormatException`

If `scale` is negative.

Description

This method creates a `BigDecimal` from the given parameters. The `scale` parameter specifies how many digits of the supplied `long` fall to the right of the decimal point.

Instance Methods

abs

public BigDecimal abs()

Returns

A `BigDecimal` that contains the absolute value of this number.

Description

This method returns the absolute value of this `BigDecimal`. If this `BigDecimal` is nonnegative, it is returned. Otherwise, a new `BigDecimal` that contains the absolute value of this `BigDecimal` is returned. The scale of the new `BigDecimal` is the same as that of this `BigDecimal`.

add

public BigDecimal add(BigDecimal val)

Parameters

`val`

The number to be added.

Returns

A new `BigDecimal` that contains the sum of this number and the given value.

Description

This method returns the sum of this `BigDecimal` and the given `BigDecimal` as a new `BigDecimal`. The value of the new `BigDecimal` is the sum of the values of the two `BigDecimal` objects being added; the scale is the maximum of their two scales.

compareTo

public int compareTo(BigDecimal val)

Parameters

`val`

The number to be compared.

Returns

`-1` if this number is less than `val`, `0` if this number is equal to `val`, or `1` if this number is greater than `val`.

Description

This method compares this `BigDecimal` to the given `BigDecimal` and returns a value that indicates the result of the comparison. The method considers two `BigDecimal` objects that have the same values but different scales to be equal. This method can be used to implement all six of the standard boolean comparison operators: `==`, `!=`, `<=`, `<`, `>=`, and `>`.

divide

`public BigDecimal divide(BigDecimal val, int roundingMode) throws ArithmeticException, IllegalArgumentException`

Parameters

`val`

The divisor.

`roundingMode`

The rounding mode.

Returns

A new `BigDecimal` that contains the result (quotient) of dividing this number by the supplied value.

Throws

`ArithmeticException`

If `val` is `0`, or if `ROUND_UNNECESSARY` is specified for the rounding mode but rounding is necessary.

`IllegalArgumentException`

If `roundingMode` is not a valid value.

Description

This method returns the quotient that results from dividing this `BigDecimal` by the given `BigDecimal` and applying the specified rounding mode. The quotient is returned as a new `BigDecimal` that has the same scale as the scale of this `BigDecimal` scale. One of the rounding constants must be specified for the rounding mode.

`public BigDecimal divide(BigDecimal val, int scale, int roundingMode) throws ArithmeticException, IllegalArgumentException`

Parameters

`val`

The divisor.

`scale`

The scale for the result.

`roundingMode`

The rounding mode.

Returns

A new `BigDecimal` that contains the result (quotient) of dividing this number by the supplied value.

Throws

`ArithmeticException`

If `val` is `0`, if `scale` is less than zero, or if `ROUND_UNNECESSARY` is specified for the rounding mode but rounding is necessary.

`IllegalArgumentException`

If `roundingMode` is not a valid value.

Description

This method returns the quotient that results from dividing dividing this `BigDecimal` by the given `BigDecimal` and applying the specified rounding mode. The quotient is returned as a new `BigDecimal` that has the given scale. One of the rounding constants must be specified for the rounding mode.

doubleValue

public double doubleValue()

Returns

The value of this `BigDecimal` as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this `BigDecimal` as a `double`. If the value exceeds the limits of a `double`, `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` is returned.

equals

public boolean equals(Object x)

Parameters

`x`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `x` is an instance of `BigDecimal`, and it represents the same value as this `BigDecimal`. In order to be considered equal using this method, two `BigDecimal` objects must have the same values and scales.

floatValue

public float floatValue()

Returns

The value of this `BigDecimal` as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this `BigDecimal` as a `float`. If the value exceeds the limits of a `float`, `Float.POSITIVE_INFINITY` or `Float.NEGATIVE_INFINITY` is returned.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `BigDecimal`.

intValue

public int intValue()

Returns

The value of this `BigDecimal` as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this `BigDecimal` as an `int`. If the value exceeds the limits of an `int`, the excessive high-order bits are discarded. Any fractional part of this `BigDecimal` is truncated.

longValue

public long longValue()

Returns

The value of this `BigDecimal` as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this `BigDecimal` as a `long`. If the value exceeds the limits of a `long`, the excessive high-order bits are discarded. Any fractional part of this `BigDecimal` is also truncated.

max

public BigDecimal max(BigDecimal val)

Parameters

`val`

The number to be compared.

Returns

The `BigDecimal` that represents the greater of this number and the given value.

Description

This method returns the greater of this `BigDecimal` and the given `BigDecimal`.

min

public BigDecimal min(BigDecimal val)

Parameters

`val`

The number to be compared.

Returns

The `BigDecimal` that represents the lesser of this number and the given value.

Description

This method returns the lesser of this `BigDecimal` and the given `BigDecimal`.

movePointLeft

public BigDecimal movePointLeft(int n)

Parameters

`n`

The number of digits to move the decimal point to the left.

Returns

A new `BigDecimal` that contains the adjusted number.

Description

This method returns a `BigDecimal` that is computed by shifting the decimal point of this `BigDecimal` left by the given number of digits. If `n` is nonnegative, the value of the new `BigDecimal` is the same as the current value, and the scale is increased by `n`. If `n` is negative, the method call is equivalent to `movePointRight(-n)`.

movePointRight

public BigDecimal movePointRight(int n)

Parameters

`n`

The number of digits to move the decimal point to the right.

Returns

A new `BigDecimal` that contains the adjusted number.

Description

This method returns a `BigDecimal` that is computed by shifting the decimal point of this `BigDecimal` right by the given number of digits. If `n` is nonnegative, the value of the new `BigDecimal` is the same as the current value, and the scale is decreased by `n`. If `n` is negative, the method call is equivalent to `movePointLeft(-n)`.

multiply

public BigDecimal multiply(BigDecimal val)

Parameters

`val`

The number to be multiplied.

Returns

A new `BigDecimal` that contains the product of this number and the given value.

Description

This method multiplies this `BigDecimal` and the given `BigDecimal`, and returns the result as a new `BigDecimal`. The value of the new `BigDecimal` is the product of the values of the two `BigDecimal` objects being added; the scale is the sum of their two scales.

negate

public BigDecimal negate()

Returns

A new `BigDecimal` that contains the negative of this number.

Description

This method returns a new `BigDecimal` that is identical to this `BigDecimal` except that its sign is reversed. The scale of the new `BigDecimal` is the same as the scale of this `BigDecimal`.

scale

public int scale()

Returns

The scale of this number.

Description

This method returns the scale of this `BigDecimal`.

setScale

`public BigDecimal setScale(int scale) throws ArithmeticException, IllegalArgumentException`

Parameters

`scale`

a The new scale.

Returns

A new `BigDecimal` that is identical to this number, except that is has the given scale.

Throws

`ArithmeticException`

If the new number cannot be calculated without rounding.

`IllegalArgumentException`

This exception is never thrown.

Description

This method creates a new `BigDecimal` that has the given scale and a value that is calculated by multiplying or dividing the value of this `BigDecimal` by the appropriate power of 10 to maintain the overall value. The method is typically used to increase the scale of a number, not decrease it. It can decrease the scale, however, if there are enough zeros in the fractional part of the value to allow for rescaling without loss of precision.

Calling this method is equivalent to calling `setScale(scale, BigDecimal.ROUND_UNNECESSARY)`.

`public BigDecimal setScale(int scale, int roundingMode) throws ArithmeticException, IllegalArgumentException`

Parameters

`scale`

The new scale.

`roundingMode`

The rounding mode.

Returns

A new `BigDecimal` that contains this number adjusted to the given scale.

Throws

`ArithmeticException`

If `scale` is less than zero, or if `ROUND_UNNECESSARY` is specified for the rounding mode but rounding is necessary.

`IllegalArgumentException`

If `roundingMode` is not a valid value.

Description

This method creates a new `BigDecimal` that has the given scale and a value that is calculated by multiplying or dividing the value of this `BigDecimal` by the appropriate power of 10 to maintain the overall value. When the scale is reduced, the value must be divided, so precision may be lost. In this case, the specified rounding mode is used.

signum

public int signum()

Returns

`-1` if this number is negative, `0` if this number is zero, or `1` if this number is positive.

Description

This method returns a value that indicates the sign of this `BigDecimal`.

subtract

public BigDecimal subtract(BigDecimal val)

Parameters

`val`

The number to be subtracted.

Returns

A new `BigDecimal` that contains the result of subtracting the given number from this number.

Description

This method subtracts the given `BigDecimal` from this `BigDecimal` and returns the result as a new `BigDecimal`. The value of the new `BigDecimal` is the result of subtracting the value of the given `BigDecimal` from this `BigDecimal`; the scale is the maximum of their two scales.

toBigInteger

public BigInteger toBigInteger()

Returns

The value of this `BigDecimal` as a `BigInteger`.

Description

This method returns the value of this `BigDecimal` as a `BigInteger`. The fractional part of this number is truncated.

toString

public String toString()

Returns

A string representation of this object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `BigDecimal`. A minus sign represents the sign, and a decimal point is used to represent the scale. The mapping from digits to characters is provided by the `Character.forDigit()` method.

Inherited Methods

Method	Inherited From	Method	Inherited From
`byteValue()`	`Number`	`clone()`	`Object`
`getClass()`	`Object`	`finalize()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`shortValue()`	`Number`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`ArithmeticException`, `BigInteger`, `Character`, `Double`, `Float`, `IllegalArgumentException`, `Integer`, `Long`, `Number`, `NumberFormatException`

JAVA Fundamental Classes Reference

15. The java.net Package

Contents:

The package `java.net` contains classes and interfaces that provide a powerful infrastructure for networking in Java. These include:

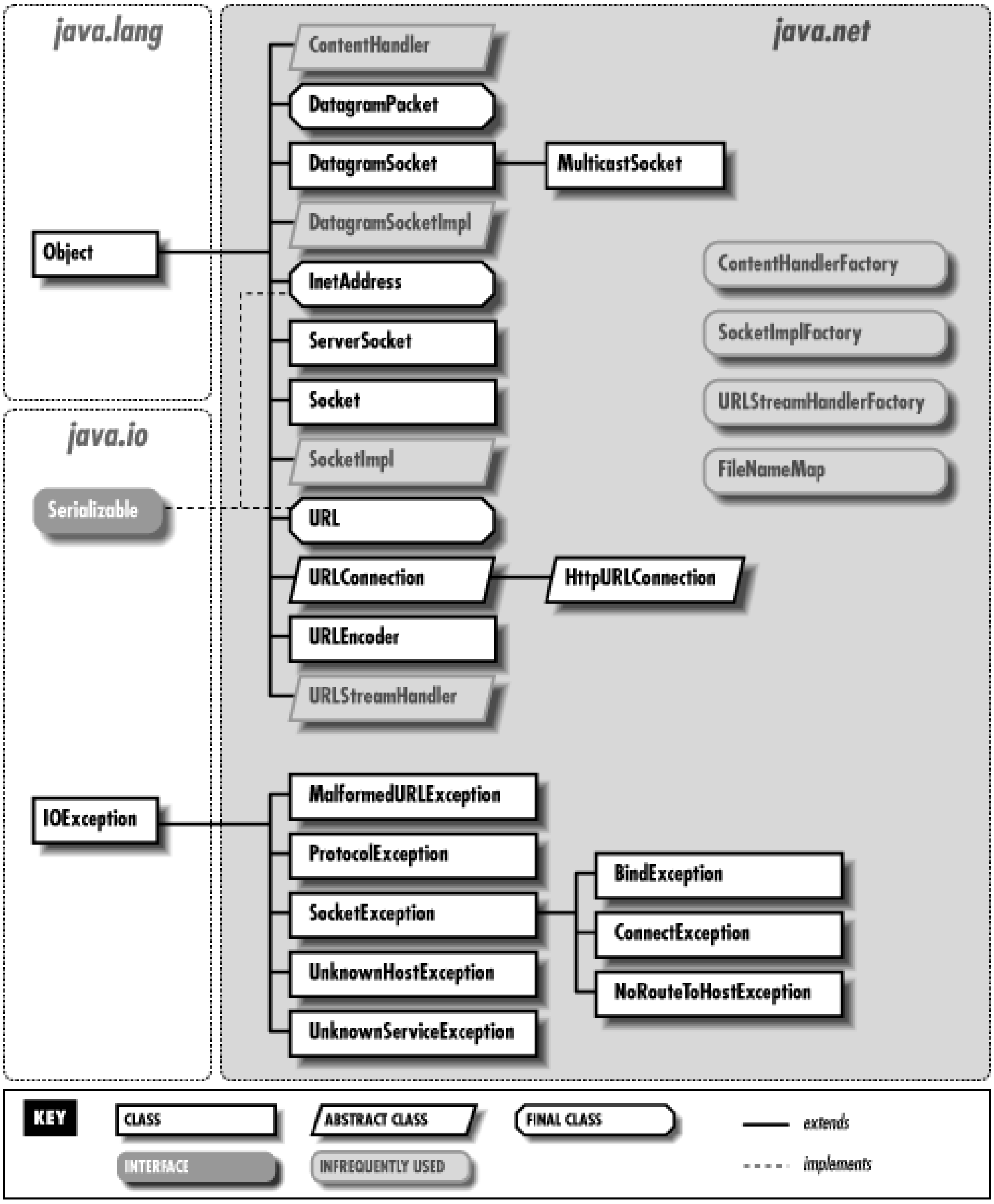
- The `URL` class for basic access to Uniform Resource Locators (URLs).
- The `URLConnection` class, which supports more complex operations on URLs.
- The `Socket` class for connecting to particular ports on specific Internet hosts and reading and writing

data using streams.

- The `ServerSocket` class for implementing servers that accept connections from clients.
- The `DatagramSocket`, `MulticastSocket`, and `DatagramPacket` classes for implementing low-level networking.
- The `InetAddress` class, which represents Internet addresses.

Figure 15.1 shows the class hierarchy for the `java.net` package.

Figure 15.1: The java.net package



BindException

Name

BindException

Synopsis

Class Name:

    `java.net.BindException`

Superclass:

    `java.net.SocketException`

Immediate Subclasses:

    None

Interfaces Implemented:

    None

Availability:

    New as of JDK 1.1

Description

A `BindException` is thrown when a socket cannot be bound to a local address and port, which can occur if the port is already in use or the address is unavailable.

Class Summary

```
public class java.net.BindException extends java.net.SocketException {
  // Constructors
  public BindException();
  public BindException(String msg);
}
```

Constructors

BindException

public BindException()

Description

This constructor `creates` a `BindException` with no associated detail message.

public BindException(String msg)

Parameters

`msg`

The detail message.

Description

This constructor creates a `BindException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`, `SocketException`

JAVA Fundamental Classes Reference

16. The java.text Package

Contents:

The package `java.text` is new as of Java 1.1. It contains classes that support the internationalization of Java programs. The internationalization classes can be grouped as follows:

- Classes for formatting string representations of dates, times, numbers, and messages based on the conventions of a locale.
- Classes that collate strings according to the rules of a locale.
- Classes for finding boundaries in text according to the rules of a locale.

Many of the classes in `java.text` rely upon a `java.util.Locale` object to provide information about the locale that is in use.

The `Format` class is the superclass of all of the classes that generate and parse string representations of various types of data. The `DateFormat` class formats and parses dates and times according to the customs and language of a particular locale. Similarly, the `NumberFormat` class formats and parses numbers, including currency values, in a locale-dependent manner.

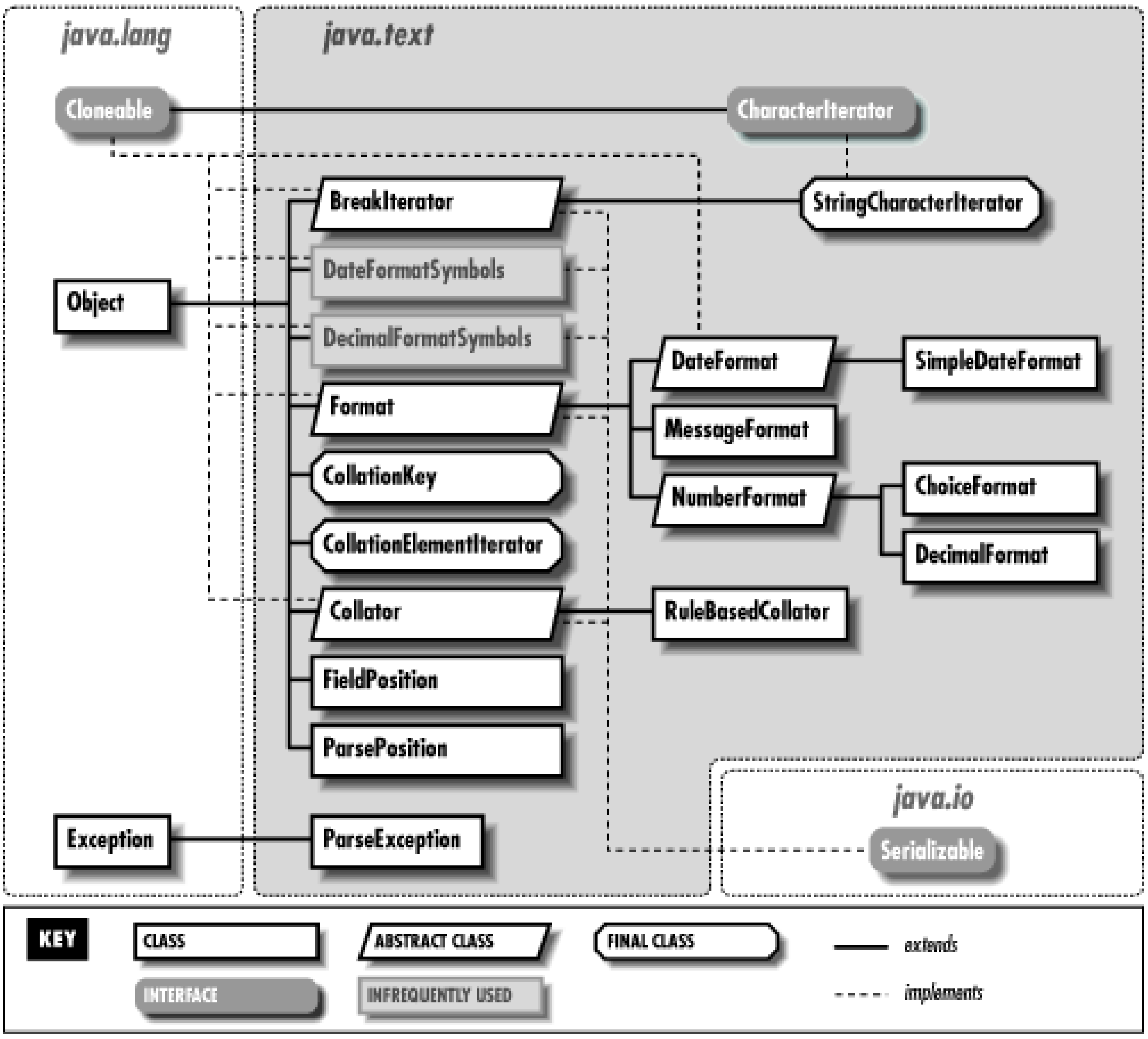
The `MessageFormat` class can create a textual message from a pattern string, while `ChoiceFormat` maps numerical ranges to strings. By themselves, these classes do not provide different results for different locales. However, they can be used in conjunction with `java.util.ResourceBundle` objects that generate locale-specific pattern strings.

The `Collator` class handles collating strings according to the rules of a particular locale. Different languages have different characters and different rules for sorting those characters; `Collator` and its subclass, `RuleBasedCollator`, are designed to take those differences into account when collating strings. In addition, the `CollationKey` class can be used to optimize the sorting of a large collection of strings.

The `BreakIterator` class finds various boundaries, such as word boundaries and line boundaries, in textual data. Again, `BreakIterator` locates these boundaries according to the rules of a particular locale.

Figure 16.1 shows the class hierarchy for the `java.text` package.

Figure 16.1: The java.text package



BreakIterator

Name

BreakIterator

Synopsis

Class Name:

    java.text.BreakIterator

Superclass:

    java.lang.Object

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `BreakIterator` class is an `abstract` class that defines methods that find the locations of boundaries in text, such as word boundaries and sentence boundaries. A `BreakIterator` operates on the object passed to its `setText()` method; that object must implement the `CharacterIterator` interface or be a `String` object. When a `String` is passed to `setText()`, the `BreakIterator` creates an internal `StringCharacterIterator` to iterate over the `String`.

When you use a `BreakIterator`, you call `first()` to get the location of the first boundary and then repeatedly call `next()` to iterate through the subsequent boundaries.

The `BreakIterator` class defines four static factory methods that return instances of `BreakIterator` that locate various kinds of boundaries. Each of these factory methods selects a concrete subclass of `BreakIterator` based either on the default locale or a specified locale. You must create a separate instance of `BreakIterator` to handle each kind of boundary you are trying to locate:

- `getWordInstance()` returns an iterator that locates word boundaries, which is useful for search-and-replace operations. A word iterator correctly handles punctuation marks.
- `getSentenceInstance()` returns an iterator that locates sentence boundaries, which is useful for textual selection. A sentence iterator correctly handle punctuation marks.
- `getLineInstance()` returns an iterator that locates line boundaries, which is useful in line wrapping. A line iterator correctly handles hyphenation and punctuation.
- `getCharacterInstance()` returns an iterator that locates boundaries between characters, which is useful for allowing the cursor to interact with characters appropriately, since some characters are stored as a base character and a diacritical mark, but only represent one display character.

Class Summary

```
public abstract class java.util.BreakIterator extends java.lang.Object
                      implements java.lang.Cloneable,
                                 java.io.Serializable {
  // Constants
  public final static int DONE;
  // Constructors
  protected BreakIterator();
  // Class Methods
  public static synchronized Locale[] getAvailableLocales();
  public static BreakIterator getCharacterInstance();
  public static BreakIterator getCharacterInstance(Locale where);
  public static BreakIterator getLineInstance();
  public static BreakIterator getLineInstance(Locale where);
  public static BreakIterator getSentenceInstance();
  public static BreakIterator getSentenceInstance(Locale where);
  public static BreakIterator getWordInstance();
  public static BreakIterator getWordInstance(Locale where);
  // Instance Methods
  public Object clone();
  public abstract int current();
  public abstract int first();
  public abstract int following(int offset);
  public abstract CharacterIterator getText();
  public abstract int last();
  public abstract int next();
  public abstract int next(int n)
  public abstract int previous();
  public abstract void setText(CharacterIterator newText);
  public void setText(String newText);
}
```

Constants

DONE

public final static int DONE

Description

> A constant that is returned by `next()` or `previous()` if there are no more breaks to be returned.

Constructors

BreakIterator

protected BreakIterator()

Description

> This constructor should be called only from constructors of subclasses.

Class Methods

getAvailableLocales

public static synchronized Locale[] getAvailableLocales()

Returns

> An array of `Locale` objects.

Description

> This method returns an array of the `Locale` objects that can be passed to `getCharacterInstance()`, `getLineInstance()`, `getSentenceInstance()`, or `getWordInstance()`.

getCharacterInstance

public static BreakIterator getCharacterInstance()

Returns

> A `BreakIterator` appropriate for the default `Locale`.

Description

> This method creates a `BreakIterator` that can locate character boundaries in the default `Locale`.

public static BreakIterator getCharacterInstance(Locale where)

Parameters

> `where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate character boundaries in the given `Locale`.

getLineInstance

public static BreakIterator getLineInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate line boundaries in the default `Locale`.

public static BreakIterator getLineInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate line boundaries in the given `Locale`.

getSentenceInstance

public static BreakIterator getSentenceInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate sentence boundaries in the default `Locale`.

public static BreakIterator getSentenceInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate sentence boundaries in the given `Locale`.

getWordInstance

public static BreakIterator getWordInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate word boundaries in the default `Locale`.

public static BreakIterator getWordInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate word boundaries in the given `Locale`.

Instance Methods

clone

public Object clone()

Returns

A copy of this `BreakIterator`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `BreakIterator` and then returns it.

current

public abstract int current()

Returns

The current position of this `BreakIterator`.

Description

This method returns the current position of this `BreakIterator`. The current position is the character index of the most recently returned boundary.

first

public abstract int first()

Returns

> The position of the first boundary of this `BreakIterator`.

Description

> This method finds the first boundary in this `BreakIterator` and returns its character index. The current position of the iterator is set to this boundary.

following

public abstract int following(int offset)

Parameters

> `offset`
>
> > An offset into this `BreakIterator`.

Returns

> The position of the first boundary after the given offset of this `BreakIterator` or `DONE` if there are no more boundaries.

Throws

> `IllegalArgumentException`
>
> > If `offset` is not a valid value for the `CharacterIterator` of this `BreakIterator`.

Description

> This method finds the first boundary after the given offset in this `BreakIterator` and returns its character index.

getText

public abstract CharacterIterator getText()

Returns

The `CharacterIterator` that this `BreakIterator` uses.

Description

This method returns a `CharacterIterator` that represents the text this `BreakIterator` examines.

last

public abstract int last()

Returns

The position of the last boundary of this `BreakIterator`.

Description

This method finds the last boundary in this `BreakIterator` and returns its character index. The current position of the iterator is set to this boundary.

next

public abstract int next()

Returns

The position of the next boundary of this `BreakIterator` or `DONE` if there are no more boundaries.

Description

This method finds the next boundary in this `BreakIterator` after the current position and returns its character index. The current position of the iterator is set to this boundary.

public abstract int next(int n)

Parameters

`n`

The boundary to return. A positive value moves to a later boundary a negative value moves to a previous boundary; the value 0 does nothing.

Returns

The position of the requested boundary of this `BreakIterator`.

Description

This method finds the `n`th boundary in this `BreakIterator`, starting from the current position, and returns its character index. The current position of the iterator is set to this boundary.

For example, `next(-2)` finds the third previous boundary. Thus `next(1)` is equivalent to `next()`, `next(-1)` is equivalent to `previous()`, and `next(0)` does nothing.

previous

public abstract int previous()

Returns

The position of the previous boundary of this `BreakIterator`.

Description

This method finds the previous boundary in this `BreakIterator`, starting from the current position, and returns its character index. The current position of the iterator is set to this boundary.

setText

public abstract void setText(CharacterIterator newText)

Parameters

`newText`

The `CharacterIterator` that contains the text to be examined.

Description

This method tells this `BreakIterator` to examine the piece of text specified by the `CharacterIterator`. This current position of this `BreakIterator` is set to `first()`.

public void setText(String newText)

Parameters

`newText`

The `String` that contains the text to be examined.

Description

This method tells this `BreakIterator` to examine the piece of text specified by the `String`, using a `StringCharacterIterator` created from the given string. This current position of this `BreakIterator` is set to `first()`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`CharacterIterator`, `Locale`, `String`, `StringCharacterIterator`

PREVIOUS
UnknownServiceException

HOME
BOOK INDEX

NEXT
CharacterIterator

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

---

17. The java.util Package

Contents:

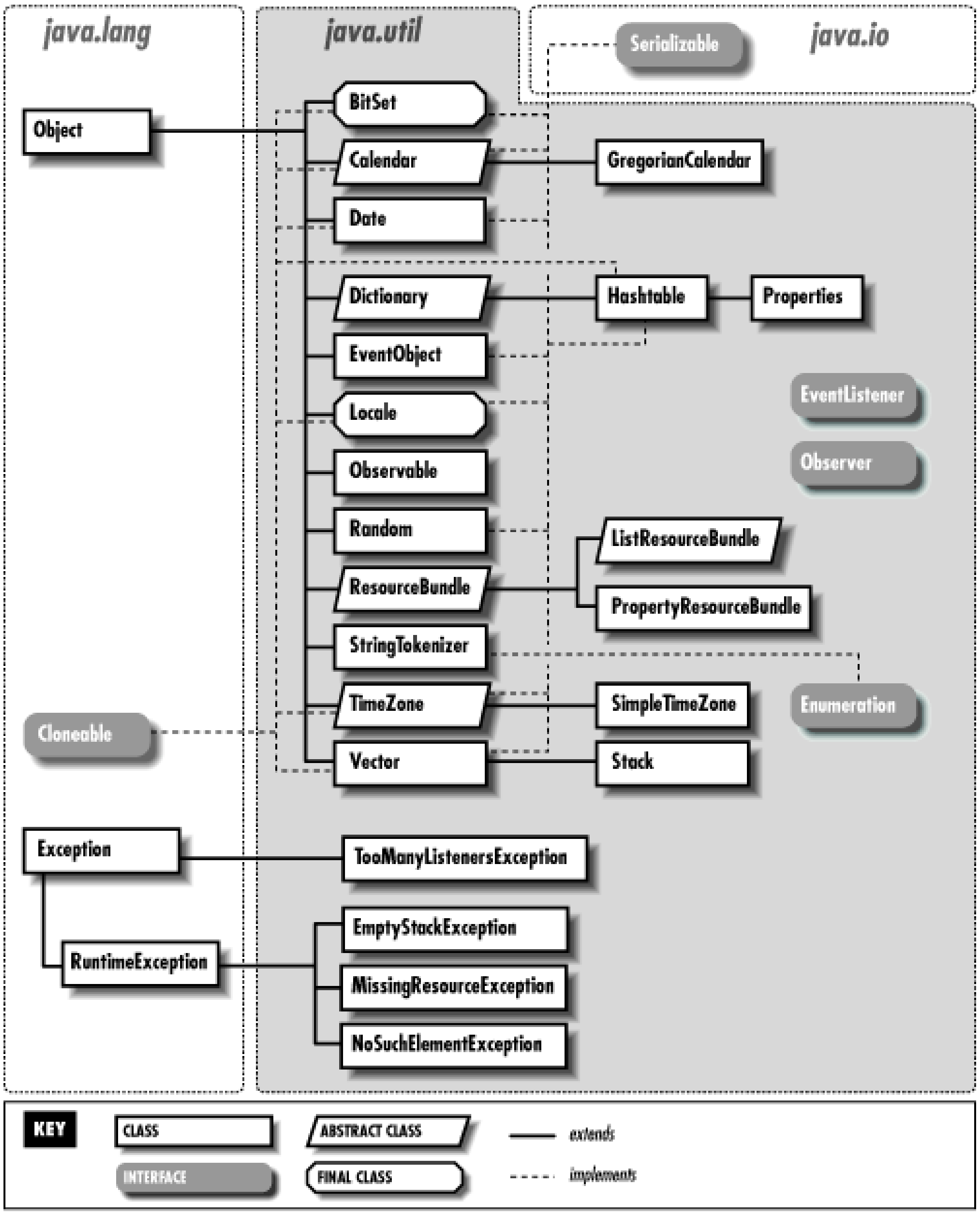
The package `java.util` contains a number of useful classes and interfaces. Although the name of the package might imply that these are utility classes, they are really more important than that. In fact, Java depends directly on several of the classes in this package, and many programs will find these classes indispensable. The classes and interfaces in `java.util` include:

- The `Hashtable` class for implementing hashtables, or associative arrays.

- The `Vector` class, which supports variable-length arrays.
- The `Enumeration` interface for iterating through a collection of elements.
- The `StringTokenizer` class for parsing strings into distinct tokens separated by delimiter characters.
- The `EventObject` class and the `EventListener` interface, which form the basis of the new AWT event model in Java 1.1.
- The `Locale` class in Java 1.1, which represents a particular locale for internationalization purposes.
- The `Calendar` and `TimeZone` classes in Java. These classes interpret the value of a `Date` object in the context of a particular calendar system.
- The `ResourceBundle` class and its subclasses, `ListResourceBundle` and `PropertyResourceBundle`, which represent sets of localized data in Java 1.1.

Figure 17.1 shows the class hierarchy for the `java.util` package.

Figure 17.1: The java.util package



BitSet

Name

BitSet

Synopsis

Class Name:

`java.util.BitSet`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `BitSet` class implements a set of bits. The set grows in size as needed. Each element of a `BitSet` has a boolean value. When a `BitSet` object is created, all of the bits are set to `false` by default. The bits in a `BitSet` are indexed by nonnegative integers, starting at 0. The size of a `BitSet` is the number of bits that it currently contains. The `BitSet` class provides methods to set, clear, and retrieve the values of the individual bits in a `BitSet`. There are also methods to perform logical AND, OR, and XOR operations.

Class Summary

```
public final class java.util.BitSet extends java.lang.Object
                   implements java.lang.Cloneable, java.io.Serializable {
  // Constructors
  public BitSet();
  public BitSet(int nbits);
  // Instance Methods
  public void and(BitSet set);
  public void clear(int bit);
  public Object clone();
  public boolean equals(Object obj);
  public boolean get(int bit);
  public int hashCode();
  public void or(BitSet set);
  public void set(int bit);
  public int size();
  public String toString();
  public void xor(BitSet set);
}
```

Constructors

BitSet

public BitSet()

Description

This constructor creates a `BitSet` with a default size of 64 bits. All of the bits in the `BitSet` are initially set to `false`.

public BitSet(int nbits)

Parameters

`nbits`

The initial number of bits.

Description

This constructor creates a `BitSet` with a size of `nbits`. All of the bits in the `BitSet` are initially set to `false`.

Instance Methods

and

public void and(BitSet set)

Parameters

`set`

The `BitSet` to AND with this `BitSet`.

Description

This method computes the logical AND of this `BitSet` and the specified `BitSet` and stores the result in this `BitSet`. In other words, for each bit in this `BitSet`, the value is set to only `true` if the bit is already `true` in this `BitSet` and the corresponding bit in `set` is `true`.

If the size of `set` is greater than the size of this `BitSet`, the extra bits in `set` are ignored. If the size of `set` is less than the size of this `BitSet`, the extra bits in this `BitSet` are set to `false`.

clear

public void clear(int bit)

Parameters

`bit`

The index of the bit to clear.

Description

This method sets the bit at the given index to `false`. If `bit` is greater than or equal to the number of bits in the `BitSet`, the size of the `BitSet` is increased so that it contains `bit` values. All of the additional bits are set to `false`.

clone

public Object clone()

Returns

A copy of this `BitSet`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `BitSet` and returns it. In other words, the returned `BitSet` has the same size as this `BitSet`, and it has the same bits set to `true`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `BitSet` and it contains the same bit values as the object this method is associated with. In other words, this method compares each bit of this `BitSet` with the corresponding bit of `obj`. If any bits do not match, the method returns `false`. If the size of this `BitSet` is different than `obj`, the extra bits in either this `BitSet` or in `obj` must be `false` for this method to return `true`.

get

public boolean get(int bit)

Parameters

`bit`

The index of the bit to retrieve.

Returns

The `boolean` value of the bit at the given index.

Description

This method returns the value of the given bit. If `bit` is greater than or equal to the number of bits in the `BitSet`, the method returns `false`.

hashCode

public int hashCode()

Returns

The hashcode for this `BitSet`.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object.

or

public void or(BitSet set)

Parameters

`set`

The `BitSet` to OR with this `BitSet`.

Description

This method computes the logical OR of this `BitSet` and the specified `BitSet`, and stores the result in this `BitSet`. In other words, for each bit in this `BitSet`, the value is set to `true` if the bit is already `true` in this `BitSet` or the corresponding bit in `set` is `true`.

If the size of `set` is greater than the size of this `BitSet`, this `BitSet` is first increased in size to accommodate the additional bits. All of the additional bits are initially set to `false`.

set

public void set(int bit)

Parameters

`bit`

The index of the bit to set.

Description

This method sets the bit at the given index to `true`. If `bit` is greater than or equal to the number of bits in the `BitSet`, the size of the `BitSet` is increased so that it contains `bit` values. All of the additional bits except the last one are set to `false`.

size

public int size()

Returns

The size of this `BitSet`.

Description

This method returns the size of this `BitSet`, which is the number of bits currently in the set.

toString

public String toString()

Returns

A string representation of this `BitSet`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `BitSet`. The string lists the indexes of all the bits in the `BitSet` that are `true`.

xor

public void xor(BitSet set)

Parameters

`set`

The `BitSet` to XOR with this `BitSet`.

Description

This method computes the logical XOR (exclusive OR) of this `BitSet` and the specified `BitSet` and stores the result in this `BitSet`. In other words, for each bit in this `BitSet`, the value is set to `true` only if the bit is already `true` in this `BitSet`, and the corresponding bit in `set` is `false`, or if the bit is `false` in this `BitSet` and the corresponding bit in `set` is `true`.

If the size of `set` is greater than the size of this `BitSet`, this `BitSet` is first increased in size to accommodate the additional bits. All of the additional bits are initially set to `false`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Cloneable`, `Serializable`

PREVIOUS
Vector

HOME
BOOK INDEX

NEXT
Calendar

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

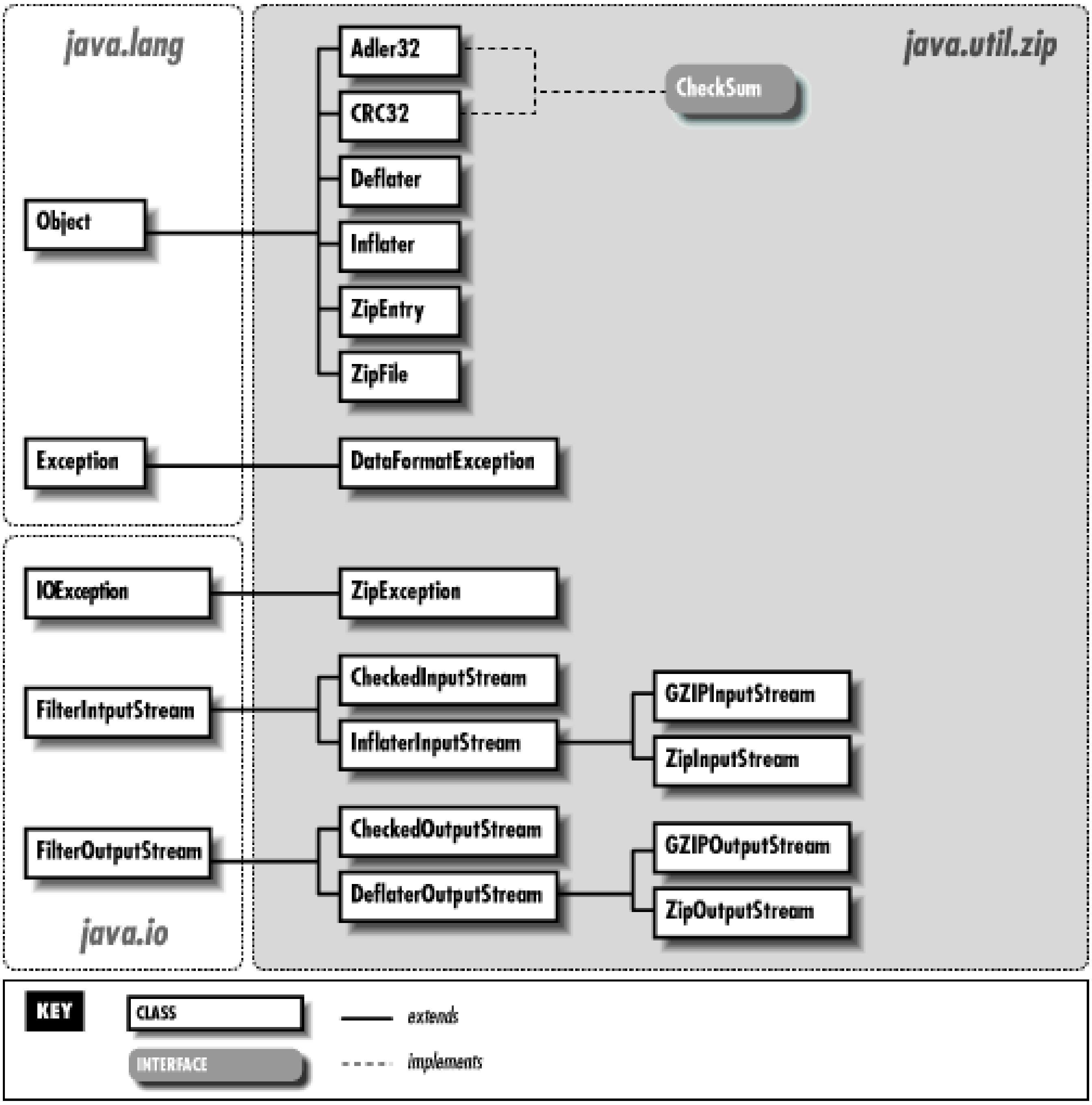
JAVA
Fundamental Classes Reference

18. The java.util.zip Package

Contents:

The package `java.util.zip` is new as of Java 1.1. It contains classes that provide support for general-purpose data compression and decompression using the ZLIB compression algorithms. The important classes in `java.util.zip` are those that provide the means to read and write data that is compatible with the popular GZIP and ZIP formats: `GZIPInputStream`, `GZIPOutputStream`, `ZipInputStream`, and `ZipOutputStream`. Figure 18.1 shows the class hierarchy for the `java.util.zip` package.

Figure 18.1: The java.text package



It is easy to use the GZIP and ZIP classes because they subclass `java.io.FilterInputStream` and `java.io.FilterOutputStream`. For example, to decompress GZIP data, you simply create a `GZIPInputStream` around the input stream that represents the compressed data. As with any `InputStream`, you could be reading from a file, a socket, or some other data source. You can then read decompressed data by calling the `read()` methods of the `GZIPInputStream`. The following code fragment creates a `GZIPInputStream` that reads data from the file *sample.gz* :

```
FileInputStream inFile;
try {
    inFile = new FileInputStream("sample.gz");
} catch (IOException e) {
    System.out.println("Couldn't open file.");
    return;
}
GZIPInputStream in = new GZIPInputStream(inFile);
```

```
// Now use in.read() to get decompressed data.
```

Similarly, you can compress data using the GZIP format by creating a `GZIPOutputStream` around an output stream and using the `write()` methods of `GZIPOutputStream`. The following code fragment creates a `GZIPOutputStream` that writes data to the file *sample.gz* :

```
FileOutputStream outFile;
try {
    outFile = new FileOutputStream("sample.gz");
} catch (IOException e) {
    System.out.println("Couldn't open file.");
    return;
}
GZIPOutputStream out = new GZIPOutputStream(outFile);
// Now use out.write() to write compressed data.
```

A ZIP file, or archive, is not quite as easy to use because it may contain more than one compressed file. A `ZipEntry` object represents each compressed file in the archive. When you are reading from a `ZipInputStream`, you must first call `getNextEntry()` to access an entry, and then you can read decompressed data from the stream, just like with a `GZIPInputStream`. When you are writing data to a `ZipOutputStream`, use `putNextEntry()` before you start writing each entry in the archive. The `ZipFile` class is provided as a convenience for reading an archive; it allows nonsequential access to the entries in a ZIP file.

The remainder of the classes in `java.util.zip` exist to support the GZIP and ZIP classes. The generic `Deflater` and `Inflater` classes implement the ZLIB algorithms; they are used by `DeflaterOutputStream` and `InflaterInputStream` to decompress and compress data. The `Checksum` interface and the classes that implement it, `Adler32` and `CRC32`, define algorithms that generate checksums from stream data. These checksums are used by the `CheckedInputStream` and `CheckedOutputStream` classes.

Adler32

Name

Adler32

Synopsis

Class Name:

```
    java.util.zip.Adler32
```

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.util.zip.Checksum`

Availability:

New as of JDK 1.1

Description

The `Adler32` class implements the `Checksum` interface using the Adler-32 algorithm. This algorithm is significantly faster than CRC-32 and almost as reliable.

Class Summary

```
public class java.util.zip.Adler32 extends java.lang.Object
             implements java.util.zip.Checksum {
  // Constructors
  public Adler32();

  // Instance Methods
  public long getValue();
  public void reset();
  public void update(int b);
  public void update(byte[] b);
  public native void update(byte[] b, int off, int len);
}
```

Constructors

Adler32

public Adler32()

Description

This constructor creates an `Adler32` object.

Instance Methods

getValue

public long getValue()

Returns

The current checksum value.

Implements

`Checksum.getValue()`

Description

This method returns the current value of this checksum.

reset

public void reset()

Implements

`Checksum.reset()`

Description

This method resets the checksum to its initial value, making it appear as though it has not been updated by any data.

update

public void update(int b)

Parameters

`b`

The value to be added to the data stream for the checksum calculation.

Implements

`Checksum.update(int)`

Description

This method adds the specified value to the data stream and updates the checksum value. The method uses only the lowest eight bits of the given `int`.

public void update(byte[] b)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

Description

This method adds the bytes from the specified array to the data stream and updates the checksum value.

public native void update(byte[] b, int off, int len)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Implements

```
Checksum.update(byte[], int, int)
```

Description

This method adds `len` bytes from the specified array, starting at `off`, to the data stream and updates the checksum value.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Checksum`, `CRC32`

PREVIOUS HOME NEXT
Vector BOOK INDEX CheckedInputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

A. The Unicode 2.0 Character Set

Characters	Description
\u0000 - \u1FFF	Alphabets
\u0020 - \u007F	Basic Latin
\u0080 - \u00FF	Latin-1 supplement
\u0100 - \u017F	Latin extended-A
\u0180 - \u024F	Latin extended-B
\u0250 - \u02AF	IPA extensions
\u02B0 - \u02FF	Spacing modifier letters
\u0300 - \u036F	Combining diacritical marks
\u0370 - \u03FF	Greek
\u0400 - \u04FF	Cyrillic
\u0530 - \u058F	Armenian
\u0590 - \u05FF	Hebrew
\u0600 - \u06FF	Arabic
\u0900 - \u097F	Devanagari
\u0980 - \u09FF	Bengali
\u0A00 - \u0A7F	Gurmukhi
\u0A80 - \u0AFF	Gujarati
\u0B00 - \u0B7F	Oriya
\u0B80 - \u0BFF	Tamil
\u0C00 - \u0C7F	Telugu
\u0C80 - \u0CFF	Kannada
\u0D00 - \u0D7F	Malayalam
\u0E00 - \u0E7F	Thai
\u0E80 - \u0EFF	Lao
\u0F00 - \u0FBF	Tibetan

`\u10A0 - \u10FF`	Georgian
`\u1100 - \u11FF`	Hangul Jamo
`\u1E00 - \u1EFF`	Latin extended additional
`\u1F00 - \u1FFF`	Greek extended
`\u2000 - \u2FFF`	Symbols and punctuation
`\u2000 - \u206F`	General punctuation
`\u2070 - \u209F`	Superscripts and subscripts
`\u20A0 - \u20CF`	Currency symbols
`\u20D0 - \u20FF`	Combining diacritical marks for symbols
`\u2100 - \u214F`	Letterlike symbols
`\u2150 - \u218F`	Number forms
`\u2190 - \u21FF`	Arrows
`\u2200 - \u22FF`	Mathematical operators
`\u2300 - \u23FF`	Miscellaneous technical
`\u2400 - \u243F`	Control pictures
`\u2440 - \u245F`	Optical character recognition
`\u2460 - \u24FF`	Enclosed alphanumerics
`\u2500 - \u257F`	Box drawing
`\u2580 - \u259F`	Block elements
`\u25A0 - \u25FF`	Geometric shapes
`\u2600 - \u26FF`	Miscellaneous symbols
`\u2700 - \u27BF`	Dingbats
`\u3000 - \u33FF`	CJK auxiliary
`\u3000 - \u303F`	CJK symbols and punctuation
`\u3040 - \u309F`	Hiragana
`\u30A0 - \u30FF`	Katakana
`\u3100 - \u312F`	Bopomofo
`\u3130 - \u318F`	Hangul compatibility Jamo
`\u3190 - \u319F`	Kanbun
`\u3200 - \u32FF`	Enclosed CJK letters and months
`\u3300 - \u33FF`	CJK compatibility
`\u4E00 - \u9FFF`	CJK unified ideographs: Han characters used in China, Japan, Korea, Taiwan, and Vietnam
`\uAC00 - \uD7A3`	Hangul syllables
`\uD800 - \uDFFF`	Surrogates
`\uD800 - \uDB7F`	High surrogates
`\uDB80 - \uDBFF`	High private use surrogates

`\uDC00`	-	`\uDFFF`	Low surrogates
`\uE000`	-	`\uF8FF`	Private use
`\uF900`	-	`\uFFFF`	Miscellaneous
`\uF900`	-	`\uFAFF`	CJK compatibility ideographs
`\uFB00`	-	`\uFB4F`	Alphabetic presentation forms
`\uFB50`	-	`\uFDFF`	Arabic presentation forms-A
`\uFE20`	-	`\uFE2F`	Combing half marks
`\uFE30`	-	`\uFE4F`	CJK compatibility forms
`\uFE50`	-	`\uFE6F`	Small form variants
`\uFE70`	-	`\uFEFE`	Arabic presentation forms-B
`\uFEFF`			Specials
`\uFF00`	-	`\uFFEF`	Halfwidth and fullwidth forms
`\uFFF0`	-	`\uFFFF`	Specials

PREVIOUS ZipOutputStream | HOME BOOK INDEX | NEXT The UTF-8 Encoding

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

JAVA Fundamental Classes Reference

B. The UTF-8 Encoding

Internally, Java always represents Unicode characters with 16 bits. However, this is an inefficient use of bits when most of the characters being used only need eight bits or less to be represented, which is the case for text written in English and a number of other languages. The UTF-8 encoding provides a more compact way of representing sequences of Unicode when most of the characters are 7-bit ASCII characters. Therefore, UTF-8 is often a more efficient way of storing or transmitting text than using 16 bits for every character.

The UTF-8 encoding is a variable-width encoding of Unicode characters. Seven-bit ASCII characters (`\u0000`-`\u007F`) are represented in one byte, so they remain untouched by the encoding (i.e., a string of ASCII characters is a legal UTF-8 string). Characters in the range `\u0080`-`\u07FF` are represented in two bytes, and characters in the range `\u0800`-`\uFFFF` are represented in three bytes. Java actually uses a slightly modified version of UTF-8, since it encodes `\u0000` using two bytes. The advantage of this approach is that a UTF-8 encoded string never contains a null character.

Java provides support for reading characters in the UTF-8 encoding with the `readUTF()` methods in `RandomAccessFile`, `DataInputStream`, and `ObjectInputStream` . The `writeUTF()` methods in `RandomAccessFile`, `DataOutputStream`, and `ObjectOutputStream` handle writing characters in the UTF-8 encoding.

The UTF-8 encoding begins with an unsigned 16-bit quantity that indicates the number of bytes of data that follow. This length value is in the format read by the `readUnsignedShort()` methods the above input classes and written by the `writeUnsignedShort()` methods in the above output classes.

The rest of the bytes are variable-length characters. A 1-byte character always has its high-order bit set to 0. A 2-byte character always begins with the high-order bits `110`, while a 3-byte character starts with the high-order bits `1110`. The second and third bytes of 2- and 3-byte characters always have their high-order bits set to `10`, which makes them easy to distinguish from 1-byte characters and the initial bytes of 2- and 3-byte characters. This encoding scheme leaves room for seven bits of data in 1-byte characters, 11 bits of data in 2-byte characters, and 16 bits of data in 3-byte characters.

The table below summarizes the UTF-8 encoding:

Bytes in Character	Minimum Character	Maximum Character	# of Data Bits	Binary Byte Sequence (x = data bit)
1	`\u0000`	`\u007F`	7	`0xxxxxxx`
2	`\u0080`	`\u07FF`	11	`110xxxxx 10xxxxxx`
3	`\u0800`	`\uFFFF`	16	`1110xxxx 10xxxxxx 10xxxxxx`

PREVIOUS
The Unicode 2.0 Character Set

HOME
BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Index