# Java Tutorials

Updated for Java SE 8


ORACLE

# Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the Java Tutorials home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your feedback.

# Legal Notices



This tutorial is a guide to developing applications for the Java Platform, Standard Edition and contains documentation (Tutorial) and sample code. The sample code made available with this Tutorial is licensed separately to you by Oracle under the Berkeley license. If you download any such sample code, you agree to the terms of the Berkeley license.

This Tutorial is provided to you by Oracle under the following license terms containing restrictions on use and disclosure and is protected by intellectual property laws. Oracle grants to you a limited, non-exclusive license to use this Tutorial for information purposes only, as an aid to learning about the Java SE platform. Except as expressly permitted in these license terms, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means this Tutorial. Reverse engineering, disassembly, or decompilation of this Tutorial is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If the Tutorial is licensed on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This Tutorial is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this Tutorial in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use.

**THE TUTORIAL IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND. ORACLE FURTHER DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT.**

**IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF ORACLE HAS BEEN**

**ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLES ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).**

# No Technical Support

Oracles technical support organization will not provide technical support, phone support, or updates to you.



The sample code and Tutorial may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

# Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

# Trail: Essential Classes

This trail discusses classes from the Java platform that are essential to most programmers.

**Exceptions** explains the exception mechanism and how it is used to handle errors and other exceptional conditions. This lesson describes what an exception is, how to throw and catch exceptions, what to do with an exception once it has been caught, and how to use the exception class hierarchy.

**Basic I/O** covers the Java platform classes used for basic input and output. It focuses primarily on *I/O Streams*, a powerful concept that greatly simplifies I/O operations. The lesson also looks at Serialization, which lets a program write whole objects out to streams and read them back again. Then the lesson looks at some file system operations, including random access files. Finally, it touchs briefly on the advanced features of the New I/O API.

**Concurrency** explains how to write applications that perform multiple tasks simultaneously. The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the `java.util.concurrent` packages.

**The Platform Environment** is defined by the underlying operating system, the Java virtual machine, the class libraries, and various configuration data supplied when the application is launched. This lesson describes some of the APIs an application uses to examine and configure its platform environment.

**Regular Expressions** are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used to search, edit, or manipulate text and data. Regular expressions vary in complexity, but once you understand the basics of how they're constructed, you'll be able to decipher (or create) any regular expression. This lesson teaches the regular expression syntax supported by the `java.util.regex` API, and presents several working examples to illustrate how the various objects interact.

---



---

# Trail: Essential Classes: Table of Contents

---



---

# Lesson: Exceptions

The Java programming language uses *exceptions* to handle errors and other exceptional events. This lesson describes when and how to use exceptions.

## What Is an Exception?

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

## The Catch or Specify Requirement

This section covers how to catch and handle exceptions. The discussion includes the `try`, `catch`, and `finally` blocks, as well as chained exceptions and logging.

## How to Throw Exceptions

This section covers the `throw` statement and the `Throwable` class and its subclasses.

## The try-with-resources Statement

This section describes the `try`-with-resources statement, which is a `try` statement that declares one or more resources. A resource is as an object that must be closed after the program is finished with it. The `try`-with-resources statement ensures that each resource is closed at the end of the statement.

## Unchecked Exceptions — The Controversy

This section explains the correct and incorrect use of the unchecked exceptions indicated by subclasses of `RuntimeException`.

## Advantages of Exceptions

The use of exceptions to manage errors has some advantages over traditional error-management techniques. You'll learn more in this section.

## Summary

## Questions and Exercises

---

**Note:** See online version of topics in this ebook to download complete source code.

---

# What Is an Exception?

The term *exception* is shorthand for the phrase "exceptional event."

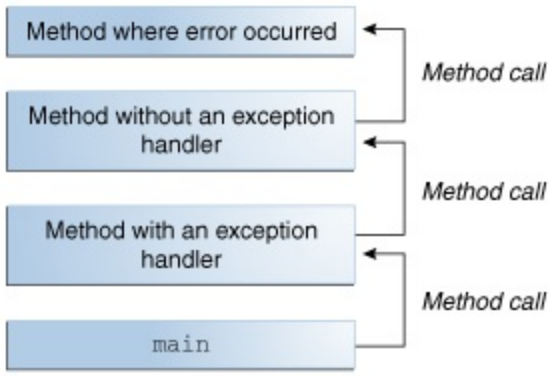
---

**Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

---

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

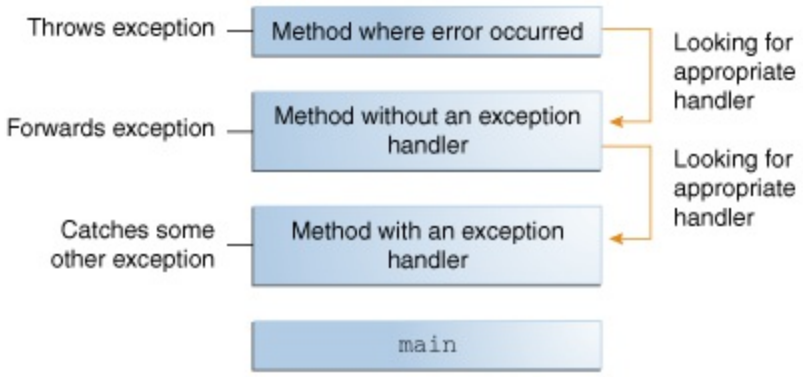
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (see the next figure).




The call stack.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.




Searching the call stack for the exception handler.

Using exceptions to manage errors has some advantages over traditional error-management techniques. You can learn more in the Advantages of Exceptions section.

# The Catch or Specify Requirement

Valid Java programming language code must honor the *Catch or Specify Requirement*. This means that code that might throw certain exceptions must be enclosed by either of the following:

- A `try` statement that catches the exception. The `try` must provide a handler for the exception, as described in Catching and Handling Exceptions.
- A method that specifies that it can throw the exception. The method must provide a `throws` clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.

Code that fails to honor the Catch or Specify Requirement will not compile.

Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

## The Three Kinds of Exceptions

The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for `java.io.FileReader`. Normally, the user provides the name of an existing, readable file, so the construction of the `FileReader` object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws `java.io.FileNotFoundException`. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by `Error`, `RuntimeException`, and their subclasses.

The second kind of exception is the *error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw `java.io.IOError`. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by `Error` and its subclasses.

The third kind of exception is the *runtime exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for `FileReader`. If a logic error causes a `null` to be passed to the constructor, the constructor will throw `NullPointerException`. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by `RuntimeException` and its subclasses.

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

## Bypassing Catch or Specify

Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended. The section Unchecked Exceptions — The Controversy talks about when it is appropriate to use unchecked exceptions.

# Catching and Handling Exceptions

This section describes how to use the three exception handler components — the `try`, `catch`, and `finally` blocks — to write an exception handler. Then, the `try`-with-resources statement, introduced in Java SE 7, is explained. The `try`-with-resources statement is particularly suited to situations that use `Closeable` resources, such as streams.

The last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named `ListOfNumbers`. When constructed, `ListOfNumbers` creates an `ArrayList` that contains 10 `Integer` elements with sequential values 0 through 9. The `ListOfNumbers` class also defines a method named `writeList`, which writes the list of numbers into a text file called `OutFile.txt`. This example uses output classes defined in `java.io`, which are covered in Basic I/O.

```
// Note: This class won't compile by design!
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an `IOException`. The second boldface line is a call to the `ArrayList` class's `get` method, which throws an `IndexOutOfBoundsException` if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the `ArrayList`).

If you try to compile the `ListOfNumbers` class, the compiler prints an error message about the exception thrown by the `FileWriter` constructor. However, it does not display an error message about the exception thrown by `get`. The reason is that the exception thrown by the constructor, `IOException`, is a checked exception, and the one thrown by the `get` method,

`IndexOutOfBoundsException`, is an unchecked exception.

Now that you're familiar with the `ListOfNumbers` class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

# The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a `try` block. In general, a `try` block looks like the following:

```
try {
    code
}
catch and finally blocks . . .
```

The segment in the example labeled *`code`* contains one or more legal lines of code that could throw an exception. (The `catch` and `finally` blocks are explained in the next two subsections.)

To construct an exception handler for the `writeList` method from the `ListOfNumbers` class, enclose the exception-throwing statements of the `writeList` method within a `try` block. There is more than one way to do this. You can put each line of code that might throw an exception within its own `try` block and provide separate exception handlers for each. Or, you can put all the `writeList` code within a single `try` block and associate multiple handlers with it. The following listing uses one `try` block for the entire method because the code in question is very short.

```
private List<Integer> list;
private static final int SIZE = 10;

PrintWriter out = null;

try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
}
catch and finally statements . . .
```

If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. To associate an exception handler with a `try` block, you must put a `catch` block after it; the next section, The catch Blocks, shows you how.

# The catch Blocks

You associate exception handlers with a `try` block by providing one or more `catch` blocks directly after the `try` block. No code can be between the end of the `try` block and the beginning of the first `catch` block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Each `catch` block is an exception handler and handles the type of exception indicated by its argument. The argument type, *`ExceptionType`*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the `Throwable` class. The handler can refer to the exception with *`name`*.

The `catch` block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose *`ExceptionType`* matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the `writeList` method — one for two types of checked exceptions that can be thrown within the `try` statement:

```
try {

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Both handlers print an error message. The second handler does nothing else. By catching any `IOException` that's not caught by the first handler, it allows the program to continue executing.

The first handler, in addition to printing a message, throws a user-defined exception. (Throwing exceptions is covered in detail later in this chapter in the How to Throw Exceptions section.)

In this example, when the `FileNotFoundException` is caught it causes a user-defined exception called `SampleException` to be thrown. You might want to do this if you want your program to handle an exception in this situation in a specific way.

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler

using chained exceptions, as described in the Chained Exceptions section.

## Catching More Than One Type of Exception with One Exception Handler

In Java SE 7 and later, a single `catch` block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the `catch` clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (`|`):

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

**Note**: If a `catch` block handles more than one exception type, then the `catch` parameter is implicitly `final`. In this example, the `catch` parameter `ex` is `final` and therefore you cannot assign any values to it within the `catch` block.

# The finally Block

The `finally` block *always* executes when the `try` block exits. This ensures that the `finally` block is executed even if an unexpected exception occurs. But `finally` is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`. Putting cleanup code in a `finally` block is always a good practice, even when no exceptions are anticipated.

---

**Note:** If the JVM exits while the `try` or `catch` code is being executed, then the `finally` block may not execute. Likewise, if the thread executing the `try` or `catch` code is interrupted or killed, the `finally` block may not execute even though the application as a whole continues.

---

The `try` block of the `writeList` method that you've been working with here opens a `PrintWriter`. The program should close that stream before exiting the `writeList` method. This poses a somewhat complicated problem because `writeList`'s `try` block can exit in one of three ways.

**1.** The `new FileWriter` statement fails and throws an `IOException`.
**2.** The `vector.elementAt(i)` statement fails and throws an `ArrayIndexOutOfBoundsException`.
**3.** Everything succeeds and the `try` block exits normally.

The runtime system always executes the statements within the `finally` block regardless of what happens within the `try` block. So it's the perfect place to perform cleanup.

The following `finally` block for the `writeList` method cleans up and then closes the `PrintWriter`.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

In the `writeList` example, you could provide for cleanup without the intervention of a `finally` block. For example, you could put the code to close the `PrintWriter` at the end of the `try` block and again within the exception handler for `ArrayIndexOutOfBoundsException`, as follows.

```
try {

    // Don't do this; it duplicates code.
    out.close();

} catch (FileNotFoundException e) {
    // Don't do this; it duplicates code.
    out.close();

    System.err.println("Caught FileNotFoundException: " + e.getMessage());
    throw new RuntimeException(e);
```

```
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

However, this duplicates code, thus making the code difficult to read and error-prone should you modify it later. For example, if you add code that can throw a new type of exception to the `try` block, you have to remember to close the `PrintWriter` within the new exception handler.

---

**Important:** The `finally` block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a `finally` block to ensure that resource is *always* recovered.

If you are using Java SE 7 or later, consider using the `try`-with-resources statement in these situations, which automatically releases system resources when no longer needed. The next section has more information.

---

# The try-with-resources Statement

The `try`-with-resources statement is a `try` statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The `try`-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

The following example reads the first line from a file. It uses an instance of `BufferedReader` to read data from the file. `BufferedReader` is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

In this example, the resource declared in the `try`-with-resources statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the `try` keyword. The class `BufferedReader`, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the `BufferedReader` instance is declared in a `try`-with-resource statement, it will be closed regardless of whether the `try` statement completes normally or abruptly (as a result of the method `BufferedReader.readLine` throwing an `IOException`).

Prior to Java SE 7, you can use a `finally` block to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly. The following example uses a `finally` block instead of a `try`-with-resources statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

However, in this example, if the methods `readLine` and `close` both throw exceptions, then the method `readFirstLineFromFileWithFinallyBlock` throws the exception thrown from the `finally` block; the exception thrown from the `try` block is suppressed. In contrast, in the example `readFirstLineFromFile`, if exceptions are thrown from both the `try` block and the `try`-with-resources statement, then the method `readFirstLineFromFile` throws the exception thrown from the `try` block; the exception thrown from the `try`-with-resources block is suppressed. In Java SE 7 and later, you can retrieve suppressed exceptions; see the section Suppressed Exceptions for more information.

You may declare one or more resources in a `try`-with-resources statement. The following example retrieves the names of the files packaged in the zip file `zipFileName` and creates a text file that contains the names of these files:

```
public static void writeToFileZipFileContents(String zipFileName,
                                               String outputFileName)
                                               throws java.io.IOException {

    java.nio.charset.Charset charset =
         java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputFilePath =
         java.nio.file.Paths.get(outputFileName);

    // Open zip file and create output file with
    // try-with-resources statement

    try (
        java.util.zip.ZipFile zf =
             new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer =
            java.nio.file.Files.newBufferedWriter(outputFilePath, charset)
    ) {
        // Enumerate each entry
        for (java.util.Enumeration entries =
                                zf.entries(); entries.hasMoreElements();) {
            // Get the entry name and write it to the output file
            String newLine = System.getProperty("line.separator");
            String zipEntryName =
                 ((java.util.zip.ZipEntry)entries.nextElement()).getName() +
                 newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

In this example, the `try`-with-resources statement contains two declarations that are separated by a semicolon: `ZipFile` and `BufferedWriter`. When the block of code that directly follows it terminates, either normally or because of an exception, the `close` methods of the `BufferedWriter` and `ZipFile` objects are automatically called in this order. Note that the `close` methods of resources are called in the *opposite* order of their creation.

The following example uses a `try`-with-resources statement to automatically close a `java.sql.Statement` object:

```
public static void viewTable(Connection con) throws SQLException {

    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";

    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");

            System.out.println(coffeeName + ", " + supplierID + ", " +
```

```
                                price + ", " + sales + ", " + total);
            }
        } catch (SQLException e) {
            JDBCTutorialUtilities.printSQLException(e);
        }
}
```

The resource `java.sql.Statement` used in this example is part of the JDBC 4.1 and later API.

**Note**: A `try`-with-resources statement can have `catch` and `finally` blocks just like an ordinary `try` statement. In a `try`-with-resources statement, any `catch` or `finally` block is run after the resources declared have been closed.

## Suppressed Exceptions

An exception can be thrown from the block of code associated with the `try`-with-resources statement. In the example `writeToFileZipFileContents`, an exception can be thrown from the `try` block, and up to two exceptions can be thrown from the `try`-with-resources statement when it tries to close the `ZipFile` and `BufferedWriter` objects. If an exception is thrown from the `try` block and one or more exceptions are thrown from the `try`-with-resources statement, then those exceptions thrown from the `try`-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the `writeToFileZipFileContents` method. You can retrieve these suppressed exceptions by calling the `Throwable.getSuppressed` method from the exception thrown by the `try` block.

## Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the `AutoCloseable` and `Closeable` interfaces for a list of classes that implement either of these interfaces. The `Closeable` interface extends the `AutoCloseable` interface. The `close` method of the `Closeable` interface throws exceptions of type `IOException` while the `close` method of the `AutoCloseable` interface throws exceptions of type `Exception`. Consequently, subclasses of the `AutoCloseable` interface can override this behavior of the `close` method to throw specialized exceptions, such as `IOException`, or no exception at all.

## Putting It All Together

The previous sections described how to construct the `try`, `catch`, and `finally` code blocks for the `writeList` method in the `ListOfNumbers` class. Now, let's walk through the code and investigate what can happen.

When all the components are put together, the `writeList` method looks like the following.

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: "
                           +  e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " +  e.getMessage());

    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

As mentioned previously, this method's `try` block has three different exit possibilities; here are two of them.

**1.** Code in the `try` statement fails and throws an exception. This could be an `IOException` caused by the `new FileWriter` statement or an `ArrayIndexOutOfBoundsException` caused by a wrong index value in the `for` loop.
**2.** Everything succeeds and the `try` statement exits normally.

Let's look at what happens in the `writeList` method during these two exit possibilities.

### Scenario 1: An Exception Occurs

The statement that creates a `FileWriter` can fail for a number of reasons. For example, the constructor for the `FileWriter` throws an `IOException` if the program cannot create or write to the file indicated.

When `FileWriter` throws an `IOException`, the runtime system immediately stops executing the `try` block; method calls being executed are not completed. The runtime system then starts searching at the

top of the method call stack for an appropriate exception handler. In this example, when the `IOException` occurs, the `FileWriter` constructor is at the top of the call stack. However, the `FileWriter` constructor doesn't have an appropriate exception handler, so the runtime system checks the next method — the `writeList` method — in the method call stack. The `writeList` method has two exception handlers: one for `IOException` and one for `ArrayIndexOutOfBoundsException`.

The runtime system checks `writeList`'s handlers in the order in which they appear after the `try` statement. The argument to the first exception handler is `ArrayIndexOutOfBoundsException`. This does not match the type of exception thrown, so the runtime system checks the next exception handler — `IOException`. This matches the type of exception that was thrown, so the runtime system ends its search for an appropriate exception handler. Now that the runtime has found an appropriate handler, the code in that `catch` block is executed.

After the exception handler executes, the runtime system passes control to the `finally` block. Code in the `finally` block executes regardless of the exception caught above it. In this scenario, the `FileWriter` was never opened and doesn't need to be closed. After the `finally` block finishes executing, the program continues with the first statement after the `finally` block.

Here's the complete output from the `ListOfNumbers` program that appears when an `IOException` is thrown.

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

The boldface code in the following listing shows the statements that get executed during this scenario:

```
public void writeList() {
   PrintWriter out = null;

    try {
        System.out.println("Entering try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: "
                           + e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

## Scenario 2: The try Block Exits Normally

In this scenario, all the statements within the scope of the `try` block execute successfully and throw no exceptions. Execution falls off the end of the `try` block, and the runtime system passes control to the `finally` block. Because everything was successful, the `PrintWriter` is open when control reaches the `finally` block, which closes the `PrintWriter`. Again, after the `finally` block finishes executing, the program continues with the first statement after the `finally` block.

Here is the output from the `ListOfNumbers` program when no exceptions are thrown.

```
Entering try statement
Closing PrintWriter
```

The boldface code in the following sample shows the statements that get executed during this scenario.

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: "
                           + e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());

    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

# Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the `writeList` method in the `ListOfNumbers` class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing the `ListOfNumbers` class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to *not* catch the exception and to allow a method further up the call stack to handle it.

If the `writeList` method doesn't catch the checked exceptions that can occur within it, the `writeList` method must specify that it can throw these exceptions. Let's modify the original `writeList` method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the `writeList` method that won't compile.

```
// Note: This method won't compile by design!
public void writeList() {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + vector.elementAt(i));
    }
    out.close();
}
```

To specify that `writeList` can throw two exceptions, add a `throws` clause to the method declaration for the `writeList` method. The `throws` clause comprises the `throws` keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
```

Remember that `ArrayIndexOutOfBoundsException` is an unchecked exception; including it in the `throws` clause is not mandatory. You could just write the following.

```
public void writeList() throws IOException {
```

# How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the `throw` statement.

As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the `Throwable` class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

You can also create *chained* exceptions. For more information, see the Chained Exceptions section.

## The throw Statement

All methods use the `throw` statement to throw an exception. The `throw` statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the `Throwable` class. Here's an example of a `throw` statement.

```
throw someThrowableObject;
```

Let's look at the `throw` statement in context. The following `pop` method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

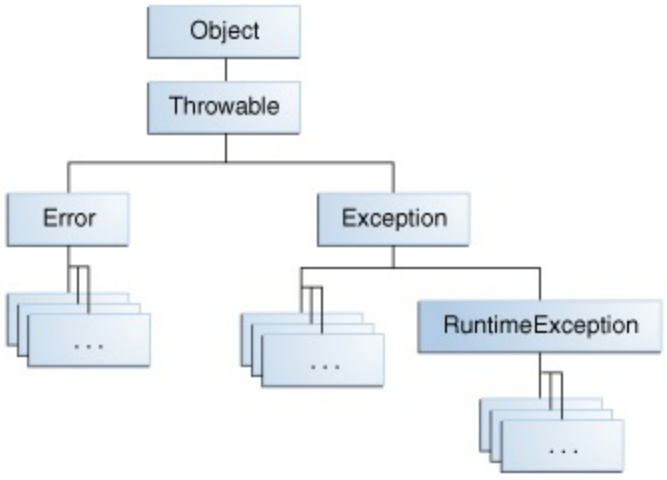
```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The `pop` method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to `0`), `pop` instantiates a new `EmptyStackException` object (a member of `java.util`) and throws it. The Creating Exception Classes section in this chapter explains how to create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the `java.lang.Throwable` class.

Note that the declaration of the `pop` method does not contain a `throws` clause. `EmptyStackException` is not a checked exception, so `pop` is not required to state that it might occur.

## Throwable Class and Its Subclasses

The objects that inherit from the `Throwable` class include direct descendants (objects that inherit directly from the `Throwable` class) and indirect descendants (objects that inherit from children or grandchildren of the `Throwable` class). The figure below illustrates the class hierarchy of the `Throwable` class and its most significant subclasses. As you can see, `Throwable` has two direct descendants: `Error` and `Exception`.




The Throwable class.

## Error Class

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an `Error`. Simple programs typically do *not* catch or throw `Error`s.

## Exception Class

Most programs throw and catch objects that derive from the `Exception` class. An `Exception` indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch `Exception`s as opposed to `Error`s.

The Java platform defines the many descendants of the `Exception` class. These descendants indicate various types of exceptions that can occur. For example, `IllegalAccessException` signals that a particular method could not be found, and `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.

One `Exception` subclass, `RuntimeException`, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is `NullPointerException`, which occurs when a method tries to access a member of an object through a `null` reference. The section Unchecked Exceptions — The Controversy discusses why most applications shouldn't throw runtime exceptions or subclass `RuntimeException`.

# Chained Exceptions

An application often responds to an exception by throwing another exception. In effect, the first exception *causes* the second exception. It can be very helpful to know when one exception causes another. *Chained Exceptions* help the programmer do this.

The following are the methods and constructors in `Throwable` that support chained exceptions.

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

The `Throwable` argument to `initCause` and the `Throwable` constructors is the exception that caused the current exception. `getCause` returns the exception that caused the current exception, and `initCause` sets the current exception's cause.

The following example shows how to use a chained exception.

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

In this example, when an `IOException` is caught, a new `SampleException` exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

## Accessing Stack Trace Information

Now let's suppose that the higher-level exception handler wants to dump the stack trace in its own format.

---

**Definition:** A *stack trace* provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.

---

The following code shows how to call the `getStackTrace` method on the exception object.

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
```

```
    }
}
```

## Logging API

The next code snippet logs where an exception occurred from within the `catch` block. However, rather than manually parsing the stack trace and sending the output to `System.err()`, it sends the output to a file using the logging facility in the `java.util.logging` package.

```
try {
    Handler handler = new FileHandler("OutFile.log");
    Logger.getLogger("").addHandler(handler);

} catch (IOException e) {
    Logger logger = Logger.getLogger("package.name");
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        logger.log(Level.WARNING, elements[i].getMethodName());
    }
}
```

# Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

- Do you need an exception type that isn't represented by those in the Java platform?
- Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?

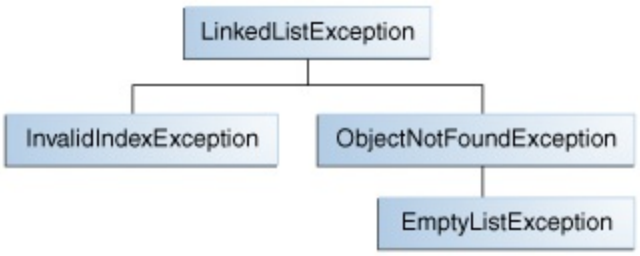
## An Example

Suppose you are writing a linked list class. The class supports the following methods, among others:

- **`objectAt(int n)`** — Returns the object in the `n`th position in the list. Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.
- **`firstObject()`** — Returns the first object in the list. Throws an exception if the list contains no objects.
- **`indexOf(Object o)`** — Searches the list for the specified `Object` and returns its position in the list. Throws an exception if the object passed into the method is not in the list.

The linked list class can throw multiple exceptions, and it would be convenient to be able to catch all exceptions thrown by the linked list with one exception handler. Also, if you plan to distribute your linked list in a package, all related code should be packaged together. Thus, the linked list should provide its own set of exception classes.

The next figure illustrates one possible class hierarchy for the exceptions thrown by the linked list.




Example exception class hierarchy.

## Choosing a Superclass

Any `Exception` subclass can be used as the parent class of `LinkedListException`. However, a quick perusal of those subclasses shows that they are inappropriate because they are either too specialized or completely unrelated to `LinkedListException`. Therefore, the parent class of `LinkedListException` should be `Exception`.

Most applets and applications you write will throw objects that are `Exception`s. `Error`s are normally used for serious, hard errors in the system, such as those that prevent the JVM from running.

---

**Note:** For readable code, it's good practice to append the string `Exception` to the names of all classes that inherit (directly or indirectly) from the `Exception` class.

---

# Unchecked Exceptions — The Controversy

Because the Java programming language does not require methods to catch or to specify unchecked exceptions (`RuntimeException`, `Error`, and their subclasses), programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from `RuntimeException`. Both of these shortcuts allow programmers to write code without bothering with compiler errors and without bothering to specify or to catch any exceptions. Although this may seem convenient to the programmer, it sidesteps the intent of the `catch` or `specify` requirement and can cause problems for others using your classes.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any `Exception` that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and `return` value.

The next question might be: "If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?" Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a `RuntimeException` is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly `null`. If an argument is `null`, the method might throw a `NullPointerException`, which is an *unchecked* exception.

Generally speaking, do not throw a `RuntimeException` or create a subclass of `RuntimeException` simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

# Advantages of Exceptions

Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

## Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To handle such cases, the `readFile` function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
```

```
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the `readFile` function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

## Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile` method is the fourth method in a series of nested method calls made by the main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`.

```
method1 {
    call method2;
}

method2 {
```

```
    call method3;
}

method3 {
    call readFile;
}
```

Suppose also that `method1` is the only method interested in the errors that might occur within `readFile`. Traditional error-notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`—the only method that is interested in them.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
```

```
}
```

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its `throws` clause.

## Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in `java.io` — `IOException` and its descendants. `IOException` is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors. For example, `FileNotFoundException` means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The `FileNotFoundException` class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
    ...
}
```

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the `catch` statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an `IOException` argument.

```
catch (IOException e) {
    ...
}
```

This handler will be able to catch all I/O exceptions, including `FileNotFoundException`, `EOFException`, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

```
catch (IOException e) {
    // Output goes to System.err.
    e.printStackTrace();
    // Send trace to stdout.
    e.printStackTrace(System.out);
}
```

You could even set up an exception handler that handles any `Exception` with the handler here.

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

The `Exception` class is close to the top of the `Throwable` class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

## Summary

A program can use exceptions to indicate that an error occurred. To throw an exception, use the `throw` statement and provide it with an exception object — a descendant of `Throwable` — to provide information about the specific error that occurred. A method that throws an uncaught, checked exception must include a `throws` clause in its declaration.

A program can catch exceptions by using a combination of the `try`, `catch`, and `finally` blocks.

- The `try` block identifies a block of code in which an exception can occur.
- The `catch` block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The `finally` block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the `try` block.

The `try` statement should contain at least one `catch` block or a `finally` block and may have multiple `catch` blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message. With exception chaining, an exception can point to the exception that caused it, which can in turn point to the exception that caused *it*, and so on.

# Questions and Exercises

## Questions

**1.** Is the following code legal?

```
try {

} finally {

}
```

**2.** What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

What is wrong with using this type of exception handler?

**3.** Is there anything wrong with the following exception handler as written? Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

**4.** Match each situation in the first list with an item in the second list.

**a.** `int[] A;`
`A[0] = 0;`

**b.** The JVM starts running your program, but the JVM can't find the Java platform classes. (The Java platform classes reside in `classes.zip` or `rt.jar`.)

**c.** A program is reading a stream and reaches the `end of stream` marker.

**d.** Before closing the stream and after reaching the `end of stream` marker, a program tries to read the stream again.

**a.** __error

**b.** __checked exception

**c.** __compile error

**d.** __no exception

## Exercises

**a.** Add a `readList` method to `ListOfNumbers.java`. This method should read in `int` values from a file, print each value, and append them to the end of the vector. You should catch all appropriate errors. You will also need a text file containing numbers to read in.

**b.** Modify the following `cat` method so that it will compile.

```
public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
```

```
            System.out.println(line);
        }
        return;
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

Check your answers.

## Questions

**1. Question:** Is the following code legal?

```
try {

} finally {

}
```

**Answer:** Yes, it's legal — and very useful A `try` statement does not have to have a `catch` block if it has a `finally` block. If the code in the `try` statement has multiple exit points and no associated `catch` clauses, the code in the `finally` block is executed no matter how the `try` block is exited. Thus it makes sense to provide a `finally` block whenever there is code that *must always* be executed. This include resource recovery code, such as the code to close I/O streams.

**2. Question:** What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

What is wrong with using this type of exception handler? **Answer:** This handler catches exceptions of type `Exception`; therefore, it catches any exception. This can be a poor implementation because you are losing valuable information about the type of exception being thrown and making your code less efficient. As a result, your program may be forced to determine the type of exception before it can decide on the best recovery strategy.

**3. Question:** Is there anything wrong with this exception handler as written? Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

**Answer:** This first handler catches exceptions of type `Exception`; therefore, it catches any exception, including `ArithmeticException`. The second handler could never be reached. This code will not compile.

**4. Question:** Match each situation in the first list with an item in the second list.

**a.** `int[] A;`
`A[0] = 0;`

**b.** The JVM starts running your program, but the JVM can't find the Java platform classes. (The Java platform classes reside in `classes.zip` or `rt.jar`.)

**c.** A program is reading a stream and reaches the `end of stream` marker.

**d.** Before closing the stream and after reaching the `end of stream` marker, a program tries to read the stream again.

**a.** __error

**b.** __checked exception

**c.** __compile error

**d.** __no exception

Answer:

**a.** 3 (compile error). The array is not initialized and will not compile.
**b.** 1 (error).
**c.** 4 (no exception). When you read a stream, you expect there to be an end of stream marker. You should use exceptions to catch unexpected behavior in your program.
**d.** 2 (checked exception).

## Exercises

**1. Exercise:** Add a `readList` method to `ListOfNumbers.java`. This method should read in `int` values from a file, print each value, and append them to the end of the vector. You should catch all appropriate errors. You will also need a text file containing numbers to read in. **Answer:** See `ListOfNumbers2.java`.
**2. Exercise:** Modify the following `cat` method so that it will compile:

```
public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

**Answer:** The code to catch exceptions is shown in bold:

```
public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } catch(FileNotFoundException fnf) {
        System.err.format("File: %s not found%n", file);
    } catch(IOException e) {
        System.err.println(e.toString());
    } finally {
        if (input != null) {
            try {
                input.close();
            } catch(IOException io) {
            }
        }
    }
}
```

# Lesson: Basic I/O

This lesson covers the Java platform classes used for basic I/O. It first focuses on *I/O Streams*, a powerful concept that greatly simplifies I/O operations. The lesson also looks at serialization, which lets a program write whole objects out to streams and read them back again. Then the lesson looks at file I/O and file system operations, including random access files.

Most of the classes covered in the `I/O Streams` section are in the `java.io` package. Most of the classes covered in the `File I/O` section are in the `java.nio.file` package.

## I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and `String` values.
- Object Streams handle binary I/O of objects.

## File I/O (Featuring NIO.2)

- What is a Path? examines the concept of a path on a file system.
- The Path Class introduces the cornerstone class of the `java.nio.file` package.
- Path Operations looks at methods in the `Path` class that deal with syntactic operations.
- File Operations introduces concepts common to many of the file I/O methods.
- Checking a File or Directory shows how to check a file's existence and its level of accessibility.
- Deleting a File or Directory.
- Copying a File or Directory.
- Moving a File or Directory.
- Managing Metadata explains how to read and set file attributes.
- Reading, Writing and Creating Files shows the stream and channel methods for reading and writing files.
- Random Access Files shows how to read or write files in a non-sequentially manner.
- Creating and Reading Directories covers API specific to directories, such as how to list a directory's contents.
- Links, Symbolic or Otherwise covers issues specific to symbolic and hard links.
- Walking the File Tree demonstrates how to recursively visit each file and directory in a file tree.
- Finding Files shows how to search for files using pattern matching.
- Watching a Directory for Changes shows how to use the watch service to detect files that are added, removed or updated in one or more directories.
- Other Useful Methods covers important API that didn't fit elsewhere in the lesson.
- Legacy File I/O Code shows how to leverage `Path` functionality if you have older code using the `java.io.File` class. A table mapping `java.io.File` API to `java.nio.file` API is provided.

## Summary

A summary of the key points covered in this trail.

## Questions and Exercises

Test what you've learned in this trail by trying these questions and exercises.

## The I/O Classes in Action

Many of the examples in the next trail, Custom Networking use the I/O streams described in this lesson to read from and write to network connections.

---

**Security consideration:** Some I/O operations are subject to approval by the current security manager. The example programs contained in these lessons are standalone applications, which by default have no security manager. To work in an applet, most of these examples would have to be modified. See What Applets Can and Cannot Do for information about the security restrictions placed on applets.

---

---

**Note:** See online version of topics in this ebook to download complete source code.

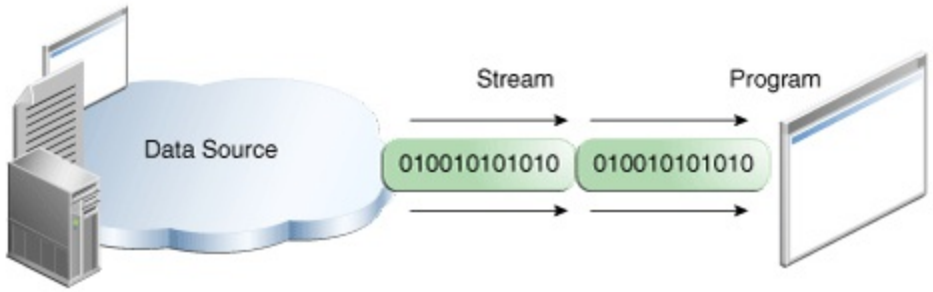
---

# I/O Streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

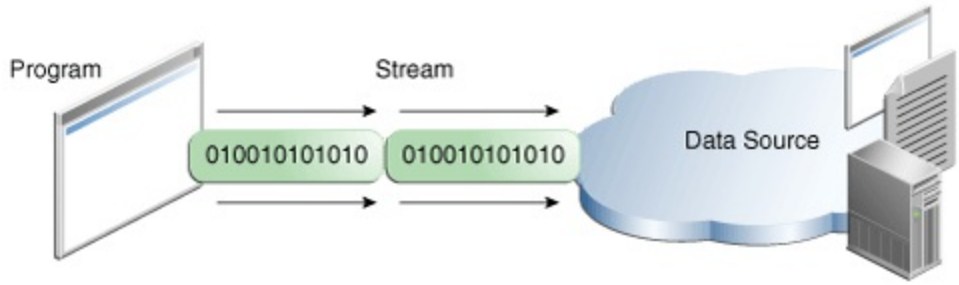
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time:




Reading information into a program.

A program uses an *output stream* to write data to a destination, one item at time:




Writing information from a program.

In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file `xanadu.txt`, which contains the following verse:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

# Byte Streams

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from `InputStream` and `OutputStream`.

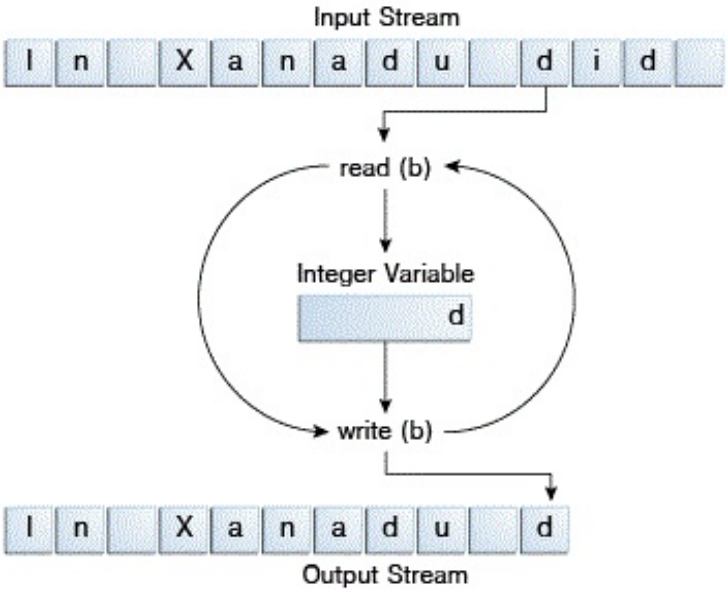
There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, `FileInputStream` and `FileOutputStream`. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

## Using Byte Streams

We'll explore `FileInputStream` and `FileOutputStream` by examining an example program named `CopyBytes`, which uses byte streams to copy `xanadu.txt`, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

`CopyBytes` spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.




Simple byte stream input and output.

## Always Close Streams

Closing a stream when it's no longer needed is very important — so important that `CopyBytes` uses a `finally` block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that `CopyBytes` was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial `null` value. That's why `CopyBytes` makes sure that each stream variable contains an object reference before invoking `close`.

## When Not to Use Byte Streams

`CopyBytes` seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since `xanadu.txt` contains character data, the best approach is to use character streams, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? Because all other stream types are built on byte streams.

# Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. See the Internationalization trail for more information.

## Using Character Streams

All character stream classes are descended from `Reader` and `Writer`. As with byte streams, there are character stream classes that specialize in file I/O: `FileReader` and `FileWriter`. The `CopyCharacters` example illustrates these classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

`CopyCharacters` is very similar to `CopyBytes`. The most important difference is that `CopyCharacters` uses `FileReader` and `FileWriter` for input and output in place of `FileInputStream` and `FileOutputStream`. Notice that both `CopyBytes` and `CopyCharacters` use an `int` variable to read to and write from. However, in `CopyCharacters`, the `int` variable holds a

character value in its last 16 bits; in `CopyBytes`, the `int` variable holds a `byte` value in its last 8 bits.

### Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. `FileReader`, for example, uses `FileInputStream`, while `FileWriter` uses `FileOutputStream`.

There are two general-purpose byte-to-character "bridge" streams: `InputStreamReader` and `OutputStreamWriter`. Use them to create character streams when there are no prepackaged character stream classes that meet your needs. The sockets lesson in the networking trail shows how to create character streams from the byte streams provided by socket classes.

## Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence (`"\r\n"`), a single carriage-return (`"\r"`), or a single line-feed (`"\n"`). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the `CopyCharacters` example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, `BufferedReader` and `PrintWriter`. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The `CopyLines` example invokes `BufferedReader.readLine` and `PrintWriter.println` to do input and output one line at a time.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
```

```
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Invoking `readLine` returns a line of text with the line. `CopyLines` outputs each line using `println`, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

There are many ways to structure text input and output beyond characters and lines. For more information, see Scanning and Formatting.

# Buffered Streams

Most of the examples we've seen so far use *unbuffered* I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the `CopyCharacters` example to use buffered I/O:

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams: `BufferedInputStream` and `BufferedOutputStream` create buffered byte streams, while `BufferedReader` and `BufferedWriter` create buffered character streams.

## Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer.

Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush `PrintWriter` object flushes the buffer on every invocation of `println` or `format`. See Formatting for more on these methods.

To flush a stream manually, invoke its `flush` method. The `flush` method is valid on any output stream, but has no effect unless the stream is buffered.

# Scanning and Formatting

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs. The scanner API breaks input into individual tokens associated with bits of data. The formatting API assembles data into nicely formatted, human-readable form.

# Scanning

Objects of type `Scanner` are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

## Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for `Character.isWhitespace`.) To see how scanning works, let's look at `ScanXan`, a program that reads the individual words in `xanadu.txt` and prints them out, one per line.

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

public class ScanXan {
    public static void main(String[] args) throws IOException {

        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Notice that `ScanXan` invokes `Scanner`'s `close` method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

The output of `ScanXan` looks like this:

```
In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome
...
```

To use a different token separator, invoke `useDelimiter()`, specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke,

```
s.useDelimiter(",\\s*");
```

## Translating Individual Tokens

The `ScanXan` example treats all input tokens as simple `String` values. `Scanner` also supports tokens for all of the Java language's primitive types (except for `char`), as well as `BigInteger` and `BigDecimal`. Also, numeric values can use thousands separators. Thus, in a `US` locale, `Scanner` correctly reads the string "32,767" as representing an integer value.

We have to mention the locale, because thousands separators and decimal symbols are locale specific. So, the following example would not work correctly in all locales if we didn't specify that the scanner should use the `US` locale. That's not something you usually have to worry about, because your input data usually comes from sources that use the same locale as you do. But this example is part of the Java Tutorial and gets distributed all over the world.

The `ScanSum` example reads a list of `double` values and adds them up. Here's the source:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

public class ScanSum {
    public static void main(String[] args) throws IOException {

        Scanner s = null;
        double sum = 0;

        try {
            s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));
            s.useLocale(Locale.US);

            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                    sum += s.nextDouble();
                } else {
                    s.next();
                }
            }
        } finally {
            s.close();
        }

        System.out.println(sum);
    }
}
```

And here's the sample input file, `usnumbers.txt`

```
8.5
32,767
3.14159
1,000,000.1
```

The output string is "1032778.74159". The period will be a different character in some locales, because `System.out` is a `PrintStream` object, and that class doesn't provide a way to override the default locale. We could override the locale for the whole program — or we could just use formatting, as described in the next topic, Formatting.

# Formatting

Stream objects that implement formatting are instances of either `PrintWriter`, a character stream class, or `PrintStream`, a byte stream class.

---

**Note:** The only `PrintStream` objects you are likely to need are `System.out` and `System.err`. (See I/O from the Command Line for more on these objects.) When you need to create a formatted output stream, instantiate `PrintWriter`, not `PrintStream`.

---

Like all byte and character stream objects, instances of `PrintStream` and `PrintWriter` implement a standard set of `write` methods for simple byte and character output. In addition, both `PrintStream` and `PrintWriter` implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:

- `print` and `println` format individual values in a standard way.
- `format` formats almost any number of values based on a format string, with many options for precise formatting.

## The `print` and `println` Methods

Invoking `print` or `println` outputs a single value after converting the value using the appropriate `toString` method. We can see this in the `Root` example:

```
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```

Here is the output of `Root`:

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```

The `i` and `r` variables are formatted twice: the first time using code in an overload of `print`, the second time by conversion code automatically generated by the Java compiler, which also utilizes `toString`. You can format any value this way, but you don't have much control over the results.

## The `format` Method

The `format` method formats multiple arguments based on a *format string*. The format string consists of static text embedded with *format specifiers*; except for the format specifiers, the format string is output unchanged.

Format strings support many features. In this tutorial, we'll just cover some basics. For a complete description, see `format string syntax` in the API specification.

The `Root2` example formats two values with a single `format` invocation:

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

Here is the output:

```
The square root of 2 is 1.414214.
```

Like the three used in this example, all format specifiers begin with a `%` and end with a 1- or 2-character *conversion* that specifies the kind of formatted output being generated. The three conversions used here are:

- `d` formats an integer value as a decimal value.
- `f` formats a floating point value as a decimal value.
- `n` outputs a platform-specific line terminator.

Here are some other conversions:

- `x` formats an integer as a hexadecimal value.
- `s` formats any value as a string.
- `tB` formats an integer as a locale-specific month name.

There are many other conversions.

---

**Note:**

Except for `%%` and `%n`, all format specifiers must match an argument. If they don't, an exception is thrown.

In the Java programming language, the `\n` escape always generates the linefeed character (`\u000A`). Don't use `\n` unless you specifically want a linefeed character. To get the correct line separator for the

local platform, use `%n`.

---

In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, `Format`, that uses every possible kind of element.

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Here's the output:

```
3.141593, +00000003.1415926536
```

The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.


Elements of a Format Specifier.

The elements must appear in the order shown. Working from the right, the optional elements are:

- **Precision**. For floating point values, this is the mathematical precision of the formatted value. For `s` and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.
- **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.
- **Flags** specify additional formatting options. In the `Format` example, the `+` flag specifies that the number should always be formatted with a sign, and the `0` flag specifies that `0` is the padding character. Other flags include `-` (pad on the right) and `,` (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
- The **Argument Index** allows you to explicitly match a designated argument. You can also specify `<` to match the same argument as the previous specifier. Thus the example could have said: `System.out.format("%f, %<+020.10f %n", Math.PI);`

# I/O from the Command Line

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

## Standard Streams

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: *Standard Input*, accessed through `System.in`; *Standard Output*, accessed through `System.out`; and *Standard Error*, accessed through `System.err`. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. For more information, refer to the documentation for your command line interpreter.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. `System.out` and `System.err` are defined as `PrintStream` objects. Although it is technically a byte stream, `PrintStream` utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, `System.in` is a byte stream with no character stream features. To use Standard Input as a character stream, wrap `System.in` in `InputStreamReader`.

```
InputStreamReader cin = new InputStreamReader(System.in);
```

## The Console

A more advanced alternative to the Standard Streams is the Console. This is a single, predefined object of type `Console` that has most of the features provided by the Standard Streams, and others besides. The Console is particularly useful for secure password entry. The Console object also provides input and output streams that are true character streams, through its `reader` and `writer` methods.

Before a program can use the Console, it must attempt to retrieve the Console object by invoking `System.console()`. If the Console object is available, this method returns it. If `System.console` returns `NULL`, then Console operations are not permitted, either because the OS doesn't support them or because the program was launched in a noninteractive environment.

The Console object supports secure password entry through its `readPassword` method. This method helps secure password entry in two ways. First, it suppresses echoing, so the password is not visible on the user's screen. Second, `readPassword` returns a character array, not a `String`, so the password

can be overwritten, removing it from memory as soon as it is no longer needed.

The `Password` example is a prototype program for changing a user's password. It demonstrates several `Console` methods.

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;

public class Password {

    public static void main (String args[]) throws IOException {

        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String login = c.readLine("Enter your login: ");
        char [] oldPassword = c.readPassword("Enter your old password: ");

        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 = c.readPassword("Enter your new password: ");
                char [] newPassword2 = c.readPassword("Enter new password again: ");
                noMatch = ! Arrays.equals(newPassword1, newPassword2);
                if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");
                } else {
                    change(login, newPassword1);
                    c.format("Password for %s changed.%n", login);
                }
                Arrays.fill(newPassword1, ' ');
                Arrays.fill(newPassword2, ' ');
            } while (noMatch);
        }

        Arrays.fill(oldPassword, ' ');
    }

    // Dummy change method.
    static boolean verify(String login, char[] password) {
        // This method always returns
        // true in this example.
        // Modify this method to verify
        // password according to your rules.
        return true;
    }

    // Dummy change method.
    static void change(String login, char[] password) {
        // Modify this method to change
        // password according to your rules.
    }
}
```

The `Password` class follows these steps:

**1.** Attempt to retrieve the Console object. If the object is not available, abort.
**2.** Invoke `Console.readLine` to prompt for and read the user's login name.

**3.** Invoke `Console.readPassword` to prompt for and read the user's existing password.
**4.** Invoke `verify` to confirm that the user is authorized to change the password. (In this example, `verify` is a dummy method that always returns `true`.)
**5.** Repeat the following steps until the user enters the same password twice:
**a.** Invoke `Console.readPassword` twice to prompt for and read a new password.
**b.** If the user entered the same password both times, invoke `change` to change it. (Again, `change` is a dummy method.)
**c.** Overwrite both passwords with blanks.
**6.** Overwrite the old password with blanks.

# Data Streams

Data streams support binary I/O of primitive data type values (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as String values. All data streams implement either the `DataInput` interface or the `DataOutput` interface. This section focuses on the most widely-used implementations of these interfaces, `DataInputStream` and `DataOutputStream`.

The `DataStreams` example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:

| Order in record | Data type | Data description | Output Method | Input Method | Sa Va |
|---|---|---|---|---|---|
| 1 | `double` | Item price | `DataOutputStream.writeDouble` | `DataInputStream.readDouble` | 19 |
| 2 | `int` | Unit count | `DataOutputStream.writeInt` | `DataInputStream.readInt` | 12 |
| 3 | `String` | Item description | `DataOutputStream.writeUTF` | `DataInputStream.readUTF` | "J T- Sh |

Let's examine crucial code in `DataStreams`. First, the program defines some constants containing the name of the data file and the data that will be written to it:

```
static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};
```

Then `DataStreams` opens an output stream. Since a `DataOutputStream` can only be created as a wrapper for an existing byte stream object, `DataStreams` provides a buffered file output byte stream.

```
out = new DataOutputStream(new BufferedOutputStream(
              new FileOutputStream(dataFile)));
```

`DataStreams` writes out the records and closes the output stream.

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
```

```
}
```

The `writeUTF` method writes out `String` values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now `DataStreams` reads the data back in again. First it must provide an input stream, and variables to hold the input data. Like `DataOutputStream`, `DataInputStream` must be constructed as a wrapper for a byte stream.

```
in = new DataInputStream(new
           BufferedInputStream(new FileInputStream(dataFile)));

double price;
int unit;
String desc;
double total = 0.0;
```

Now `DataStreams` can read each record in the stream, reporting on the data it encounters.

```
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d" + " units of %s at $%.2f%n",
            unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
}
```

Notice that `DataStreams` detects an end-of-file condition by catching `EOFException`, instead of testing for an invalid return value. All implementations of `DataInput` methods use `EOFException` instead of return values.

Also notice that each specialized `write` in `DataStreams` is exactly matched by the corresponding specialized `read`. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

`DataStreams` uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as `0.1`) do not have a binary representation.

The correct type to use for currency values is `java.math.BigDecimal`. Unfortunately, `BigDecimal` is an object type, so it won't work with data streams. However, `BigDecimal` *will* work with object streams, which are covered in the next section.

# Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface `Serializable`.

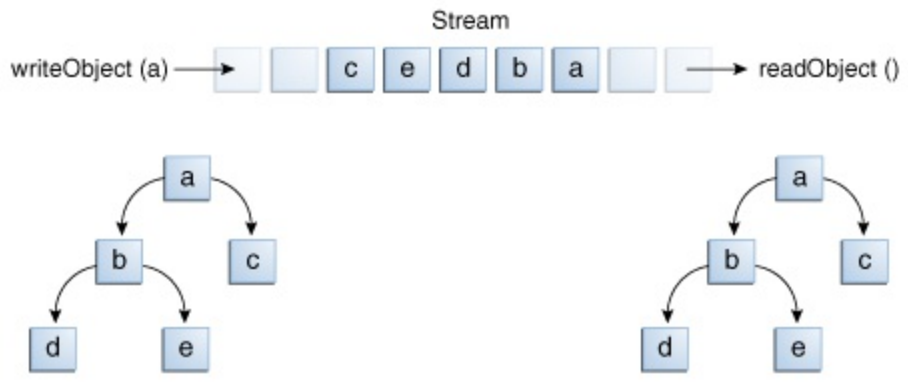
The object stream classes are `ObjectInputStream` and `ObjectOutputStream`. These classes implement `ObjectInput` and `ObjectOutput`, which are subinterfaces of `DataInput` and `DataOutput`. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values. The `ObjectStreams` example illustrates this. `ObjectStreams` creates the same application as `DataStreams`, with a couple of changes. First, prices are now `BigDecimal`objects, to better represent fractional values. Second, a `Calendar` object is written to the data file, indicating an invoice date.

If `readObject()` doesn't return the object type expected, attempting to cast it to the correct type may throw a `ClassNotFoundException`. In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding `ClassNotFoundException` to the `main` method's `throws` clause.

## Output and Input of Complex Objects

The `writeObject` and `readObject` methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If `readObject` is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, `writeObject` traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of `writeObject` can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where `writeObject` is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking `writeobject(a)` writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by `readObject`, the other four objects are read back as well, and all the original object references are preserved.



You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object `ob` twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each `writeObject` has to be matched by a `readObject`, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

This results in two variables, `ob1` and `ob2`, that are references to a single object.

However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

# File I/O (Featuring NIO.2)

---

**Note:** This tutorial reflects the file I/O mechanism introduced in the JDK 7 release. The Java SE 6 version of the File I/O tutorial was brief, but you can download the Java SE Tutorial 2008-03-14 version of the tutorial which contains the earlier File I/O content.

---

The `java.nio.file` package and its related package, `java.nio.file.attribute`, provide comprehensive support for file I/O and for accessing the default file system. Though the API has many classes, you need to focus on only a few entry points. You will see that this API is very intuitive and easy to use.

The tutorial starts by asking what is a path? Then, the Path class, the primary entry point for the package, is introduced. Methods in the `Path` class relating to syntactic operations are explained. The tutorial then moves on to the other primary class in the package, the `Files` class, which contains methods that deal with file operations. First, some concepts common to many file operations are introduced. The tutorial then covers methods for checking, deleting, copying, and moving files.

The tutorial shows how metadata is managed, before moving on to file I/O and directory I/O. Random access files are explained and issues specific to symbolic and hard links are examined.

Next, some of the very powerful, but more advanced, topics are covered. First, the capability to recursively walk the file tree is demonstrated, followed by information about how to search for files using wild cards. Next, how to watch a directory for changes is explained and demonstrated. Then, methods that didn't fit elsewhere are given some attention.

Finally, if you have file I/O code written prior to the Java SE 7 release, there is a map from the old API to the new API, as well as important information about the `File.toPath` method for developers who would like to leverage the new API without rewriting existing code.

# What Is a Path? (And Other File System Facts)

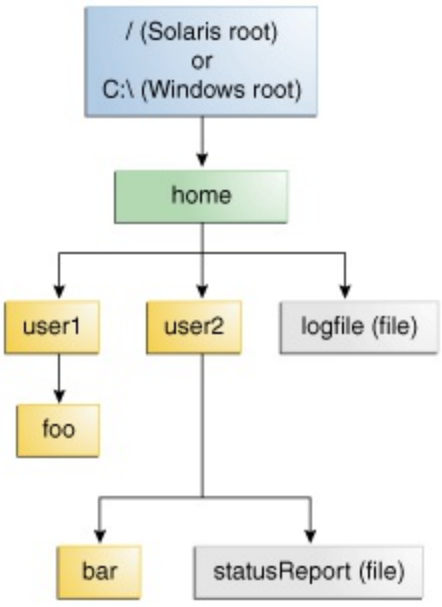
A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or *hierarchical*) structure. At the top of the tree is one (or more) root nodes. Under the root node, there are files and directories (*folders* in Microsoft Windows). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

This section covers the following:



## What Is a Path?

The following figure shows a sample directory tree containing a single root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as `C:\` or `D:\`. The Solaris OS supports a single root node, which is denoted by the slash character, `/`.




Sample Directory Structure

A file is identified by its path through the file system, beginning from the root node. For example, the `statusReport` file in the previous figure is described by the following notation in the Solaris OS:

```
/home/sally/statusReport
```

In Microsoft Windows, `statusReport` is described by the following notation:

```
C:\home\sally\statusReport
```

The character used to separate the directory names (also called the *delimiter*) is specific to the file system: The Solaris OS uses the forward slash (`/`), and Microsoft Windows uses the backslash slash (`\`).

## Relative or Absolute?

A path is either *relative* or *absolute*. An absolute path always contains the root element and the complete directory list required to locate the file. For example, `/home/sally/statusReport` is an absolute path. All of the information needed to locate the file is contained in the path string.

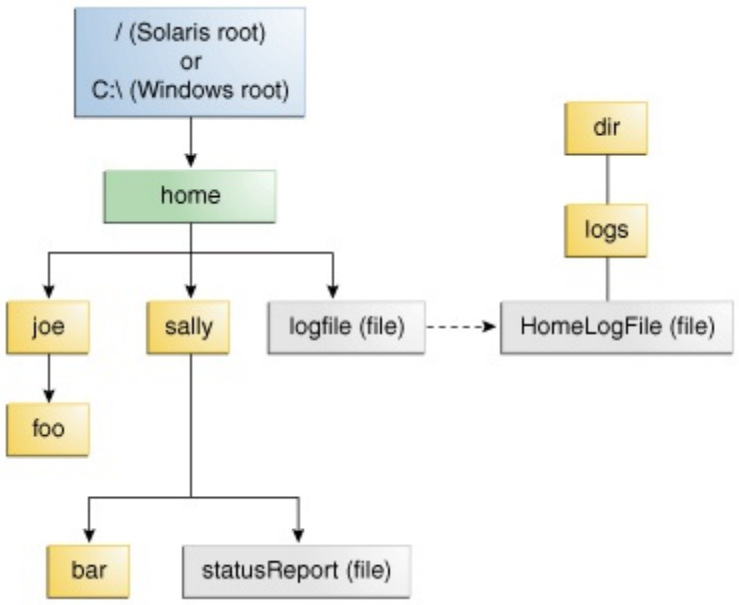
A relative path needs to be combined with another path in order to access a file. For example, `joe/foo` is a relative path. Without more information, a program cannot reliably locate the `joe/foo` directory in the file system.

## Symbolic Links

File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a *symlink* or a *soft link*.

A *symbolic link* is a special file that serves as a reference to another file. For the most part, symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link. (The file or directory being pointed to is called the *target* of the link.) Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.

In the following figure, `logFile` appears to be a regular file to the user, but it is actually a symbolic link to `dir/logs/HomeLogFile`. `HomeLogFile` is the target of the link.




Example of a Symbolic Link.

A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

The phrase *resolving a link* means to substitute the actual location in the file system for the symbolic link. In the example, resolving `logFile` yields `dir/logs/HomeLogFile`.

In real-world scenarios, most file systems make liberal use of symbolic links. Occasionally, a carelessly created symbolic link can cause a circular reference. A circular reference occurs when the target of a link points back to the original link. The circular reference might be indirect: directory `a` points to directory `b`, which points to directory `c`, which contains a subdirectory pointing back to directory `a`. Circular references can cause havoc when a program is recursively walking a directory structure. However, this scenario has been accounted for and will not cause your program to loop infinitely.

The next page discusses the heart of file I/O support in the Java programming language, the `Path` class.

# The Path Class

The `Path` class, introduced in the Java SE 7 release, is one of the primary entrypoints of the `java.nio.file` package. If your application uses file I/O, you will want to learn about the powerful features of this class.

---

**Version Note:** If you have pre-JDK7 code that uses `java.io.File`, you can still take advantage of the `Path` class functionality by using the `File.toPath` method. See Legacy File I/O Code for more information.

---

As its name implies, the `Path` class is a programmatic representation of a path in the file system. A `Path` object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files.

A `Path` instance reflects the underlying platform. In the Solaris OS, a `Path` uses the Solaris syntax (`/home/joe/foo`) and in Microsoft Windows, a `Path` uses the Windows syntax (`C:\home\joe\foo`). A `Path` is not system independent. You cannot compare a `Path` from a Solaris file system and expect it to match a `Path` from a Windows file system, even if the directory structure is identical and both instances locate the same relative file.

The file or directory corresponding to the `Path` might not exist. You can create a `Path` instance and manipulate it in various ways: you can append to it, extract pieces of it, compare it to another path. At the appropriate time, you can use the methods in the `Files` class to check the existence of the file corresponding to the `Path`, create the file, open it, delete it, change its permissions, and so on.

The next page examines the `Path` class in detail.

# Path Operations

The `Path` class includes various methods that can be used to obtain information about the path, access elements of the path, convert the path to other forms, or extract portions of a path. There are also methods for matching the path string and methods for removing redundancies in a path. This lesson addresses these `Path` methods, sometimes called *syntactic* operations, because they operate on the path itself and don't access the file system.

This section covers the following:



## Creating a Path

A `Path` instance contains the information used to specify the location of a file or directory. At the time it is defined, a `Path` is provided with a series of one or more names. A root element or a file name might be included, but neither are required. A `Path` might consist of just a single directory or file name.

You can easily create a `Path` object by using one of the following `get` methods from the `Paths` (note the plural) helper class:

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

The `Paths.get` method is shorthand for the following code:

```
Path p4 = FileSystems.getDefault().getPath("/users/sally");
```

The following example creates `/u/joe/logs/foo.log` assuming your home directory is `/u/joe`, or `C:\joe\logs\foo.log` if you are on Windows.

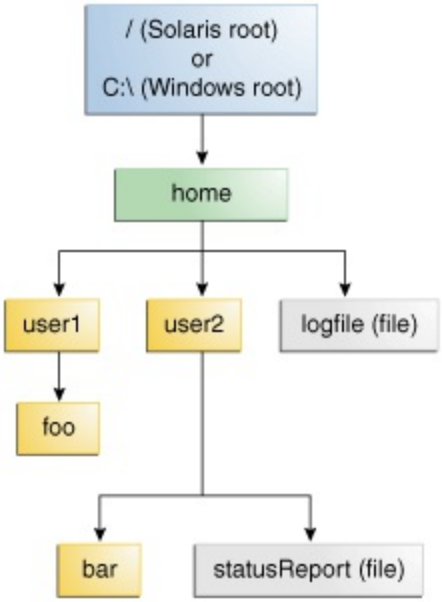
```
Path p5 = Paths.get(System.getProperty("user.home"),"logs", "foo.log");
```

## Retrieving Information about a Path

You can think of the `Path` as storing these name elements as a sequence. The highest element in the directory structure would be located at index 0. The lowest element in the directory structure would

be located at index `[n-1]`, where `n` is the number of name elements in the `Path`. Methods are available for retrieving individual elements or a subsequence of the `Path` using these indexes.

The examples in this lesson use the following directory structure.




Sample Directory Structure

The following code snippet defines a `Path` instance and then invokes several methods to obtain information about the path:

```
// None of these methods requires that the file corresponding
// to the Path exists.
// Microsoft Windows syntax
Path path = Paths.get("C:\\home\\joe\\foo");

// Solaris syntax
Path path = Paths.get("/home/joe/foo");

System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

Here is the output for both Windows and the Solaris OS:

| Method Invoked | Returns in the Solaris OS | Returns in Microsoft Windows | Comment |
|---|---|---|---|
| | | | Returns the string representation of the `Path`. If t path was created using `Filesystems.getDefault().getPath(Strin` |

| | | | |
|---|---|---|---|
| `toString` | `/home/joe/foo` | `C:\home\joe\foo` | or `Paths.get` (the latter is a convenience metho for `getPath`), the method performs minor syntactic cleanup. For example, in a UNIX operating system, it will correct the input string `//home/joe/foo` to `/home/joe/foo`. |
| `getFileName` | `foo` | `foo` | Returns the file name or the last element of the sequence of name elements. |
| `getName(0)` | `home` | `home` | Returns the path element corresponding to the specified index. The 0th element is the path element closest to the root. |
| `getNameCount` | `3` | `3` | Returns the number of elements in the path. |
| `subpath(0,2)` | `home/joe` | `home\joe` | Returns the subsequence of the `Path` (not includi a root element) as specified by the beginning anc ending indexes. |
| `getParent` | `/home/joe` | `\home\joe` | Returns the path of the parent directory. |
| `getRoot` | `/` | `C:\` | Returns the root of the path. |

The previous example shows the output for an absolute path. In the following example, a relative path is specified:

```
// Solaris syntax
Path path = Paths.get("sally/bar");
or
// Microsoft Windows syntax
Path path = Paths.get("sally\\bar");
```

Here is the output for Windows and the Solaris OS:

| Method Invoked | Returns in the Solaris OS | Returns in Microsoft Windows |
|---|---|---|
| `toString` | `sally/bar` | `sally\bar` |
| `getFileName` | `bar` | `bar` |
| `getName(0)` | `sally` | `sally` |
| `getNameCount` | `2` | `2` |
| `subpath(0,1)` | `sally` | `sally` |
| `getParent` | `sally` | `sally` |
| `getRoot` | `null` | `null` |

## Removing Redundancies From a Path

Many file systems use "." notation to denote the current directory and ".." to denote the parent directory. You might have a situation where a `Path` contains redundant directory information. Perhaps a server is configured to save its log files in the "`/dir/logs/.`" directory, and you want to delete the

trailing "`/.`" notation from the path.

The following examples both include redundancies:

```
/home/./joe/foo
/home/sally/../joe/foo
```

The `normalize` method removes any redundant elements, which includes any "`.`" or "*`directory`*`/..`" occurrences. Both of the preceding examples normalize to `/home/joe/foo`.

It is important to note that `normalize` doesn't check at the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if `sally` were a symbolic link, removing `sally/..` might result in a `Path` that no longer locates the intended file.

To clean up a path while ensuring that the result locates the correct file, you can use the `toRealPath` method. This method is described in the next section, Converting a Path.

## Converting a Path

You can use three methods to convert the `Path`. If you need to convert the path to a string that can be opened from a browser, you can use `toUri`. For example:

```
Path p1 = Paths.get("/home/logfile");
// Result is file:///home/logfile
System.out.format("%s%n", p1.toUri());
```

The `toAbsolutePath` method converts a path to an absolute path. If the passed-in path is already absolute, it returns the same `Path` object. The `toAbsolutePath` method can be very helpful when processing user-entered file names. For example:

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

        if (args.length < 1) {
            System.out.println("usage: FileTest file");
            System.exit(-1);
        }

        // Converts the input string to a Path object.
        Path inputPath = Paths.get(args[0]);

        // Converts the input Path
        // to an absolute path.
        // Generally, this means prepending
        // the current working
        // directory.  If this example
        // were called like this:
        //     java FileTest foo
        // the getRoot and getParent methods
        // would return null
        // on the original "inputPath"
        // instance.  Invoking getRoot and
        // getParent on the "fullPath"
```

```
        // instance returns expected values.
        Path fullPath = inputPath.toAbsolutePath();
    }
}
```

The `toAbsolutePath` method converts the user input and returns a `Path` that returns useful values when queried. The file does not need to exist for this method to work.

The `toRealPath` method returns the *real* path of an existing file. This method performs several operations in one:

- If `true` is passed to this method and the file system supports symbolic links, this method resolves any symbolic links in the path.
- If the `Path` is relative, it returns an absolute path.
- If the `Path` contains any redundant elements, it returns a path with those elements removed.

This method throws an exception if the file does not exist or cannot be accessed. You can catch the exception when you want to handle any of these cases. For example:

```
try {
    Path fp = path.toRealPath();
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
    // Logic for case when file doesn't exist.
} catch (IOException x) {
    System.err.format("%s%n", x);
    // Logic for other sort of file error.
}
```

## Joining Two Paths

You can combine paths by using the `resolve` method. You pass in a *partial path* , which is a path that does not include a root element, and that partial path is appended to the original path.

For example, consider the following code snippet:

```
// Solaris
Path p1 = Paths.get("/home/joe/foo");
// Result is /home/joe/foo/bar
System.out.format("%s%n", p1.resolve("bar"));

or

// Microsoft Windows
Path p1 = Paths.get("C:\\home\\joe\\foo");
// Result is C:\home\joe\foo\bar
System.out.format("%s%n", p1.resolve("bar"));
```

Passing an absolute path to the `resolve` method returns the passed-in path:

```
// Result is /home/joe
Paths.get("foo").resolve("/home/joe");
```

## Creating a Path Between Two Paths

A common requirement when you are writing file I/O code is the capability to construct a path from one location in the file system to another location. You can meet this using the `relativize` method. This method constructs a path originating from the original path and ending at the location specified by the passed-in path. The new path is *relative* to the original path.

For example, consider two relative paths defined as `joe` and `sally`:

```
Path p1 = Paths.get("joe");
Path p2 = Paths.get("sally");
```

In the absence of any other information, it is assumed that `joe` and `sally` are siblings, meaning nodes that reside at the same level in the tree structure. To navigate from `joe` to `sally`, you would expect to first navigate one level up to the parent node and then down to `sally`:

```
// Result is ../sally
Path p1_to_p2 = p1.relativize(p2);
// Result is ../joe
Path p2_to_p1 = p2.relativize(p1);
```

Consider a slightly more complicated example:

```
Path p1 = Paths.get("home");
Path p3 = Paths.get("home/sally/bar");
// Result is sally/bar
Path p1_to_p3 = p1.relativize(p3);
// Result is ../..
Path p3_to_p1 = p3.relativize(p1);
```

In this example, the two paths share the same node, `home`. To navigate from `home` to `bar`, you first navigate one level down to `sally` and then one more level down to `bar`. Navigating from `bar` to `home` requires moving up two levels.

A relative path cannot be constructed if only one of the paths includes a root element. If both paths include a root element, the capability to construct a relative path is system dependent.

The recursive `Copy` example uses the `relativize` and `resolve` methods.

## Comparing Two Paths

The `Path` class supports `equals`, enabling you to test two paths for equality. The `startsWith` and `endsWith` methods enable you to test whether a path begins or ends with a particular string. These methods are easy to use. For example:

```
Path path = ...;
```

```
Path otherPath = ...;
Path beginning = Paths.get("/home");
Path ending = Paths.get("foo");

if (path.equals(otherPath)) {
    // equality logic here
} else if (path.startsWith(beginning)) {
    // path begins with "/home"
} else if (path.endsWith(ending)) {
    // path ends with "foo"
}
```

The `Path` class implements the `Iterable` interface. The `iterator` method returns an object that enables you to iterate over the name elements in the path. The first element returned is that closest to the root in the directory tree. The following code snippet iterates over a path, printing each name element:

```
Path path = ...;
for (Path name: path) {
    System.out.println(name);
}
```

The `Path` class also implements the `Comparable` interface. You can compare `Path` objects by using `compareTo` which is useful for sorting.

You can also put `Path` objects into a `Collection`. See the Collections trail for more information about this powerful feature.

When you want to verify that two `Path` objects locate the same file, you can use the `isSameFile` method, as described in Checking Whether Two Paths Locate the Same File.

# File Operations

The `Files` class is the other primary entrypoint of the `java.nio.file` package. This class offers a rich set of static methods for reading, writing, and manipulating files and directories. The `Files` methods work on instances of `Path` objects. Before proceeding to the remaining sections, you should familiarize yourself with the following common concepts:

- Releasing System Resources
- Catching Exceptions
- Varargs
- Atomic Operations
- Method Chaining
- What *Is* a Glob?
- Link Awareness

## Releasing System Resources

Many of the resources that are used in this API, such as streams or channels, implement or extend the `java.io.Closeable` interface. A requirement of a `Closeable` resource is that the `close` method must be invoked to release the resource when no longer required. Neglecting to close a resource can have a negative implication on an application's performance. The `try`-with-resources statement, described in the next section, handles this step for you.

## Catching Exceptions

With file I/O, unexpected conditions are a fact of life: a file exists (or doesn't exist) when expected, the program doesn't have access to the file system, the default file system implementation does not support a particular function, and so on. Numerous errors can be encountered.

All methods that access the file system can throw an `IOException`. It is best practice to catch these exceptions by embedding these methods into a `try`-with-resources statement, introduced in the Java SE 7 release. The `try`-with-resources statement has the advantage that the compiler automatically generates the code to close the resource(s) when no longer required. The following code shows how this might look:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

For more information, see The try-with-resources Statement.

Alternatively, you can embed the file I/O methods in a `try` block and then catch any exceptions in a `catch` block. If your code has opened any streams or channels, you should close them in a `finally`

block. The previous example would look something like the following using the try-catch-finally approach:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
} finally {
    if (writer != null) writer.close();
}
```

For more information, see Catching and Handling Exceptions.

In addition to `IOException`, many specific exceptions extend `FileSystemException`. This class has some useful methods that return the file involved (`getFile`), the detailed message string (`getMessage`), the reason why the file system operation failed (`getReason`), and the "other" file involved, if any (`getOtherFile`).

The following code snippet shows how the `getFile` method might be used:

```
try (...) {
    ...
} catch (NoSuchFileException x) {
    System.err.format("%s does not exist\n", x.getFile());
}
```

For purposes of clarity, the file I/O examples in this lesson may not show exception handling, but your code should always include it.

## Varargs

Several `Files` methods accept an arbitrary number of arguments when flags are specified. For example, in the following method signature, the ellipses notation after the `CopyOption` argument indicates that the method accepts a variable number of arguments, or *varargs*, as they are typically called:

```
Path Files.move(Path, Path, CopyOption...)
```

When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (`CopyOption[]`) of values.

In the `move` example, the method can be invoked as follows:

```
import static java.nio.file.StandardCopyOption.*;
```

```
Path source = ...;
Path target = ...;
Files.move(source,
           target,
           REPLACE_EXISTING,
           ATOMIC_MOVE);
```

For more information about varargs syntax, see Arbitrary Number of Arguments.

## Atomic Operations

Several `Files` methods, such as `move`, can perform certain operations atomically in some file systems.

An *atomic file operation* is an operation that cannot be interrupted or "partially" performed. Either the entire operation is performed or the operation fails. This is important when you have multiple processes operating on the same area of the file system, and you need to guarantee that each process accesses a complete file.

## Method Chaining

Many of the file I/O methods support the concept of *method chaining*.

You first invoke a method that returns an object. You then immediately invoke a method on *that* object, which returns yet another object, and so on. Many of the I/O examples use the following technique:

```
String value = Charset.defaultCharset().decode(buf).toString();
UserPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService().
         lookupPrincipalByName("me");
```

This technique produces compact code and enables you to avoid declaring temporary variables that you don't need.

## What *Is* a Glob?

Two methods in the `Files` class accept a glob argument, but what is a *glob*?

You can use glob syntax to specify pattern-matching behavior.

A glob pattern is specified as a string and is matched against other strings, such as directory or file names. Glob syntax follows several simple rules:

- An asterisk, `*`, matches any number of characters (including none).
- Two asterisks, `**`, works like `*` but crosses directory boundaries. This syntax is generally used for matching complete paths.
- A question mark, `?`, matches exactly one character.

- Braces specify a collection of subpatterns. For example:
  - `{sun,moon,stars}` matches "sun", "moon", or "stars".
  - `{temp*,tmp*}` matches all strings beginning with "temp" or "tmp".
- Square brackets convey a set of single characters or, when the hyphen character (-) is used, a range of characters. For example:
  - `[aeiou]` matches any lowercase vowel.
  - `[0-9]` matches any digit.
  - `[A-Z]` matches any uppercase letter.
  - `[a-z,A-Z]` matches any uppercase or lowercase letter.

  Within the square brackets, `*`, `?`, and `\` match themselves.
- All other characters match themselves.
- To match `*`, `?`, or the other special characters, you can escape them by using the backslash character, `\`. For example: `\\` matches a single backslash, and `\?` matches the question mark.

Here are some examples of glob syntax:

- `*.html` Matches all strings that end in *.html*
- `???` Matches all strings with exactly three letters or digits
- `*[0-9]*` Matches all strings containing a numeric value
- `*.{htm,html,pdf}` Matches any string ending with *.htm*, *.html* or *.pdf*
- `a?*.java` Matches any string beginning with `a`, followed by at least one letter or digit, and ending with *.java*
- `{foo*,*[0-9]*}` Matches any string beginning with *foo* or any string containing a numeric value

---

**Note:** If you are typing the glob pattern at the keyboard and it contains one of the special characters, you must put the pattern in quotes (`"*"`), use the backslash (`\*`), or use whatever escape mechanism is supported at the command line.

---

The glob syntax is powerful and easy to use. However, if it is not sufficient for your needs, you can also use a regular expression. For more information, see the Regular Expressions lesson.

For more information about the glob sytnax, see the API specification for the `getPathMatcher` method in the `FileSystem` class.

## Link Awareness

The `Files` class is "link aware." Every `Files` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

# Checking a File or Directory

You have a `Path` instance representing a file or directory, but does that file exist on the file system? Is it readable? Writable? Executable?

## Verifying the Existence of a File or Directory

The methods in the `Path` class are syntactic, meaning that they operate on the `Path` instance. But eventually you must access the file system to verify that a particular `Path` exists, or does not exist. You can do so with the `exists(Path, LinkOption...)` and the `notExists(Path, LinkOption...)` methods. Note that `!Files.exists(path)` is not equivalent to `Files.notExists(path)`. When you are testing a file's existence, three results are possible:

- The file is verified to exist.
- The file is verified to not exist.
- The file's status is unknown. This result can occur when the program does not have access to the file.

If both `exists` and `notExists` return `false`, the existence of the file cannot be verified.

## Checking File Accessibility

To verify that the program can access a file as needed, you can use the `isReadable(Path)`, `isWritable(Path)`, and `isExecutable(Path)` methods.

The following code snippet verifies that a particular file exists and that the program has the ability to execute the file.

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
         Files.isReadable(file) & Files.isExecutable(file);
```

---

**Note:** Once any of these methods completes, there is no guarantee that the file can be accessed. A common security flaw in many applications is to perform a check and then access the file. For more information, use your favorite search engine to look up `TOCTTOU` (pronounced *TOCK-too*).

---

## Checking Whether Two Paths Locate the Same File

When you have a file system that uses symbolic links, it is possible to have two different paths that locate the same file. The `isSameFile(Path, Path)` method compares two paths to determine if they locate the same file on the file system. For example:

```
Path p1 = ...;
Path p2 = ...;

if (Files.isSameFile(p1, p2)) {
    // Logic when the paths locate the same file
```

```
}
```

# Deleting a File or Directory

You can delete files, directories or links. With symbolic links, the link is deleted and not the target of the link. With directories, the directory must be empty, or the deletion fails.

The `Files` class provides two deletion methods.

The `delete(Path)` method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist a `NoSuchFileException` is thrown. You can catch the exception to determine why the delete failed as follows:

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

The `deleteIfExists(Path)` method also deletes the file, but if the file does not exist, no exception is thrown. Failing silently is useful when you have multiple threads deleting files and you don't want to throw an exception just because one thread did so first.

# Copying a File or Directory

You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method. The copy fails if the target file exists, unless the `REPLACE_EXISTING` option is specified.

Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.

When copying a symbolic link, the target of the link is copied. If you want to copy the link itself, and not the contents of the link, specify either the `NOFOLLOW_LINKS` or `REPLACE_EXISTING` option.

This method takes a varargs argument. The following `StandardCopyOption` and `LinkOption` enums are supported:

- `REPLACE_EXISTING` Performs the copy even when the target file already exists. If the target is a symbolic link, the link itself is copied (and not the target of the link). If the target is a non-empty directory, the copy fails with the `FileAlreadyExistsException` exception.
- `COPY_ATTRIBUTES` Copies the file attributes associated with the file to the target file. The exact file attributes supported are file system and platform dependent, but `last-modified-time` is supported across platforms and is copied to the target file.
- `NOFOLLOW_LINKS` Indicates that symbolic links should not be followed. If the file to be copied is a symbolic link, the link is copied (and not the target of the link).

If you are not familiar with `enums`, see Enum Types.

The following shows how to use the `copy` method:

```
import static java.nio.file.StandardCopyOption.*;
...
Files.copy(source, target, REPLACE_EXISTING);
```

In addition to file copy, the `Files` class also defines methods that may be used to copy between a file and a stream. The `copy(InputStream, Path, CopyOptions...)` method may be used to copy all bytes from an input stream to a file. The `copy(Path, OutputStream)` method may be used to copy all bytes from a file to an output stream.

The `Copy` example uses the `copy` and `Files.walkFileTree` methods to support a recursive copy. See Walking the File Tree for more information.

# Moving a File or Directory

You can move a file or directory by using the `move(Path, Path, CopyOption...)` method. The move fails if the target file exists, unless the `REPLACE_EXISTING` option is specified.

Empty directories can be moved. If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory. On UNIX systems, moving a directory within the same partition generally consists of renaming the directory. In that situation, this method works even when the directory contains files.

This method takes a varargs argument the following `StandardCopyOption` enums are supported:

- `REPLACE_EXISTING` Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.
- `ATOMIC_MOVE` Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

The following shows how to use the `move` method:

```
import static java.nio.file.StandardCopyOption.*;
...
Files.move(source, target, REPLACE_EXISTING);
```

Though you can implement the `move` method on a single directory as shown, the method is most often used with the file tree recursion mechanism. For more information, see Walking the File Tree.

# Managing Metadata (File and File Store Attributes)

The definition of *metadata* is "data about other data." With a file system, the data is contained in its files and directories, and the metadata tracks information about each of these objects: Is it a regular file, a directory, or a link? What is its size, creation date, last modified date, file owner, group owner, and access permissions?

A file system's metadata is typically referred to as its *file attributes*. The `Files` class includes methods that can be used to obtain a single attribute of a file, or to set an attribute.

| Methods | Comment |
|---|---|
| `size(Path)` | Returns the size of the specified file in bytes. |
| `isDirectory(Path, LinkOption)` | Returns true if the specified `Path` locates a file that is a directory. |
| `isRegularFile(Path, LinkOption...)` | Returns true if the specified `Path` locates a file that is a regular file. |
| `isSymbolicLink(Path)` | Returns true if the specified `Path` locates a file that is a symbolic link. |
| `isHidden(Path)` | Returns true if the specified `Path` locates a file that is considered hidden by the file system. |
| `getLastModifiedTime(Path, LinkOption...)` `setLastModifiedTime(Path, FileTime)` | Returns or sets the specified file's last modified time. |
| `getOwner(Path, LinkOption...)` `setOwner(Path, UserPrincipal)` | Returns or sets the owner of the file. |
| `getPosixFilePermissions(Path, LinkOption...)` `setPosixFilePermissions(Path, Set<PosixFilePermission>)` | Returns or sets a file's POSIX file permissions. |
| `getAttribute(Path, String, LinkOption...)` `setAttribute(Path, String, Object, LinkOption...)` | Returns or sets the value of a file attribute. |

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the `Files` class provides two `readAttributes` methods to fetch a file's attributes in one bulk operation.

| Method | Comment |
|---|---|
| `readAttributes(Path,` | Reads a file's attributes as a bulk operation. The `String` parameter |

| | |
|---|---|
| String, LinkOption...) | identifies the attributes to be read. |
| readAttributes(Path, Class<A>, LinkOption...) | Reads a file's attributes as a bulk operation. The Class<A> parameter is the type of attributes requested and the method returns an object of that class. |

Before showing examples of the readAttributes methods, it should be mentioned that different file systems have different notions about which attributes should be tracked. For this reason, related file attributes are grouped together into views. A *view* maps to a particular file system implementation, such as POSIX or DOS, or to a common functionality, such as file ownership.

The supported views are as follows:

- BasicFileAttributeView Provides a view of basic attributes that are required to be supported by all file system implementations.
- DosFileAttributeView Extends the basic attribute view with the standard four bits supported on file systems that support the DOS attributes.
- PosixFileAttributeView Extends the basic attribute view with attributes supported on file systems that support the POSIX family of standards, such as UNIX. These attributes include file owner, group owner, and the nine related access permissions.
- FileOwnerAttributeView Supported by any file system implementation that supports the concept of a file owner.
- AclFileAttributeView Supports reading or updating a file's Access Control Lists (ACL). The NFSv4 ACL model is supported. Any ACL model, such as the Windows ACL model, that has a well-defined mapping to the NFSv4 model might also be supported.
- UserDefinedFileAttributeView Enables support of metadata that is user defined. This view can be mapped to any extension mechanisms that a system supports. In the Solaris OS, for example, you can use this view to store the MIME type of a file.

A specific file system implementation might support only the basic file attribute view, or it may support several of these file attribute views. A file system implementation might support other attribute views not included in this API.

In most instances, you should not have to deal directly with any of the FileAttributeView interfaces. (If you do need to work directly with the FileAttributeView, you can access it via the getFileAttributeView(Path, Class<V>, LinkOption...) method.)

The readAttributes methods use generics and can be used to read the attributes for any of the file attributes views. The examples in the rest of this page use the readAttributes methods.

The remainder of this section covers the following topics:

- Basic File Attributes
- Setting Time Stamps
- DOS File Attributes
- POSIX File Permissions

- Setting a File or Group Owner
- User-Defined File Attributes
- File Store Attributes

## Basic File Attributes

As mentioned previously, to read the basic attributes of a file, you can use one of the `Files.readAttributes` methods, which reads all the basic attributes in one bulk operation. This is far more efficient than accessing the file system separately to read each individual attribute. The varargs argument currently supports the `LinkOption` enum, `NOFOLLOW_LINKS`. Use this option when you do not want symbolic links to be followed.

---

**A word about time stamps:** The set of basic attributes includes three time stamps: `creationTime`, `lastModifiedTime`, and `lastAccessTime`. Any of these time stamps might not be supported in a particular implementation, in which case the corresponding accessor method returns an implementation-specific value. When supported, the time stamp is returned as an `FileTime` object.

---

The following code snippet reads and prints the basic file attributes for a given file and uses the methods in the `BasicFileAttributes` class.

```
Path file = ...;
BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class);

System.out.println("creationTime: " + attr.creationTime());
System.out.println("lastAccessTime: " + attr.lastAccessTime());
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());

System.out.println("isDirectory: " + attr.isDirectory());
System.out.println("isOther: " + attr.isOther());
System.out.println("isRegularFile: " + attr.isRegularFile());
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
System.out.println("size: " + attr.size());
```

In addition to the accessor methods shown in this example, there is a `fileKey` method that returns either an object that uniquely identifies the file or `null` if no file key is available.

## Setting Time Stamps

The following code snippet sets the last modified time in milliseconds:

```
Path file = ...;
BasicFileAttributes attr =
    Files.readAttributes(file, BasicFileAttributes.class);
long currentTime = System.currentTimeMillis();
FileTime ft = FileTime.fromMillis(currentTime);
Files.setLastModifiedTime(file, ft);
}
```

## DOS File Attributes

DOS file attributes are also supported on file systems other than DOS, such as Samba. The following snippet uses the methods of the `DosFileAttributes` class.

```
Path file = ...;
try {
    DosFileAttributes attr =
        Files.readAttributes(file, DosFileAttributes.class);
    System.out.println("isReadOnly is " + attr.isReadOnly());
    System.out.println("isHidden is " + attr.isHidden());
    System.out.println("isArchive is " + attr.isArchive());
    System.out.println("isSystem is " + attr.isSystem());
} catch (UnsupportedOperationException x) {
    System.err.println("DOS file" +
        " attributes not supported:" + x);
}
```

However, you can set a DOS attribute using the `setAttribute(Path, String, Object, LinkOption...)` method, as follows:

```
Path file = ...;
Files.setAttribute(file, "dos:hidden", true);
```

## POSIX File Permissions

*POSIX* is an acronym for Portable Operating System Interface for UNIX and is a set of IEEE and ISO standards designed to ensure interoperability among different flavors of UNIX. If a program conforms to these POSIX standards, it should be easily ported to other POSIX-compliant operating systems.

Besides file owner and group owner, POSIX supports nine file permissions: read, write, and execute permissions for the file owner, members of the same group, and "everyone else."

The following code snippet reads the POSIX file attributes for a given file and prints them to standard output. The code uses the methods in the `PosixFileAttributes` class.

```
Path file = ...;
PosixFileAttributes attr =
    Files.readAttributes(file, PosixFileAttributes.class);
System.out.format("%s %s %s%n",
    attr.owner().getName(),
    attr.group().getName(),
    PosixFilePermissions.toString(attr.permissions()));
```

The `PosixFilePermissions` helper class provides several useful methods, as follows:

- The `toString` method, used in the previous code snippet, converts the file permissions to a string (for example, `rw-r--r--`).
- The `fromString` method accepts a string representing the file permissions and constructs a `Set` of file permissions.
- The `asFileAttribute` method accepts a `Set` of file permissions and constructs a file attribute that can be passed to the `Path.createFile` or `Path.createDirectory` method.

The following code snippet reads the attributes from one file and creates a new file, assigning the attributes from the original file to the new file:

```
Path sourceFile = ...;
Path newFile = ...;
PosixFileAttributes attrs =
    Files.readAttributes(sourceFile, PosixFileAttributes.class);
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(attrs.permissions());
Files.createFile(file, attr);
```

The `asFileAttribute` method wraps the permissions as a `FileAttribute`. The code then attempts to create a new file with those permissions. Note that the `umask` also applies, so the new file might be more secure than the permissions that were requested.

To set a file's permissions to values represented as a hard-coded string, you can use the following code:

```
Path file = ...;
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rw-------");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.setPosixFilePermissions(file, perms);
```

The `Chmod` example recursively changes the permissions of files in a manner similar to the `chmod` utility.

## Setting a File or Group Owner

To translate a name into an object you can store as a file owner or a group owner, you can use the `UserPrincipalLookupService` service. This service looks up a name or group name as a string and returns a `UserPrincipal` object representing that string. You can obtain the user principal look-up service for the default file system by using the `FileSystem.getUserPrincipalLookupService` method.

The following code snippet shows how to set the file owner by using the `setOwner` method:

```
Path file = ...;
UserPrincipal owner = file.GetFileSystem().getUserPrincipalLookupService()
        .lookupPrincipalByName("sally");
Files.setOwner(file, owner);
```

There is no special-purpose method in the `Files` class for setting a group owner. However, a safe way to do so directly is through the POSIX file attribute view, as follows:

```
Path file = ...;
GroupPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService()
        .lookupPrincipalByGroupName("green");
```

```
Files.getFileAttributeView(file, PosixFileAttributeView.class)
     .setGroup(group);
```

## User-Defined File Attributes

If the file attributes supported by your file system implementation aren't sufficient for your needs, you can use the `UserDefinedAttributeView` to create and track your own file attributes.

Some implementations map this concept to features like NTFS Alternative Data Streams and extended attributes on file systems such as ext3 and ZFS. Most implementations impose restrictions on the size of the value, for example, ext3 limits the size to 4 kilobytes.

A file's MIME type can be stored as a user-defined attribute by using this code snippet:

```
Path file = ...;
UserDefinedFileAttributeView view = Files
    .getFileAttributeView(file, UserDefinedFileAttributeView.class);
view.write("user.mimetype",
           Charset.defaultCharset().encode("text/html");
```

To read the MIME type attribute, you would use this code snippet:

```
Path file = ...;
UserDefinedFileAttributeView view = Files
.getFileAttributeView(file,UserDefinedFileAttributeView.class);
String name = "user.mimetype";
ByteBuffer buf = ByteBuffer.allocate(view.size(name));
view.read(name, buf);
buf.flip();
String value = Charset.defaultCharset().decode(buf).toString();
```

The `Xdd` example shows how to get, set, and delete a user-defined attribute.

---

**Note:** In Linux, you might have to enable extended attributes for user-defined attributes to work. If you receive an `UnsupportedOperationException` when trying to access the user-defined attribute view, you need to remount the file system. The following command remounts the root partition with extended attributes for the ext3 file system. If this command does not work for your flavor of Linux, consult the documentation.

```
$ sudo mount -o remount,user_xattr /
```

If you want to make the change permanent, add an entry to `/etc/fstab`.

---

## File Store Attributes

You can use the `FileStore` class to learn information about a file store, such as how much space is available. The `getFileStore(Path)` method fetches the file store for the specified file.

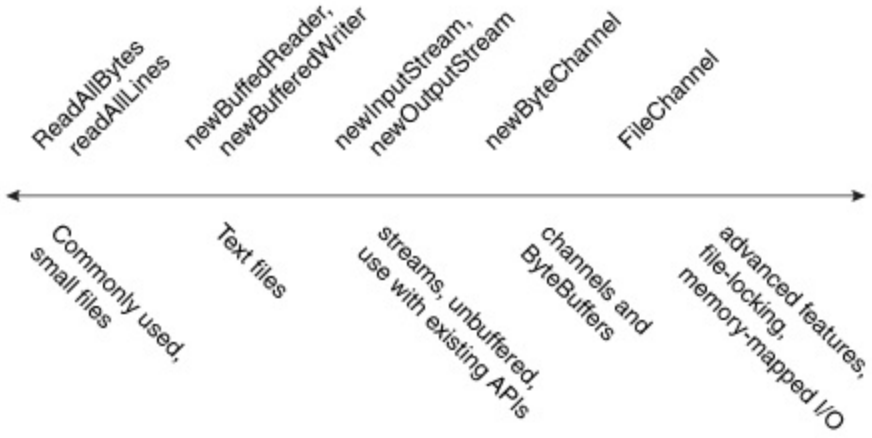
The following code snippet prints the space usage for the file store where a particular file resides:

```
Path file = ...;
FileStore store = Files.getFileStore(file);

long total = store.getTotalSpace() / 1024;
long used = (store.getTotalSpace() -
             store.getUnallocatedSpace()) / 1024;
long avail = store.getUsableSpace() / 1024;
```

The `DiskUsage` example uses this API to print disk space information for all the stores in the default file system. This example uses the `getFileStores` method in the `FileSystem` class to fetch all the file stores for the the file system.

# Reading, Writing, and Creating Files

This page discusses the details of reading, writing, creating, and opening files. There are a wide array of file I/O methods to choose from. To help make sense of the API, the following diagram arranges the file I/O methods by complexity.




File I/O Methods Arranged from Less Complex to More Complex

On the far left of the diagram are the utility methods `readAllBytes`, `readAllLines`, and the `write` methods, designed for simple, common cases. To the right of those are the methods used to iterate over a stream or lines of text, such as `newBufferedReader`, `newBufferedWriter`, then `newInputStream` and `newOutputStream`. These methods are interoperable with the `java.io` package. To the right of those are the methods for dealing with `ByteChannels`, `SeekableByteChannels`, and `ByteBuffers`, such as the `newByteChannel` method. Finally, on the far right are the methods that use `FileChannel` for advanced applications needing file locking or memory-mapped I/O.

---

**Note:** The methods for creating a new file enable you to specify an optional set of initial attributes for the file. For example, on a file system that supports the POSIX set of standards (such as UNIX), you can specify a file owner, group owner, or file permissions at the time the file is created. The Managing Metadata page explains file attributes, and how to access and set them.

---

This page has the following topics:



---

## The `OpenOptions` Parameter

Several of the methods in this section take an optional `OpenOptions` parameter. This parameter is optional and the API tells you what the default behavior is for the method when none is specified.

The following `StandardOpenOptions` enums are supported:

- `WRITE` Opens the file for write access.
- `APPEND` Appends the new data to the end of the file. This option is used with the `WRITE` or `CREATE` options.
- `TRUNCATE_EXISTING` Truncates the file to zero bytes. This option is used with the `WRITE` option.
- `CREATE_NEW` Creates a new file and throws an exception if the file already exists.
- `CREATE` Opens the file if it exists or creates a new file if it does not.
- `DELETE_ON_CLOSE` Deletes the file when the stream is closed. This option is useful for temporary files.
- `SPARSE` Hints that a newly created file will be sparse. This advanced option is honored on some file systems, such as NTFS, where large files with data "gaps" can be stored in a more efficient manner where those empty gaps do not consume disk space.
- `SYNC` Keeps the file (both content and metadata) synchronized with the underlying storage device.
- `DSYNC` Keeps the file content synchronized with the underlying storage device.

---

## Commonly Used Methods for Small Files

### Reading All Bytes or Lines from a File

If you have a small-ish file and you would like to read its entire contents in one pass, you can use the `readAllBytes(Path)` or `readAllLines(Path, Charset)` method. These methods take care of most of the work for you, such as opening and closing the stream, but are not intended for handling large files. The following code shows how to use the `readAllBytes` method:

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

### Writing All Bytes or Lines to a File

You can use one of the write methods to write bytes, or lines, to a file.

- `write(Path, byte[], OpenOption...)`
- `write(Path, Iterable< extends CharSequence>, Charset, OpenOption...)`

The following code snippet shows how to use a `write` method.

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);
```

---

## Buffered I/O Methods for Text Files

The `java.nio.file` package supports channel I/O, which moves data in buffers, bypassing some of the layers that can bottleneck stream I/O.

### Reading a File by Using Buffered Stream I/O

The `newBufferedReader(Path, Charset)` method opens a file for reading, returning a `BufferedReader` that can be used to read text from a file in an efficient manner.

The following code snippet shows how to use the `newBufferedReader` method to read from a file. The file is encoded in "US-ASCII."

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

### Writing a File by Using Buffered Stream I/O

You can use the `newBufferedWriter(Path, Charset, OpenOption...)` method to write to a file using a `BufferedWriter`.

The following code snippet shows how to create a file encoded in "US-ASCII" using this method:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

---

## Methods for Unbuffered Streams and Interoperable with `java.io` APIs

### Reading a File by Using Stream I/O

To open a file for reading, you can use the `newInputStream(Path, OpenOption...)` method. This method returns an unbuffered input stream for reading bytes from the file.

```
Path file = ...;
try (InputStream in = Files.newInputStream(file);
    BufferedReader reader =
      new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
```

```
} catch (IOException x) {
    System.err.println(x);
}
```

### Creating and Writing a File by Using Stream I/O

You can create a file, append to a file, or write to a file by using the `newOutputStream(Path, OpenOption...)` method. This method opens or creates a file for writing bytes and returns an unbuffered output stream.

The method takes an optional `OpenOption` parameter. If no open options are specified, and the file does not exist, a new file is created. If the file exists, it is truncated. This option is equivalent to invoking the method with the `CREATE` and `TRUNCATE_EXISTING` options.

The following code snippet opens a log file. If the file does not exist, it is created. If the file exists, it is opened for appending.

```
import static java.nio.file.StandardOpenOption.*;

// Convert the string to a
// byte array.
String s = ...;
byte data[] = s.getBytes();

try (OutputStream out = new BufferedOutputStream(
                Files.newOutputStream(CREATE, APPEND))) {
    ...
    out.write(data, 0, data.length);
} catch (IOException x) {
    System.err.println(x);
}
```

---

## Methods for Channels and `ByteBuffers`

### Reading and Writing Files by Using Channel I/O

While stream I/O reads a character at a time, channel I/O reads a buffer at a time. The `ByteChannel` interface provides basic `read` and `write` functionality. A `SeekableByteChannel` is a `ByteChannel` that has the capability to maintain a position in the channel and to change that position. A `SeekableByteChannel` also supports truncating the file associated with the channel and querying the file for its size.

The capability to move to different points in the file and then read from or write to that location makes random access of a file possible. See Random Access Files for more information.

There are two methods for reading and writing channel I/O.

- `newByteChannel(Path, OpenOption...)`
- `newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)`

---

**Note:** The `newByteChannel` methods return an instance of a `SeekableByteChannel`. With a default file system, you can cast this seekable byte channel to a `FileChannel` providing access to more advanced features such mapping a region of the file directly into memory for faster access, locking a region of the file so other processes cannot access it, or reading and writing bytes from an absolute position without affecting the channel's current position.

---

Both `newByteChannel` methods enable you to specify a list of `OpenOption` options. The same open options used by the `newOutputStream` methods are supported, in addition to one more option: `READ` is required because the `SeekableByteChannel` supports both reading and writing.

Specifying `READ` opens the channel for reading. Specifying `WRITE` or `APPEND` opens the channel for writing. If none of these options is specified, the channel is opened for reading.

The following code snippet reads a file and prints it to standard output:

```
// Defaults to READ
try (SeekableByteChannel sbc = Files.newByteChannel(file)) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    // Read the bytes with the proper encoding for this platform.  If
    // you skip this step, you might see something that looks like
    // Chinese characters when you expect Latin-style characters.
    String encoding = System.getProperty("file.encoding");
    while (sbc.read(buf) > 0) {
        buf.rewind();
        System.out.print(Charset.forName(encoding).decode(buf));
        buf.flip();
    }
} catch (IOException x) {
    System.out.println("caught exception: " + x);
```

The following code snippet, written for UNIX and other POSIX file systems, creates a log file with a specific set of file permissions. This code creates a log file or appends to the log file if it already exists. The log file is created with read/write permissions for owner and read only permissions for group.

```
import static java.nio.file.StandardCopyOption.*;

// Create the set of options for appending to the file.
Set<OpenOptions> options = new HashSet<OpenOption>();
options.add(APPEND);
options.add(CREATE);

// Create the custom permissions attribute.
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rw-r------");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);

// Convert the string to a ByteBuffer.
String s = ...;
byte data[] = s.getBytes();
ByteBuffer bb = ByteBuffer.wrap(data);

try (SeekableByteChannel sbc = Files.newByteChannel(file, options, attr)) {
    sbc.write(bb);
```

```
} catch (IOException x) {
    System.out.println("exception thrown: " + x);
}
```

---

## Methods for Creating Regular and Temporary Files

### Creating Files

You can create an empty file with an initial set of attributes by using the `createFile(Path, FileAttribute<?>)` method. For example, if, at the time of creation, you want a file to have a particular set of file permissions, use the `createFile` method to do so. If you do not specify any attributes, the file is created with default attributes. If the file already exists, `createFile` throws an exception.

In a single atomic operation, the `createFile` method checks for the existence of the file and creates that file with the specified attributes, which makes the process more secure against malicious code.

The following code snippet creates a file with default attributes:

```
Path file = ...;
try {
    // Create the empty file with default permissions, etc.
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
    System.err.format("file named %s" +
        " already exists%n", file);
} catch (IOException x) {
    // Some other sort of failure, such as permissions.
    System.err.format("createFile error: %s%n", x);
}
```

POSIX File Permissions has an example that uses `createFile(Path, FileAttribute<?>)` to create a file with pre-set permissions.

You can also create a new file by using the `newOutputStream` methods, as described in Creating and Writing a File using Stream I/O. If you open a new output stream and close it immediately, an empty file is created.

### Creating Temporary Files

You can create a temporary file using one of the following `createTempFile` methods:

- `createTempFile(Path, String, String, FileAttribute<?>)`
- `createTempFile(String, String, FileAttribute<?>)`

The first method allows the code to specify a directory for the temporary file and the second method creates a new file in the default temporary-file directory. Both methods allow you to specify a suffix for the filename and the first method allows you to also specify a prefix. The following code snippet gives an example of the second method:

```
try {
    Path tempFile = Files.createTempFile(null, ".myapp");
    System.out.format("The temporary file" +
        " has been created: %s%n", tempFile)
;
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

The result of running this file would be something like the following:

```
The temporary file has been created: /tmp/509668702974537184.myapp
```

The specific format of the temporary file name is platform specific.

# Random Access Files

*Random access files* permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.

This functionality is possible with the `SeekableByteChannel` interface. The `SeekableByteChannel` interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:

- `position` Returns the channel's current position
- `position(long)` Sets the channel's position
- `read(ByteBuffer)` Reads bytes into the buffer from the channel
- `write(ByteBuffer)` Writes bytes from the buffer to the channel
- `truncate(long)` Truncates the file (or other entity) connected to the channel

Reading and Writing Files With Channel I/O shows that the `Path.newByteChannel` methods return an instance of a `SeekableByteChannel`. On the default file system, you can use that channel as is, or you can cast it to a `FileChannel` giving you access to more advanced features, such as mapping a region of the file directly into memory for faster access, locking a region of the file, or reading and writing bytes from an absolute location without affecting the channel's current position.

The following code snippet opens a file for both reading and writing by using one of the `newByteChannel` methods. The `SeekableByteChannel` that is returned is cast to a `FileChannel`. Then, 12 bytes are read from the beginning of the file, and the string "I was here!" is written at that location. The current position in the file is moved to the end, and the 12 bytes from the beginning are appended. Finally, the string, "I was here!" is appended, and the channel on the file is closed.

```
String s = "I was here!\n";
byte data[] = s.getBytes();
ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
    // Read the first 12
    // bytes of the file.
    int nread;
    do {
        nread = fc.read(copy);
    } while (nread != -1 && copy.hasRemaining());

    // Write "I was here!" at the beginning of the file.
    fc.position(0);
    while (out.hasRemaining())
        fc.write(out);
    out.rewind();

    // Move to the end of the file.  Copy the first 12 bytes to
    // the end of the file.  Then write "I was here!" again.
    long length = fc.size();
    fc.position(length-1);
    copy.flip();
    while (copy.hasRemaining())
```

```
        fc.write(copy);
    while (out.hasRemaining())
        fc.write(out);
} catch (IOException x) {
    System.out.println("I/O Exception: " + x);
}
```

# Creating and Reading Directories

Some of the methods previously discussed, such as `delete`, work on files, links *and* directories. But how do you list all the directories at the top of a file system? How do you list the contents of a directory or create a directory?

This section covers the following functionality specific to directories:



## Listing a File System's Root Directories

You can list all the root directories for a file system by using the `FileSystem.getRootDirectories` method. This method returns an `Iterable`, which enables you to use the enhanced for statement to iterate over all the root directories.

The following code snippet prints the root directories for the default file system:

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name);
}
```

## Creating a Directory

You can create a new directory by using the `createDirectory(Path, FileAttribute<?>)` method. If you don't specify any `FileAttributes`, the new directory will have default attributes. For example:

```
Path dir = ...;
Files.createDirectory(path);
```

The following code snippet creates a new directory on a POSIX file system that has specific permissions:

```
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createDirectory(file, attr);
```

To create a directory several levels deep when one or more of the parent directories might not yet

exist, you can use the convenience method, `createDirectories(Path, FileAttribute<?>)`. As with the `createDirectory(Path, FileAttribute<?>)` method, you can specify an optional set of initial file attributes. The following code snippet uses default attributes:

```
Files.createDirectories(Paths.get("foo/bar/test"));
```

The directories are created, as needed, from the top down. In the `foo/bar/test` example, if the `foo` directory does not exist, it is created. Next, the `bar` directory is created, if needed, and, finally, the `test` directory is created.

It is possible for this method to fail after creating some, but not all, of the parent directories.

## Creating a Temporary Directory

You can create a temporary directory using one of `createTempDirectory` methods:

- `createTempDirectory(Path, String, FileAttribute<?>...)`
- `createTempDirectory(String, FileAttribute<?>...)`

The first method allows the code to specify a location for the temporary directory and the second method creates a new directory in the default temporary-fle directory.

## Listing a Directory's Contents

You can list all the contents of a directory by using the `newDirectoryStream(Path)` method. This method returns an object that implements the `DirectoryStream` interface. The class that implements the `DirectoryStream` interface also implements `Iterable`, so you can iterate through the directory stream, reading all of the objects. This approach scales well to very large directories.

---

**Remember:** The returned `DirectoryStream` is a *stream*. If you are not using a `try`-with-resources statement, don't forget to close the stream in the `finally` block. The `try`-with-resources statement takes care of this for you.

---

The following code snippet shows how to print the contents of a directory:

```
Path dir = ...;
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
    for (Path file: stream) {
        System.out.println(file.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    // IOException can never be thrown by the iteration.
    // In this snippet, it can only be thrown by newDirectoryStream.
    System.err.println(x);
}
```

The `Path` objects returned by the iterator are the names of the entries resolved against the directory.

So, if you are listing the contents of the `/tmp` directory, the entries are returned with the form `/tmp/a`, `/tmp/b`, and so on.

This method returns the entire contents of a directory: files, links, subdirectories, and hidden files. If you want to be more selective about the contents that are retrieved, you can use one of the other `newDirectoryStream` methods, as described later in this page.

Note that if there is an exception during directory iteration then `DirectoryIteratorException` is thrown with the `IOException` as the cause. Iterator methods cannot throw exception exceptions.

## Filtering a Directory Listing By Using Globbing

If you want to fetch only files and subdirectories where each name matches a particular pattern, you can do so by using the `newDirectoryStream(Path, String)` method, which provides a built-in glob filter. If you are not familiar with glob syntax, see What Is a Glob?

For example, the following code snippet lists files relating to Java: *.class*, *.java*, and *.jar* files.:

```
Path dir = ...;
try (DirectoryStream<Path> stream =
     Files.newDirectoryStream(dir, "*.{java,class,jar}")) {
    for (Path entry: stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException x) {
    // IOException can never be thrown by the iteration.
    // In this snippet, it can // only be thrown by newDirectoryStream.
    System.err.println(x);
}
```

## Writing Your Own Directory Filter

Perhaps you want to filter the contents of a directory based on some condition other than pattern matching. You can create your own filter by implementing the `DirectoryStream.Filter<T>` interface. This interface consists of one method, `accept`, which determines whether a file fulfills the search requirement.

For example, the following code snippet implements a filter that retrieves only directories:

```
DirectoryStream.Filter<Path> filter =
    newDirectoryStream.Filter<Path>() {
    public boolean accept(Path file) throws IOException {
        try {
            return (Files.isDirectory(path));
        } catch (IOException x) {
            // Failed to determine if it's a directory.
            System.err.println(x);
            return false;
        }
    }
};
```

Once the filter has been created, it can be invoked by using the `newDirectoryStream(Path, DirectoryStream.Filter<? super Path>)` method. The following code snippet uses the `isDirectory` filter to print only the directory's subdirectories to standard output:

```
Path dir = ...;
try (DirectoryStream<Path>
                      stream = Files.newDirectoryStream(dir, filter)) {
    for (Path entry: stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException x) {
    System.err.println(x);
}
```

This method is used to filter a single directory only. However, if you want to find all the subdirectories in a file tree, you would use the mechanism for Walking the File Tree.

# Links, Symbolic or Otherwise

As mentioned previously, the `java.nio.file` package, and the `Path` class in particular, is "link aware." Every `Path` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

The discussion so far has been about symbolic or *soft* links, but some file systems also support hard links. *Hard links* are more restrictive than symbolic links, as follows:

- The target of the link must exist.
- Hard links are generally not allowed on directories.
- Hard links are not allowed to cross partitions or volumes. Therefore, they cannot exist across file systems.
- A hard link looks, and behaves, like a regular file, so they can be hard to find.
- A hard link is, for all intents and purposes, the same entity as the original file. They have the same file permissions, time stamps, and so on. All attributes are identical.

Because of these restrictions, hard links are not used as often as symbolic links, but the `Path` methods work seamlessly with hard links.

Several methods deal specifically with links and are covered in the following sections:

- Creating a Symbolic Link
- Creating a Hard Link
- Detecting a Symbolic Link
- Finding the Target of a Link

## Creating a Symbolic Link

If your file system supports it, you can create a symbolic link by using the `createSymbolicLink(Path, Path, FileAttribute<?>)` method. The second `Path` argument represents the target file or directory and might or might not exist. The following code snippet creates a symbolic link with default permissions:

```
Path newLink = ...;
Path target = ...;
try {
    Files.createSymbolicLink(newLink, target);
} catch (IOException x) {
    System.err.println(x);
} catch (UnsupportedOperationException x) {
    // Some file systems do not support symbolic links.
    System.err.println(x);
}
```

The `FileAttributes` vararg enables you to specify initial file attributes that are set atomically when the link is created. However, this argument is intended for future use and is not currently implemented.

## Creating a Hard Link

You can create a hard (or *regular*) link to an existing file by using the `createLink(Path, Path)` method. The second `Path` argument locates the existing file, and it must exist or a `NoSuchFileException` is thrown. The following code snippet shows how to create a link:

```
Path newLink = ...;
Path existingFile = ...;
try {
    Files.createLink(newLink, existingFile);
} catch (IOException x) {
    System.err.println(x);
} catch (UnsupportedOperationException x) {
    // Some file systems do not
    // support adding an existing
    // file to a directory.
    System.err.println(x);
}
```

## Detecting a Symbolic Link

To determine whether a `Path` instance is a symbolic link, you can use the `isSymbolicLink(Path)` method. The following code snippet shows how:

```
Path file = ...;
boolean isSymbolicLink =
    Files.isSymbolicLink(file);
```

For more information, see Managing Metadata.

## Finding the Target of a Link

You can obtain the target of a symbolic link by using the `readSymbolicLink(Path)` method, as follows:

```
Path link = ...;
try {
    System.out.format("Target of link" +
        " '%s' is '%s'%n", link,
        Files.readSymbolicLink(link));
} catch (IOException x) {
    System.err.println(x);
}
```

If the `Path` is not a symbolic link, this method throws a `NotLinkException`.

# Walking the File Tree

Do you need to create an application that will recursively visit all the files in a file tree? Perhaps you need to delete every `.class` file in a tree, or find every file that hasn't been accessed in the last year. You can do so with the `FileVisitor` interface.

This section covers the following:



## The FileVisitor Interface

To walk a file tree, you first need to implement a `FileVisitor`. A `FileVisitor` specifies the required behavior at key points in the traversal process: when a file is visited, before a directory is accessed, after a directory is accessed, or when a failure occurs. The interface has four methods that correspond to these situations:

- `preVisitDirectory` Invoked before a directory's entries are visited.
- `postVisitDirectory` Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
- `visitFile` Invoked on the file being visited. The file's `BasicFileAttributes` is passed to the method, or you can use the file attributes package to read a specific set of attributes. For example, you can choose to read the file's `DosFileAttributeView` to determine if the file has the "hidden" bit set.
- `visitFileFailed` Invoked when the file cannot be accessed. The specific exception is passed to the method. You can choose whether to throw the exception, print it to the console or a log file, and so on.

If you don't need to implement all four of the `FileVisitor` methods, instead of implementing the `FileVisitor` interface, you can extend the `SimpleFileVisitor` class. This class, which implements the `FileVisitor` interface, visits all files in a tree and throws an `IOError` when an error is encountered. You can extend this class and override only the methods that you require.

Here is an example that extends `SimpleFileVisitor` to print all entries in a file tree. It prints the entry whether the entry is a regular file, a symbolic link, a directory, or some other "unspecified" type of file. It also prints the size, in bytes, of each file. Any exception that is encountered is printed to the console.

The `FileVisitor` methods are shown in bold:

```
import static java.nio.file.FileVisitResult.*;
```

```
public static class PrintFiles
    extends SimpleFileVisitor<Path> {

    // Print information about
    // each type of file.
    @Override
    public FileVisitResult visitFile(Path file,
                                     BasicFileAttributes attr) {
        if (attr.isSymbolicLink()) {
            System.out.format("Symbolic link: %s ", file);
        } else if (attr.isRegularFile()) {
            System.out.format("Regular file: %s ", file);
        } else {
            System.out.format("Other: %s ", file);
        }
        System.out.println("(" + attr.size() + "bytes)");
        return CONTINUE;
    }

    // Print each directory visited.
    @Override
    public FileVisitResult postVisitDirectory(Path dir,
                                          IOException exc) {
        System.out.format("Directory: %s%n", dir);
        return CONTINUE;
    }

    // If there is some error accessing
    // the file, let the user know.
    // If you don't override this method
    // and an error occurs, an IOException
    // is thrown.
    @Override
    public FileVisitResult visitFileFailed(Path file,
                                       IOException exc) {
        System.err.println(exc);
        return CONTINUE;
    }
}
```

## Kickstarting the Process

Once you have implemented your `FileVisitor`, how do you initiate the file walk? There are two `walkFileTree` methods in the `Files` class.

- `walkFileTree(Path, FileVisitor)`
- `walkFileTree(Path, Set<FileVisitOption>, int, FileVisitor)`

The first method requires only a starting point and an instance of your `FileVisitor`. You can invoke the `PrintFiles` file visitor as follows:

```
Path startingDir = ...;
PrintFiles pf = new PrintFiles();
Files.walkFileTree(startingDir, pf);
```

The second `walkFileTree` method enables you to additionally specify a limit on the number of levels visited and a set of `FileVisitOption` enums. If you want to ensure that this method walks the entire file tree, you can specify `Integer.MAX_VALUE` for the maximum depth argument.

You can specify the `FileVisitOption` enum, `FOLLOW_LINKS`, which indicates that symbolic links should be followed.

This code snippet shows how the four-argument method can be invoked:

```
import static java.nio.file.FileVisitResult.*;

Path startingDir = ...;

EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW_LINKS);

Finder finder = new Finder(pattern);
Files.walkFileTree(startingDir, opts, Integer.MAX_VALUE, finder);
```

## Considerations When Creating a FileVisitor

A file tree is walked depth first, but you cannot make any assumptions about the iteration order that subdirectories are visited.

If your program will be changing the file system, you need to carefully consider how you implement your `FileVisitor`.

For example, if you are writing a recursive delete, you first delete the files in a directory before deleting the directory itself. In this case, you delete the directory in `postVisitDirectory`.

If you are writing a recursive copy, you create the new directory in `preVisitDirectory` before attempting to copy the files to it (in `visitFiles`). If you want to preserve the attributes of the source directory (similar to the UNIX `cp -p` command), you need to do that *after* the files have been copied, in `postVisitDirectory`. The `Copy` example shows how to do this.

If you are writing a file search, you perform the comparison in the `visitFile` method. This method finds all the files that match your criteria, but it does not find the directories. If you want to find both files and directories, you must also perform the comparison in either the `preVisitDirectory` or `postVisitDirectory` method. The `Find` example shows how to do this.

You need to decide whether you want symbolic links to be followed. If you are deleting files, for example, following symbolic links might not be advisable. If you are copying a file tree, you might want to allow it. By default, `walkFileTree` does not follow symbolic links.

The `visitFile` method is invoked for files. If you have specified the `FOLLOW_LINKS` option and your file tree has a circular link to a parent directory, the looping directory is reported in the `visitFileFailed` method with the `FileSystemLoopException`. The following code snippet shows how to catch a circular link and is from the `Copy` example:

```
@Override
public FileVisitResult
    visitFileFailed(Path file,
        IOException exc) {
    if (exc instanceof FileSystemLoopException) {
        System.err.println("cycle detected: " + file);
```

```
    } else {
        System.err.format("Unable to copy:" + " %s: %s%n", file, exc);
    }
    return CONTINUE;
}
```

This case can occur only when the program is following symbolic links.

## Controlling the Flow

Perhaps you want to walk the file tree looking for a particular directory and, when found, you want the process to terminate. Perhaps you want to skip specific directories.

The `FileVisitor` methods return a `FileVisitResult` value. You can abort the file walking process or control whether a directory is visited by the values you return in the `FileVisitor` methods:

- `CONTINUE` Indicates that the file walking should continue. If the `preVisitDirectory` method returns `CONTINUE`, the directory is visited.
- `TERMINATE` Immediately aborts the file walking. No further file walking methods are invoked after this value is returned.
- `SKIP_SUBTREE` When `preVisitDirectory` returns this value, the specified directory and its subdirectories are skipped. This branch is "pruned out" of the tree.
- `SKIP_SIBLINGS` When `preVisitDirectory` returns this value, the specified directory is not visited, `postVisitDirectory` is not invoked, and no further unvisited siblings are visited. If returned from the `postVisitDirectory` method, no further siblings are visited. Essentially, nothing further happens in the specified directory.

In this code snippet, any directory named `SCCS` is skipped:

```
import static java.nio.file.FileVisitResult.*;

public FileVisitResult
     preVisitDirectory(Path dir,
         BasicFileAttributes attrs) {
    (if (dir.getFileName().toString().equals("SCCS")) {
         return SKIP_SUBTREE;
    }
    return CONTINUE;
}
```

In this code snippet, as soon as a particular file is located, the file name is printed to standard output, and the file walking terminates:

```
import static java.nio.file.FileVisitResult.*;

// The file we are looking for.
Path lookingFor = ...;

public FileVisitResult
    visitFile(Path file,
        BasicFileAttributes attr) {
    if (file.getFileName().equals(lookingFor)) {
```

```
        System.out.println("Located file: " + file);
        return TERMINATE;
    }
    return CONTINUE;
}
```

## Examples

The following examples demonstrate the file walking mechanism:

- `Find` Recurses a file tree looking for files and directories that match a particular glob pattern. This example is discussed in Finding Files.
- `Chmod` Recursively changes permissions on a file tree (for POSIX systems only).
- `Copy` Recursively copies a file tree.
- `WatchDir` Demonstrates the mechanism that watches a directory for files that have been created, deleted or modified. Calling this program with the `-r` option watches an entire tree for changes. For more information about the file notification service, see Watching a Directory for Changes.

# Finding Files

If you have ever used a shell script, you have most likely used pattern matching to locate files. In fact, you have probably used it extensively. If you haven't used it, pattern matching uses special characters to create a pattern and then file names can be compared against that pattern. For example, in most shell scripts, the asterisk, `*`, matches any number of characters. For example, the following command lists all the files in the current directory that end in `.html`:

```
% ls *.html
```

The `java.nio.file` package provides programmatic support for this useful feature. Each file system implementation provides a `PathMatcher`. You can retrieve a file system's `PathMatcher` by using the `getPathMatcher(String)` method in the `FileSystem` class. The following code snippet fetches the path matcher for the default file system:

```
String pattern = ...;
PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher("glob:" + pattern);
```

The string argument passed to `getPathMatcher` specifies the syntax flavor and the pattern to be matched. This example specifies *glob* syntax. If you are unfamiliar with glob syntax, see What is a Glob.

Glob syntax is easy to use and flexible but, if you prefer, you can also use regular expressions, or *regex*, syntax. For further information about regex, see the Regular Expressions lesson. Some file system implementations might support other syntaxes.

If you want to use some other form of string-based pattern matching, you can create your own `PathMatcher` class. The examples in this page use glob syntax.

Once you have created your `PathMatcher` instance, you are ready to match files against it. The `PathMatcher` interface has a single method, `matches`, that takes a `Path` argument and returns a boolean: It either matches the pattern, or it does not. The following code snippet looks for files that end in `.java` or `.class` and prints those files to standard output:

```
PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher("glob:*.{java,class}");

Path filename = ...;
if (matcher.matches(filename)) {
    System.out.println(filename);
}
```

## Recursive Pattern Matching

Searching for files that match a particular pattern goes hand-in-hand with walking a file tree. How many times do you know a file is *somewhere* on the file system, but where? Or perhaps you need to

find all files in a file tree that have a particular file extension.

The `Find` example does precisely that. `Find` is similar to the UNIX `find` utility, but has pared down functionally. You can extend this example to include other functionality. For example, the `find` utility supports the `-prune` flag to exclude an entire subtree from the search. You could implement that functionality by returning `SKIP_SUBTREE` in the `preVisitDirectory` method. To implement the `-L` option, which follows symbolic links, you could use the four-argument `walkFileTree` method and pass in the `FOLLOW_LINKS` enum (but make sure that you test for circular links in the `visitFile` method).

To run the Find application, use the following format:

```
% java Find <path> -name "<glob_pattern>"
```

The pattern is placed inside quotation marks so any wildcards are not interpreted by the shell. For example:

```
% java Find . -name "*.html"
```

Here is the source code for the `Find` example:

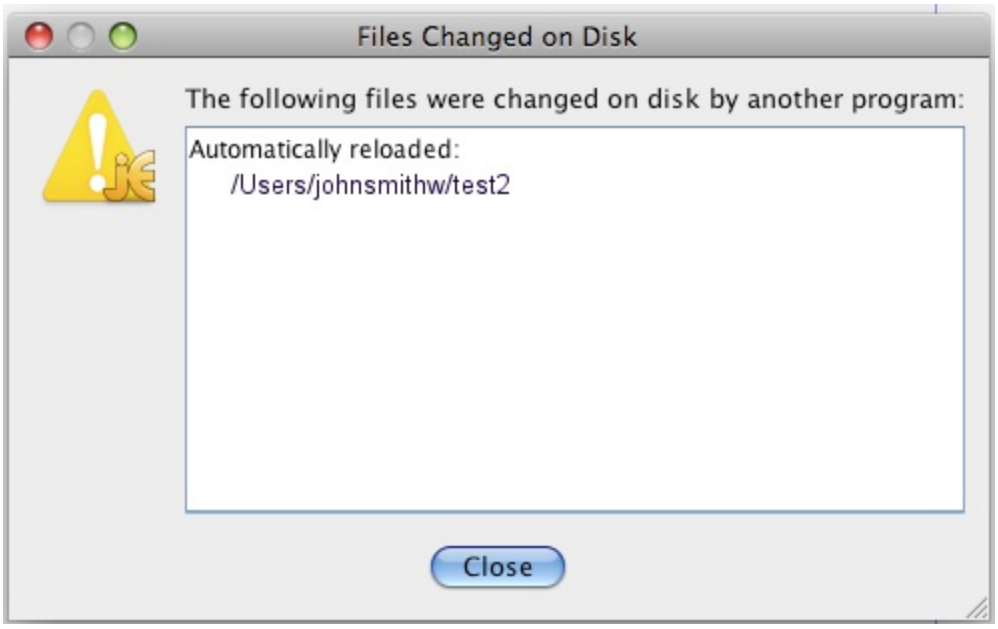
```
/**
 * Sample code that finds files that match the specified glob pattern.
 * For more information on what constitutes a glob pattern, see
 * http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob
 *
 * The file or directories that match the pattern are printed to
 * standard out.  The number of matches is also printed.
 *
 * When executing this application, you must put the glob pattern
 * in quotes, so the shell will not expand any wild cards:
 *              java Find . -name "*.java"
 */

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import static java.nio.file.FileVisitResult.*;
import static java.nio.file.FileVisitOption.*;
import java.util.*;


public class Find {

    public static class Finder
        extends SimpleFileVisitor<Path> {

        private final PathMatcher matcher;
        private int numMatches = 0;

        Finder(String pattern) {
            matcher = FileSystems.getDefault()
                    .getPathMatcher("glob:" + pattern);
        }

        // Compares the glob pattern against
```

```
        // the file or directory name.
        void find(Path file) {
            Path name = file.getFileName();
            if (name != null && matcher.matches(name)) {
                numMatches++;
                System.out.println(file);
            }
        }

        // Prints the total number of
        // matches to standard out.
        void done() {
            System.out.println("Matched: "
                + numMatches);
        }

        // Invoke the pattern matching
        // method on each file.
        @Override
        public FileVisitResult visitFile(Path file,
                BasicFileAttributes attrs) {
            find(file);
            return CONTINUE;
        }

        // Invoke the pattern matching
        // method on each directory.
        @Override
        public FileVisitResult preVisitDirectory(Path dir,
                BasicFileAttributes attrs) {
            find(dir);
            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFileFailed(Path file,
                IOException exc) {
            System.err.println(exc);
            return CONTINUE;
        }
    }

    static void usage() {
        System.err.println("java Find <path>" +
            " -name \"<glob_pattern>\"");
        System.exit(-1);
    }

    public static void main(String[] args)
        throws IOException {

        if (args.length < 3 || !args[1].equals("-name"))
            usage();

        Path startingDir = Paths.get(args[0]);
        String pattern = args[2];

        Finder finder = new Finder(pattern);
        Files.walkFileTree(startingDir, finder);
        finder.done();
    }
}
```

Recursively walking a file tree is covered in Walking the File Tree.

# Watching a Directory for Changes

Have you ever found yourself editing a file, using an IDE or another editor, and a dialog box appears to inform you that one of the open files has changed on the file system and needs to be reloaded? Or perhaps, like the NetBeans IDE, the application just quietly updates the file without notifying you. The following sample dialog box shows how this notification looks with the free editor, jEdit:




jEdit Dialog Box Showing That a Modified File Is Detected

To implement this functionality, called *file change notification*, a program must be able to detect what is happening to the relevant directory on the file system. One way to do so is to poll the file system looking for changes, but this approach is inefficient. It does not scale to applications that have hundreds of open files or directories to monitor.

The `java.nio.file` package provides a file change notification API, called the Watch Service API. This API enables you to register a directory (or directories) with the watch service. When registering, you tell the service which types of events you are interested in: file creation, file deletion, or file modification. When the service detects an event of interest, it is forwarded to the registered process. The registered process has a thread (or a pool of threads) dedicated to watching for any events it has registered for. When an event comes in, it is handled as needed.

This section covers the following:



## Watch Service Overview

The `WatchService` API is fairly low level, allowing you to customize it. You can use it as is, or you can choose to create a high-level API on top of this mechanism so that it is suited to your particular needs.

Here are the basic steps required to implement a watch service:

- Create a `WatchService` "watcher" for the file system.
- For each directory that you want monitored, register it with the watcher. When registering a directory, you specify the type of events for which you want notification. You receive a `WatchKey` instance for each directory that you register.
- Implement an infinite loop to wait for incoming events. When an event occurs, the key is signaled and placed into the watcher's queue.
- Retrieve the key from the watcher's queue. You can obtain the file name from the key.
- Retrieve each pending event for the key (there might be multiple events) and process as needed.
- Reset the key, and resume waiting for events.
- Close the service: The watch service exits when either the thread exits or when it is closed (by invoking its `closed` method).

`WatchKeys` are thread-safe and can be used with the `java.nio.concurrent` package. You can dedicate a thread pool to this effort.

## Try It Out

Because this API is more advanced, try it out before proceeding. Save the `WatchDir` example to your computer, and compile it. Create a `test` directory that will be passed to the example. `WatchDir` uses a single thread to process all events, so it blocks keyboard input while waiting for events. Either run the program in a separate window, or in the background, as follows:

```
java WatchDir test &
```

Play with creating, deleting, and editing files in the `test` directory. When any of these events occurs, a message is printed to the console. When you have finished, delete the `test` directory and `WatchDir` exits. Or, if you prefer, you can manually kill the process.

You can also watch an entire file tree by specifying the `-r` option. When you specify `-r`, `WatchDir` walks the file tree, registering each directory with the watch service.

## Creating a Watch Service and Registering for Events

The first step is to create a new `WatchService` by using the `newWatchService` method in the `FileSystem` class, as follows:

```
WatchService watcher = FileSystems.getDefault().newWatchService();
```

Next, register one or more objects with the watch service. Any object that implements the `Watchable`

interface can be registered. The `Path` class implements the `Watchable` interface, so each directory to be monitored is registered as a `Path` object.

As with any `Watchable`, the `Path` class implements two `register` methods. This page uses the two-argument version, `register(WatchService, WatchEvent.Kind<?>...)`. (The three-argument version takes a `WatchEvent.Modifier`, which is not currently implemented.)

When registering an object with the watch service, you specify the types of events that you want to monitor. The supported `StandardWatchEventKinds` event types follow:

- `ENTRY_CREATE` A directory entry is created.
- `ENTRY_DELETE` A directory entry is deleted.
- `ENTRY_MODIFY` A directory entry is modified.
- `OVERFLOW` Indicates that events might have been lost or discarded. You do not have to register for the `OVERFLOW` event to receive it.

The following code snippet shows how to register a `Path` instance for all three event types:

```
import static java.nio.file.StandardWatchEventKinds.*;

Path dir = ...;
try {
    WatchKey key = dir.register(watcher,
                           ENTRY_CREATE,
                           ENTRY_DELETE,
                           ENTRY_MODIFY);
} catch (IOException x) {
    System.err.println(x);
}
```

## Processing Events

The order of events in an event processing loop follow:

**1.** Get a watch key. Three methods are provided:

- `poll` Returns a queued key, if available. Returns immediately with a `null` value, if unavailable.
- `poll(long, TimeUnit)` Returns a queued key, if one is available. If a queued key is not immediately available, the program waits until the specified time. The `TimeUnit` argument determines whether the specified time is nanoseconds, milliseconds, or some other unit of time.
- `take` Returns a queued key. If no queued key is available, this method waits.

**2.** Process the pending events for the key. You fetch the `List` of `WatchEvents` from the `pollEvents` method.
**3.** Retrieve the type of event by using the `kind` method. No matter what events the key has registered for, it is possible to receive an `OVERFLOW` event. You can choose to handle the overflow or ignore it, but you should test for it.
**4.** Retrieve the file name associated with the event. The file name is stored as the context of the event, so the `context` method is used to retrieve it.

**5.** After the events for the key have been processed, you need to put the key back into a `ready` state by invoking `reset`. If this method returns `false`, the key is no longer valid and the loop can exit. This step is very **important**. If you fail to invoke `reset`, this key will not receive any further events.

A watch key has a state. At any given time, its state might be one of the following:

- `Ready` indicates that the key is ready to accept events. When first created, a key is in the ready state.
- `Signaled` indicates that one or more events are queued. Once the key has been signaled, it is no longer in the ready state until the `reset` method is invoked.
- `Invalid` indicates that the key is no longer active. This state happens when one of the following events occurs:
  - The process explicitly cancels the key by using the `cancel` method.
  - The directory becomes inaccessible.
  - The watch service is closed.

Here is an example of an event processing loop. It is taken from the `Email` example, which watches a directory, waiting for new files to appear. When a new file becomes available, it is examined to determine if it is a `text/plain` file by using the `probeContentType(Path)` method. The intention is that `text/plain` files will be emailed to an alias, but that implementation detail is left to the reader.

The methods specific to the watch service API are shown in bold:

```
for (;;) {

    // wait for key to be signaled
    WatchKey key;
    try {
        key = watcher.take();
    } catch (InterruptedException x) {
        return;
    }

    for (WatchEvent<?> event: key.pollEvents()) {
        WatchEvent.Kind<?> kind = event.kind();

        // This key is registered only
        // for ENTRY_CREATE events,
        // but an OVERFLOW event can
        // occur regardless if events
        // are lost or discarded.
        if (kind == OVERFLOW) {
            continue;
        }

        // The filename is the
        // context of the event.
        WatchEvent<Path> ev = (WatchEvent<Path>)event;
        Path filename = ev.context();

        // Verify that the new
        //  file is a text file.
        try {
            // Resolve the filename against the directory.
            // If the filename is "test" and the directory is "foo",
            // the resolved name is "test/foo".
```

```
            Path child = dir.resolve(filename);
            if (!Files.probeContentType(child).equals("text/plain")) {
                System.err.format("New file '%s'" +
                    " is not a plain text file.%n", filename);
                continue;
            }
        } catch (IOException x) {
            System.err.println(x);
            continue;
        }

        // Email the file to the
        //  specified email alias.
        System.out.format("Emailing file %s%n", filename);
        //Details left to reader....
    }

    // Reset the key -- this step is critical if you want to
    // receive further watch events.  If the key is no longer valid,
    // the directory is inaccessible so exit the loop.
    boolean valid = key.reset();
    if (!valid) {
        break;
    }
}
```

## Retrieving the File Name

The file name is retrieved from the event context. The `Email` example retrieves the file name with this code:

```
WatchEvent<Path> ev = (WatchEvent<Path>)event;
Path filename = ev.context();
```

When you compile the `Email` example, it generates the following error:

```
Note: Email.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This error is a result of the line of code that casts the `WatchEvent<T>` to a `WatchEvent<Path>`. The `WatchDir` example avoids this error by creating a utility `cast` method that suppresses the unchecked warning, as follows:

```
@SuppressWarnings("unchecked")
static <T> WatchEvent<T> cast(WatchEvent<?> event) {
    return (WatchEvent<Path>)event;
}
```

If you are unfamiliar with the `@SuppressWarnings` syntax, see Annotations.

## When to Use and Not Use This API

The Watch Service API is designed for applications that need to be notified about file change events. It is well suited for any application, like an editor or IDE, that potentially has many open files and

needs to ensure that the files are synchronized with the file system. It is also well suited for an application server that watches a directory, perhaps waiting for `.jsp` or `.jar` files to drop, in order to deploy them.

This API is *not* designed for indexing a hard drive. Most file system implementations have native support for file change notification. The Watch Service API takes advantage of this support where available. However, when a file system does not support this mechanism, the Watch Service will poll the file system, waiting for events.

# Other Useful Methods

A few useful methods did not fit elsewhere in this lesson and are covered here. This section covers the following:



## Determining MIME Type

To determine the MIME type of a file, you might find the `probeContentType(Path)` method useful. For example:

```
try {
    String type = Files.probeContentType(filename);
    if (type == null) {
        System.err.format("'%s' has an" + " unknown filetype.%n", filename);
    } else if (!type.equals("text/plain") {
        System.err.format("'%s' is not" + " a plain text file.%n", filename);
        continue;
    }
} catch (IOException x) {
    System.err.println(x);
}
```

Note that `probeContentType` returns null if the content type cannot be determined.

The implementation of this method is highly platform specific and is not infallible. The content type is determind by the platform's default file type detector. For example, if the detector determines a file's content type to be `application/x-java` based on the `.class` extension, it might be fooled.

You can provide a custom `FileTypeDetector` if the default is not sufficient for your needs.

The `Email` example uses the `probeContentType` method.

## Default File System

To retrieve the default file system, use the `getDefault` method. Typically, this `FileSystems` method (note the plural) is chained to one of the `FileSystem` methods (note the singular), as follows:

```
PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher("glob:*.*");
```

## Path String Separator

The path separator for POSIX file systems is the forward slash, `/`, and for Microsoft Windows is the backslash, `\`. Other file systems might use other delimiters. To retrieve the `Path` separator for the

default file system, you can use one of the following approaches:

```
String separator = File.separator;
String separator = FileSystems.getDefault().getSeparator();
```

The `getSeparator` method is also used to retrieve the path separator for any available file system.

## File System's File Stores

A file system has one or more file stores to hold its files and directories. The *file store* represents the underlying storage device. In UNIX operating systems, each mounted file system is represented by a file store. In Microsoft Windows, each volume is represented by a file store: `C:`, `D:`, and so on.

To retrieve a list of all the file stores for the file system, you can use the `getFileStores` method. This method returns an `Iterable`, which allows you to use the enhanced for statement to iterate over all the root directories.

```
for (FileStore store: FileSystems.getDefault().getFileStores()) {
   ...
}
```

If you want to retrive the file store where a particular file is located, use the `getFileStore` method in the `Files` class, as follows:

```
Path file = ...;
FileStore store= Files.getFileStore(file);
```

The `DiskUsage` example uses the `getFileStores` method.

# Legacy File I/O Code

## Interoperability With Legacy Code

Prior to the Java SE 7 release, the `java.io.File` class was the mechanism used for file I/O, but it had several drawbacks.

- Many methods didn't throw exceptions when they failed, so it was impossible to obtain a useful error message. For example, if a file deletion failed, the program would receive a "delete fail" but wouldn't know if it was because the file didn't exist, the user didn't have permissions, or there was some other problem.
- The `rename` method didn't work consistently across platforms.
- There was no real support for symbolic links.
- More support for metadata was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient.
- Many of the `File` methods didn't scale. Requesting a large directory listing over a server could result in a hang. Large directories could also cause memory resource problems, resulting in a denial of service.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.

Perhaps you have legacy code that uses `java.io.File` and would like to take advantage of the `java.nio.file.Path` functionality with minimal impact to your code.

The `java.io.File` class provides the `toPath` method, which converts an old style `File` instance to a `java.nio.file.Path` instance, as follows:

```
Path input = file.toPath();
```

You can then take advantage of the rich feature set available to the `Path` class.

For example, assume you had some code that deleted a file:

```
file.delete();
```

You could modify this code to use the `Files.delete` method, as follows:

```
Path fp = file.toPath();
Files.delete(fp);
```

Conversely, the `Path.toFile` method constructs a `java.io.File` object for a `Path` object.

## Mapping java.io.File Functionality to java.nio.file

Because the Java implementation of file I/O has been completely re-architected in the Java SE 7 release, you cannot swap one method for another method. If you want to use the rich functionality offered by the `java.nio.file` package, your easiest solution is to use the `File.toPath` method as suggested in the previous section. However, if you do not want to use that approach or it is not sufficient for your needs, you must rewrite your file I/O code.

There is no one-to-one correspondence between the two APIs, but the following table gives you a general idea of what functionality in the `java.io.File` API maps to in the `java.nio.file` API and tells you where you can obtain more information.

| java.io.File Functionality | java.nio.file Functionality | Tutorial Coverage |
|---|---|---|
| `java.io.File` | `java.nio.file.Path` | The Path Class |
| `java.io.RandomAccessFile` | The `SeekableByteChannel` functionality. | Random Access Files |
| `File.canRead`, `canWrite`, `canExecute` | `Files.isReadable`, `Files.isWritable`, and `Files.isExecutable`. On UNIX file systems, the Managing Metadata (File and File Store Attributes) package is used to check the nine file permissions. | Checking a File or Directory Managing Metadata |
| `File.isDirectory()`, `File.isFile()`, and `File.length()` | `Files.isDirectory(Path, LinkOption...)`, `Files.isRegularFile(Path, LinkOption...)`, and `Files.size(Path)` | Managing Metadata |
| `File.lastModified()` and `File.setLastModified(long)` | `Files.getLastModifiedTime(Path, LinkOption...)` and `Files.setLastMOdifiedTime(Path, FileTime)` | Managing Metadata |
| The `File` methods that set various attributes: `setExecutable`, `setReadable`, `setReadOnly`, `setWritable` | These methods are replaced by the `Files` method `setAttribute(Path, String, Object, LinkOption...)`. | Managing Metadata |
| `new File(parent, "newfile")` | `parent.resolve("newfile")` | Path Operations |
| `File.renameTo` | `Files.move` | Moving a File or Directory |
| `File.delete` | `Files.delete` | Deleting a File or Directory |
| `File.createNewFile` | `Files.createFile` | Creating Files |
| `File.deleteOnExit` | Replaced by the `DELETE_ON_CLOSE` option specified in the `createFile` method. | Creating Files |
| | | Creating Files |

| | | |
|---|---|---|
| `File.createTempFile` | `Files.createTempFile(Path, String, FileAttributes<?>)`, `Files.createTempFile(Path, String, String, FileAttributes<?>)` | Creating and Writing a File by Using Stream I/O<br>Reading and Writing Files by Using Channel I/O |
| `File.exists` | `Files.exists` and `Files.notExists` | Verifying the Existence of a File or Directory |
| `File.compareTo` and `equals` | `Path.compareTo` and `equals` | Comparing Two Paths |
| `File.getAbsolutePath` and `getAbsoluteFile` | `Path.toAbsolutePath` | Converting a Path |
| `File.getCanonicalPath` and `getCanonicalFile` | `Path.toRealPath` or `normalize` | Converting a Path (`toRealPath`)<br>Removing Redundancies From a Path (`normalize`) |
| `File.toURI` | `Path.toURI` | Converting a Path |
| `File.isHidden` | `Files.isHidden` | Retrieving Information About the Path |
| `File.list` and `listFiles` | `Path.newDirectoryStream` | Listing a Directory's Contents |
| `File.mkdir` and `mkdirs` | `Path.createDirectory` | Creating a Directory |
| `File.listRoots` | `FileSystem.getRootDirectories` | Listing a File System's Root Directories |
| `File.getTotalSpace`, `File.getFreeSpace`, `File.getUsableSpace` | `FileStore.getTotalSpace`, `FileStore.getUnallocatedSpace`, `FileStore.getUsableSpace`, `FileStore.getTotalSpace` | File Store Attributes |

## Summary

The `java.io` package contains many classes that your programs can use to read and write data. Most of the classes implement sequential access streams. The sequential access streams can be divided into two groups: those that read and write bytes and those that read and write Unicode characters. Each sequential access stream has a speciality, such as reading from or writing to a file, filtering data as its read or written, or serializing an object.

The `java.nio.file` package provides extensive support for file and file system I/O. This is a very comprehensive API, but the key entry points are as follows:

- The `Path` class has methods for manipulating a path.
- The `Files` class has methods for file operations, such as moving, copy, deleting, and also methods for retrieving and setting file attributes.
- The `FileSystem` class has a variety of methods for obtaining information about the file system.

More information on NIO.2 can be found on the OpenJDK: NIO project website on java.net. This site includes resources for features provided by NIO.2 that are beyond the scope of this tutorial, such as multicasting, asynchronous I/O, and creating your own file system implementation.

# Questions and Exercises: Basic I/O

## Questions

1. What class and method would you use to read a few pieces of data that are at known positions near the end of a large file?

2. When invoking `format`, what is the best way to indicate a new line?

3. How would you determine the MIME type of a file?

4. What method(s) would you use to determine whether a file is a symbolic link?

## Exercises

1. Write an example that counts the number of times a particular character, such as `e`, appears in a file. The character can be specified at the command line. You can use `xanadu.txt` as the input file.

2. The file `datafile` begins with a single `long` that tells you the offset of a single `int` piece of data within the same file. Write a program that gets the `int` piece of data. What is the `int` data?

Check your answers.

## Questions

Question 1. What class and method would you use to read a few pieces of data that are at known positions near the end of a large file?

Answer 1. `Files.newByteChannel` returns an instance of `SeekableByteChannel` which allows you to read from (or write to) any position in the file.

Question 2. When invoking `format`, what is the best way to indicate a new line?

Answer 2. Use the `%n` conversion — the `\n` escape is not platform independent!

Question 3. How would you determine the MIME type of a file?

Answer 3. The `Files.probeContentType` method uses the platform's underlying file type detector to evaluate and return the MIME type.

Question 4. What method(s) would you use to determine whether a file is a symbolic link?

Answer 4. You would use the `Files.isSymbolicLink` method.

## Exercises

Exercise 1. Write an example that counts the number of times a particular character, such as `e`, appears in a file. The character can be specified at the command line. You can use `xanadu.txt` as the input file.

Answer 1. See `CountLetter.java` for the solution.

Exercise 2. The file `datafile` begins with a single `long` that tells you the offset of a single `int` piece of data within the same file. Write a program that gets the `int` piece of data. What is the `int` data?

Answer 2. `123`. See `FindInt.java` for the solution.

# Lesson: Concurrency

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as *concurrent* software.

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the `java.util.concurrent` packages.

---

**Note:** See online version of topics in this ebook to download complete source code.

---

# Processes and Threads

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

## Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a `ProcessBuilder` object. Multiprocess applications are beyond the scope of this lesson.

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

# Thread Objects

Each thread is associated with an instance of the class `Thread`. There are two basic strategies for using `Thread` objects to create a concurrent application.

- To directly control thread creation and management, simply instantiate `Thread` each time the application needs to initiate an asynchronous task.
- To abstract thread management from the rest of your application, pass the application's tasks to an *executor*.

This section documents the use of `Thread` objects. Executors are discussed with other high-level concurrency objects.

# Defining and Starting a Thread

An application that creates an instance of `Thread` must provide the code that will run in that thread. There are two ways to do this:

- *Provide a `Runnable` object.* The `Runnable` interface defines a single method, `run`, meant to contain the code executed in the thread. The `Runnable` object is passed to the `Thread` constructor, as in the `HelloRunnable` example:

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

- *Subclass `Thread`.* The `Thread` class itself implements `Runnable`, though its `run` method does nothing. An application can subclass `Thread`, providing its own implementation of `run`, as in the `HelloThread` example:

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Notice that both examples invoke `Thread.start` in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a `Runnable` object, is more general, because the `Runnable` object can subclass a class other than `Thread`. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of `Thread`. This lesson focuses on the first approach, which separates the `Runnable` task from the `Thread` object that executes the task. Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

The `Thread` class defines a number of methods useful for thread management. These include `static` methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and `Thread` object. We'll examine some of these methods in the following sections.

# Pausing Execution with Sleep

`Thread.sleep` causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The `sleep` method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the `SimpleThreads` example in a later section.

Two overloaded versions of `sleep` are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we'll see in a later section. In any case, you cannot assume that invoking `sleep` will suspend the thread for precisely the time period specified.

The `SleepMessages` example uses `sleep` to print messages at four-second intervals:

```
public class SleepMessages {
    public static void main(String args[])
        throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };

        for (int i = 0;
             i < importantInfo.length;
             i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

Notice that `main` declares that it `throws InterruptedException`. This is an exception that `sleep` throws when another thread interrupts the current thread while `sleep` is active. Since this application has not defined another thread to cause the interrupt, it doesn't bother to catch `InterruptedException`.

# Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking `interrupt` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

## Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw `InterruptedException`, it simply returns from the `run` method after it catches that exception. For example, suppose the central message loop in the `SleepMessages` example were in the `run` method of a thread's `Runnable` object. Then it might be modified as follows to support interrupts:

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Many methods that throw `InterruptedException`, such as `sleep`, are designed to cancel their current operation and return immediately when an interrupt is received.

What if a thread goes a long time without invoking a method that throws `InterruptedException`? Then it must periodically invoke `Thread.interrupted`, which returns `true` if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an `InterruptedException`:

```
if (Thread.interrupted()) {
```

```
    throw new InterruptedException();
}
```

This allows interrupt handling code to be centralized in a `catch` clause.

## The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. Invoking `Thread.interrupt` sets this flag. When a thread checks for an interrupt by invoking the static method `Thread.interrupted`, interrupt status is cleared. The non-static `isInterrupted` method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an `InterruptedException` clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking `interrupt`.

# Joins

The `join` method allows one thread to wait for the completion of another. If `t` is a `Thread` object whose thread is currently executing,

```
t.join();
```

causes the current thread to pause execution until `t`'s thread terminates. Overloads of `join` allow the programmer to specify a waiting period. However, as with `sleep`, `join` is dependent on the OS for timing, so you should not assume that `join` will wait exactly as long as you specify.

Like `sleep`, `join` responds to an interrupt by exiting with an `InterruptedException`.

# The SimpleThreads Example

The following example brings together some of the concepts of this section. `SimpleThreads` consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the `Runnable` object, `MessageLoop`, and waits for it to finish. If the `MessageLoop` thread takes too long to finish, the main thread interrupts it.

The `MessageLoop` thread prints out a series of messages. If interrupted before it has printed all its messages, the `MessageLoop` thread prints a message and exits.

```
public class SimpleThreads {

    // Display a message, preceded by
    // the name of the current thread
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
                          threadName,
                          message);
    }

    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0;
                     i < importantInfo.length;
                     i++) {
                    // Pause for 4 seconds
                    Thread.sleep(4000);
                    // Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }

    public static void main(String args[])
        throws InterruptedException {

        // Delay, in milliseconds before
        // we interrupt MessageLoop
        // thread (default one hour).
        long patience = 1000 * 60 * 60;

        // If command line argument
        // present, gives patience
        // in seconds.
        if (args.length > 0) {
            try {
                patience = Long.parseLong(args[0]) * 1000;
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
```

```
                System.exit(1);
            }
        }

        threadMessage("Starting MessageLoop thread");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Waiting for MessageLoop thread to finish");
        // loop until MessageLoop
        // thread exits
        while (t.isAlive()) {
            threadMessage("Still waiting...");
            // Wait maximum of 1 second
            // for MessageLoop thread
            // to finish.
            t.join(1000);
            if (((System.currentTimeMillis() - startTime) > patience)
                  && t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();
                // Shouldn't be long now
                // -- wait indefinitely
                t.join();
            }
        }
        threadMessage("Finally!");
    }
}
```

# Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

However, synchronization can introduce *thread contention*, which occurs when two or more threads try to access the same resource simultaneously *and* cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention. See the section Liveness for more information.

This section covers the following topics:

- Thread Interference describes how errors are introduced when multiple threads access shared data.
- Memory Consistency Errors describes errors that result from inconsistent views of shared memory.
- Synchronized Methods describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
- Implicit Locks and Synchronization describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
- Atomic Access talks about the general idea of operations that can't be interfered with by other threads.

# Thread Interference

Consider a simple class called `Counter`

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

`Counter` is designed so that each invocation of `increment` will add 1 to `c`, and each invocation of `decrement` will subtract 1 from `c`. However, if a `Counter` object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

It might not seem possible for operations on instances of `Counter` to interleave, since both operations on `c` are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expression `c++` can be decomposed into three steps:

**1.** Retrieve the current value of `c`.
**2.** Increment the retrieved value by 1.
**3.** Store the incremented value back in `c`.

The expression `c--` can be decomposed the same way, except that the second step decrements instead of increments.

Suppose Thread A invokes `increment` at about the same time Thread B invokes `decrement`. If the initial value of `c` is `0`, their interleaved actions might follow this sequence:

**1.** Thread A: Retrieve c.
**2.** Thread B: Retrieve c.
**3.** Thread A: Increment retrieved value; result is 1.
**4.** Thread B: Decrement retrieved value; result is -1.
**5.** Thread A: Store result in c; c is now 1.

**6.** Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

# Memory Consistency Errors

*Memory consistency errors* occur when different threads have inconsistent views of what should be the same data. The causes of memory consistency errors are complex and beyond the scope of this tutorial. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple `int` field is defined and initialized:

```
int counter = 0;
```

The `counter` field is shared between two threads, A and B. Suppose thread A increments `counter`:

```
counter++;
```

Then, shortly afterwards, thread B prints out `counter`:

```
System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to `counter` will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

There are several actions that create happens-before relationships. One of them is synchronization, as we will see in the following sections.

We've already seen two actions that create happens-before relationships.

- When a statement invokes `Thread.start`, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.
- When a thread terminates and causes a `Thread.join` in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

For a list of actions that create happens-before relationships, refer to the Summary page of the `java.util.concurrent` package..

# Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

---

**Warning:** When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

---

Synchronized methods enable a simple strategy for preventing thread interference and memory

consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through `synchronized` methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

# Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquistion of the same lock.

## Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

## Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

## Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

# Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as `c++`, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`).
- Reads and writes are atomic for *all* variables declared `volatile` (*including* `long` and `double` variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using `volatile` variables reduces the risk of memory consistency errors, because any write to a `volatile` variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a `volatile` variable are always visible to other threads. What's more, it also means that when a thread reads a `volatile` variable, it sees not just the latest change to the `volatile`, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Some of the classes in the `java.util.concurrent` package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on High Level Concurrency Objects.

# Liveness

A concurrent application's ability to execute in a timely manner is known as its *liveness*. This section describes the most common kind of liveness problem, deadlock, and goes on to briefly describe two other liveness problems, starvation and livelock.

# Deadlock

*Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. This example application, `Deadlock`, models this possibility:

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

When `Deadlock` runs, it's extremely likely that both threads will block when they attempt to invoke `bowBack`. Neither block will ever end, because each thread is waiting for the other to exit `bow`.

# Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

## Starvation

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

## Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

# Guarded Blocks

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose, for example `guardedJoy` is a method that must not proceed until a shared variable `joy` has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

A more efficient guard invokes `Object.wait` to suspend the current thread. The invocation of `wait` does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

---

**Note:** Always invoke `wait` inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.

---

Like many methods that suspend execution, `wait` can throw `InterruptedException`. In this example, we can just ignore that exception — we only care about the value of `joy`.

Why is this version of `guardedJoy` synchronized? Suppose `d` is the object we're using to invoke `wait`. When a thread invokes `d.wait`, it must own the intrinsic lock for `d` — otherwise an error is thrown. Invoking `wait` inside a synchronized method is a simple way to acquire the intrinsic lock.

When `wait` is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke `Object.notifyAll`, informing all threads waiting on that lock that something important has happened:

```
public synchronized notifyJoy() {
    joy = true;
```

```
    notifyAll();
}
```

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of `wait`.

---

**Note:** There is a second notification method, `notify`, which wakes up a single thread. Because `notify` doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.

---

Let's use guarded blocks to create a *Producer-Consumer* application. This kind of application shares data between two threads: the *producer*, that creates the data, and the *consumer*, that does something with it. The two threads communicate using a shared object. Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data.

In this example, the data is a series of text messages, which are shared through an object of type `Drop`:

```
public class Drop {
    // Message sent from producer
    // to consumer.
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
```

```
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

The producer thread, defined in `Producer`, sends a series of familiar messages. The string "DONE" indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

```
import java.util.Random;

public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0;
             i < importantInfo.length;
             i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

The consumer thread, defined in `Consumer`, simply retrieves the messages and prints them out, until it retrieves the "DONE" string. This thread also pauses for random intervals.

```
import java.util.Random;

public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             ! message.equals("DONE");
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
```

```
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

Finally, here is the main thread, defined in `ProducerConsumerExample`, that launches the producer and consumer threads.

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

---

**Note:** The `Drop` class was written in order to demonstrate guarded blocks. To avoid re-inventing the wheel, examine the existing data structures in the Java Collections Framework before trying to code your own data-sharing objects. For more information, refer to the Questions and Exercises section.

---

# Immutable Objects

An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.

The following subsections take a class whose instances are mutable and derives a class with immutable instances from it. In so doing, they give general rules for this kind of conversion and demonstrate some of the advantages of immutable objects.

# A Synchronized Class Example

The class, `SynchronizedRGB`, defines objects that represent colors. Each object represents the color as three integers that stand for primary color values and a string that gives the name of the color.

```
public class SynchronizedRGB {

    // Values must be between 0 and 255.
    private int red;
    private int green;
    private int blue;
    private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public SynchronizedRGB(int red,
                           int green,
                           int blue,
                           String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }

    public void set(int red,
                    int green,
                    int blue,
                    String name) {
        check(red, green, blue);
        synchronized (this) {
            this.red = red;
            this.green = green;
            this.blue = blue;
            this.name = name;
        }
    }

    public synchronized int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public synchronized String getName() {
        return name;
    }

    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

`SynchronizedRGB` must be used carefully to avoid being seen in an inconsistent state. Suppose, for example, a thread executes the following code:

```
SynchronizedRGB color =
    new SynchronizedRGB(0, 0, 0, "Pitch Black");
...
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

If another thread invokes `color.set` after Statement 1 but before Statement 2, the value of `myColorInt` won't match the value of `myColorName`. To avoid this outcome, the two statements must be bound together:

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable version of `SynchronizedRGB`.

# A Strategy for Defining Immutable Objects

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

**1.** Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
**2.** Make all fields `final` and `private`.
**3.** Don't allow subclasses to override methods. The simplest way to do this is to declare the class as `final`. A more sophisticated approach is to make the constructor `private` and construct instances in factory methods.
**4.** If the instance fields include references to mutable objects, don't allow those objects to be changed:

- Don't provide methods that modify the mutable objects.
- Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Applying this strategy to `SynchronizedRGB` results in the following steps:

**1.** There are two setter methods in this class. The first one, `set`, arbitrarily transforms the object, and has no place in an immutable version of the class. The second one, `invert`, can be adapted by having it create a new object instead of modifying the existing one.
**2.** All fields are already `private`; they are further qualified as `final`.
**3.** The class itself is declared `final`.
**4.** Only one field refers to an object, and that object is itself immutable. Therefore, no safeguards against changing the state of "contained" mutable objects are necessary.

After these changes, we have `ImmutableRGB`:

```
final public class ImmutableRGB {

    // Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
```

```
    }

    public ImmutableRGB(int red,
                        int green,
                        int blue,
                        String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }


    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                        255 - green,
                        255 - blue,
                        "Inverse of " + name);
    }
}
```

# High Level Concurrency Objects

So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new `java.util.concurrent` packages. There are also new concurrent data structures in the Java Collections Framework.

- Lock objects support locking idioms that simplify many concurrent applications.
- Executors define a high-level API for launching and managing threads. Executor implementations provided by `java.util.concurrent` provide thread pool management suitable for large-scale applications.
- Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
- `ThreadLocalRandom` (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

# Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the `java.util.concurrent.locks` package. We won't examine this package in detail, but instead will focus on its most basic interface, `Lock`.

`Lock` objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a `Lock` object at a time. `Lock` objects also support a `wait/notify` mechanism, through their associated `Condition` objects.

The biggest advantage of `Lock` objects over implicit locks is their ability to back out of an attempt to acquire a lock. The `tryLock` method backs out if the lock is not available immediately or before a timeout expires (if specified). The `lockInterruptibly` method backs out if another thread sends an interrupt before the lock is acquired.

Let's use `Lock` objects to solve the deadlock problem we saw in Liveness. Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our `Friend` objects must acquire locks for *both* participants before proceeding with the bow. Here is the source code for the improved model, `Safelock`. To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) {
            this.name = name;
        }

        public String getName() {
            return this.name;
        }

        public boolean impendingBow(Friend bower) {
            Boolean myLock = false;
            Boolean yourLock = false;
            try {
                myLock = lock.tryLock();
                yourLock = bower.lock.tryLock();
            } finally {
                if (! (myLock && yourLock)) {
                    if (myLock) {
                        lock.unlock();
                    }
                    if (yourLock) {
                        bower.lock.unlock();
                    }
                }
            }
```

```
            return myLock && yourLock;
        }

        public void bow(Friend bower) {
            if (impendingBow(bower)) {
                try {
                    System.out.format("%s: %s has"
                        + " bowed to me!%n",
                        this.name, bower.getName());
                    bower.bowBack(this);
                } finally {
                    lock.unlock();
                    bower.lock.unlock();
                }
            } else {
                System.out.format("%s: %s started"
                    + " to bow to me, but saw that"
                    + " I was already bowing to"
                    + " him.%n",
                    this.name, bower.getName());
            }
        }

        public void bowBack(Friend bower) {
            System.out.format("%s: %s has" +
                " bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    static class BowLoop implements Runnable {
        private Friend bower;
        private Friend bowee;

        public BowLoop(Friend bower, Friend bowee) {
            this.bower = bower;
            this.bowee = bowee;
        }

        public void run() {
            Random random = new Random();
            for (;;) {
                try {
                    Thread.sleep(random.nextInt(10));
                } catch (InterruptedException e) {}
                bowee.bow(bower);
            }
        }
    }


    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new BowLoop(alphonse, gaston)).start();
        new Thread(new BowLoop(gaston, alphonse)).start();
    }
}
```

# Executors

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its `Runnable` object, and the thread itself, as defined by a `Thread` object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors*. The following subsections describe executors in detail.

- Executor Interfaces define the three executor object types.
- Thread Pools are the most common kind of executor implementation.
- Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.

# Executor Interfaces

The `java.util.concurrent` package defines three executor interfaces:

- `Executor`, a simple interface that supports launching new tasks.
- `ExecutorService`, a subinterface of `Executor`, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
- `ScheduledExecutorService`, a subinterface of `ExecutorService`, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

## The `Executor` Interface

The `Executor` interface provides a single method, `execute`, designed to be a drop-in replacement for a common thread-creation idiom. If `r` is a `Runnable` object, and `e` is an `Executor` object you can replace

```
(new Thread(r)).start();
```

with

```
e.execute(r);
```

However, the definition of `execute` is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the `Executor` implementation, `execute` may do the same thing, but is more likely to use an existing worker thread to run `r`, or to place `r` in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in `java.util.concurrent` are designed to make full use of the more advanced `ExecutorService` and `ScheduledExecutorService` interfaces, although they also work with the base `Executor` interface.

## The `ExecutorService` Interface

The `ExecutorService` interface supplements `execute` with a similar, but more versatile `submit` method. Like `execute`, `submit` accepts `Runnable` objects, but also accepts `Callable` objects, which allow the task to return a value. The `submit` method returns a `Future` object, which is used to retrieve the `Callable` return value and to manage the status of both `Callable` and `Runnable` tasks.

`ExecutorService` also provides methods for submitting large collections of `Callable` objects. Finally, `ExecutorService` provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

## The `ScheduledExecutorService` Interface

The `ScheduledExecutorService` interface supplements the methods of its parent `ExecutorService` with `schedule`, which executes a `Runnable` or `Callable` task after a specified delay. In addition, the interface defines `scheduleAtFixedRate` and `scheduleWithFixedDelay`, which executes specified tasks repeatedly, at defined intervals.

# Thread Pools

Most of the executor implementations in `java.util.concurrent` use *thread pools*, which consist of *worker threads*. This kind of thread exists separately from the `Runnable` and `Callable` tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the `newFixedThreadPool` factory method in `java.util.concurrent.Executors` This class also provides the following factory methods:

- The `newCachedThreadPool` method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
- The `newSingleThreadExecutor` method creates an executor that executes a single task at a time.
- Several factory methods are `ScheduledExecutorService` versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of `java.util.concurrent.ThreadPoolExecutor` or `java.util.concurrent.ScheduledThreadPoolExecutor` will give you additional options.

# Fork/Join

The fork/join framework is an implementation of the `ExecutorService` interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any `ExecutorService` implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the `ForkJoinPool` class, an extension of the `AbstractExecutorService` class. `ForkJoinPool` implements the core work-stealing algorithm and can execute `ForkJoinTask` processes.

## Basic Use

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

Wrap this code in a `ForkJoinTask` subclass, typically using one of its more specialized types, either `RecursiveTask` (which can return a result) or `RecursiveAction`.

After your `ForkJoinTask` subclass is ready, create the object that represents all the work to be done and pass it to the `invoke()` method of a `ForkJoinPool` instance.

## Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

```
public class ForkBlur extends RecursiveAction {
```

```
    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;

    // Processing window size; should be odd.
    private int mBlurWidth = 15;

    public ForkBlur(int[] src, int start, int length, int[] dst) {
        mSource = src;
        mStart = start;
        mLength = length;
        mDestination = dst;
    }

    protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index++) {
            // Calculate average.
            float rt = 0, gt = 0, bt = 0;
            for (int mi = -sidePixels; mi <= sidePixels; mi++) {
                int mindex = Math.min(Math.max(mi + index, 0),
                                      mSource.length - 1);
                int pixel = mSource[mindex];
                rt += (float)((pixel & 0x00ff0000) >> 16)
                      / mBlurWidth;
                gt += (float)((pixel & 0x0000ff00) >>  8)
                      / mBlurWidth;
                bt += (float)((pixel & 0x000000ff) >>  0)
                      / mBlurWidth;
            }

            // Reassemble destination pixel.
            int dpixel = (0xff000000     ) |
                   (((int)rt) << 16) |
                   (((int)gt) <<  8) |
                   (((int)bt) <<  0);
            mDestination[index] = dpixel;
        }
    }

  ...
```

Now you implement the abstract `compute()` method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

```
protected static int sThreshold = 100000;

protected void compute() {
    if (mLength < sThreshold) {
        computeDirectly();
        return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
              new ForkBlur(mSource, mStart + split, mLength - split,
                           mDestination));
}
```

If the previous methods are in a subclass of the `RecursiveAction` class, then setting up the task to

run in a `ForkJoinPool` is straightforward, and involves the following steps:

**1.** Create a task that represents all of the work to be done.

```
// source image pixels are in src
// destination image pixels are in dst
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
```

**2.** Create the `ForkJoinPool` that will run the task.

```
ForkJoinPool pool = new ForkJoinPool();
```

**3.** Run the task.

```
pool.invoke(fb);
```

For the full source code, including some extra code that creates the destination image file, see the `ForkBlur` example.

## Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the `ForkBlur.java` example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the `java.util.Arrays` class for its `parallelSort()` methods. These methods are similar to `sort()`, but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the `java.util.streams` package, which is part of Project Lambda scheduled for the Java SE 8 release. For more information, see the Lambda Expressions section.

# Concurrent Collections

The `java.util.concurrent` package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- `ConcurrentMap` is a subinterface of `java.util.Map` that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`.
- `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches. The standard general-purpose implementation of `ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

# Atomic Variables

The java.util.concurrent.atomic package defines classes that support atomic operations on single variables. All classes have `get` and `set` methods that work like reads and writes on `volatile` variables. That is, a `set` has a happens-before relationship with any subsequent `get` on the same variable. The atomic `compareAndSet` method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the `Counter` class we originally used to demonstrate thread interference:

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

One way to make `Counter` safe from thread interference is to make its methods synchronized, as in `SynchronizedCounter`:

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

}
```

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the `int` field with an `AtomicInteger` allows us to prevent thread interference without resorting to synchronization, as in `AtomicCounter`:

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

# Concurrent Random Numbers

In JDK 7, `java.util.concurrent` includes a convenience class, `ThreadLocalRandom`, for applications that expect to use random numbers from multiple threads or `ForkJoinTask`s.

For concurrent access, using `ThreadLocalRandom` instead of `Math.random()` results in less contention and, ultimately, better performance.

All you need to do is call `ThreadLocalRandom.current()`, then call one of its methods to retrieve a random number. Here is one example:

```
int r = ThreadLocalRandom.current() .nextInt(4, 77);
```

# For Further Reading

- *Concurrent Programming in Java: Design Principles and Pattern (2nd Edition)* by Doug Lea. A comprehensive work by a leading expert, who's also the architect of the Java platform's concurrency framework.
- *Java Concurrency in Practice* by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. A practical guide designed to be accessible to the novice.
- *Effective Java Programming Language Guide (2nd Edition)* by Joshua Bloch. Though this is a general programming guide, its chapter on threads contains essential "best practices" for concurrent programming.
- *Concurrency: State Models & Java Programs (2nd Edition)*, by Jeff Magee and Jeff Kramer. An introduction to concurrent programming through a combination of modeling and practical examples.
- *Java Concurrent Animated:* Animations that show usage of concurrency features.

# Questions and Exercises: Concurrency

## Questions

**1.** Can you pass a `Thread` object to `Executor.execute`? Would such an invocation make sense?

## Exercises

**1.** Compile and run `BadThreads.java`:

```
public class BadThreads {

    static String message;

    private static class CorrectorThread
        extends Thread {

        public void run() {
            try {
                sleep(1000);
            } catch (InterruptedException e) {}
            // Key statement 1:
            message = "Mares do eat oats.";
        }
    }

    public static void main(String args[])
        throws InterruptedException {

        (new CorrectorThread()).start();
        message = "Mares do not eat oats.";
        Thread.sleep(2000);
        // Key statement 2:
        System.out.println(message);
    }
}
```

The application should print out "Mares do eat oats." Is it guaranteed to always do this? If not, why not? Would it help to change the parameters of the two invocations of `Sleep`? How would you guarantee that all changes to `message` will be visible in the main thread?

**2.** Modify the producer-consumer example in Guarded Blocks to use a standard library class instead of the `Drop` class.

Check your answers.

## Questions

**1. Question:** Can you pass a `Thread` object to `Executor.execute`? Would such an invocation make sense? Why or why not? **Answer:** `Thread` implements the `Runnable` interface, so you can pass an instance of `Thread` to `Executor.execute`. However it doesn't make sense to use `Thread` objects this way. If the object is directly instantiated from `Thread`, its `run` method doesn't do anything. You can define a subclass of `Thread` with a useful `run` method — but such a class would implement features that the executor would not use.

## Exercises

**1. Exercise:** Compile and run `BadThreads.java`:

```
public class BadThreads {

    static String message;

    private static class CorrectorThread
        extends Thread {

        public void run() {
            try {
                sleep(1000);
            } catch (InterruptedException e) {}
            // Key statement 1:
            message = "Mares do eat oats.";
        }
    }

    public static void main(String args[])
        throws InterruptedException {

        (new CorrectorThread()).start();
        message = "Mares do not eat oats.";
        Thread.sleep(2000);
        // Key statement 2:
        System.out.println(message);
    }
}
```

The application should print out "Mares do eat oats." Is it guaranteed to always do this? If not, why not? Would it help to change the parameters of the two invocations of `Sleep`? How would you guarantee that all changes to `message` will be visible to the main thread?
**Solution:** The program will almost always print out "Mares do eat oats." However, this result is not guaranteed, because there is no happens-before relationship between "Key statement 1" and "Key statment 2". This is true even if "Key statement 1" actually executes before "Key statement 2" — remember, a happens-before relationship is about visibility, not sequence.
There are two ways you can guarantee that all changes to `message` will be visible to the main thread:

- In the main thread, retain a reference to the `CorrectorThread` instance. Then invoke `join` on that instance before referring to `message`
- Encapsulate `message` in an object with synchronized methods. Never reference `message` except through those methods.

Both of these techniques establish the necessary happens-before relationship, making changes to

`message` visible.
A third technique is to simply declare `message` as `volatile`. This guarantees that any write to `message` (as in "Key statement 1") will have a happens-before relationship with any subsequent reads of `message` (as in "Key statement 2"). But it does not guarantee that "Key statement 1" will *literally* happen before "Key statement 2". They will *probably* happen in sequence, but because of scheduling uncertainities and the unknown granularity of `sleep`, this is not guaranteed.
Changing the arguments of the two `sleep` invocations does not help either, since this does nothing to guarantee a happens-before relationship.
**2. Exercise:** Modify the producer-consumer example in Guarded Blocks to use a standard library class instead of the `Drop` class. **Solution:** The `java.util.concurrent.BlockingQueue` interface defines a `get` method that blocks if the queue is empty, and a `put` methods that blocks if the queue is full. These are effectively the same operations defined by `Drop` — except that `Drop` is not a queue! However, there's another way of looking at Drop: it's a queue with a capacity of zero. Since there's no room in the queue for *any* elements, every `get` blocks until the corresponding `take` and every `take` blocks until the corresponding `get`. There is an implementation of `BlockingQueue` with precisely this behavior: `java.util.concurrent.SynchronousQueue`.
`BlockingQueue` is almost a drop-in replacement for `Drop`. The main problem in `Producer` is that with `BlockingQueue`, the `put` and `get` methods throw `InterruptedException`. This means that the existing `try` must be moved up a level:

```
import java.util.Random;
import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {
    private BlockingQueue<String> drop;

    public Producer(BlockingQueue<String> drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        try {
            for (int i = 0;
                 i < importantInfo.length;
                 i++) {
                drop.put(importantInfo[i]);
                Thread.sleep(random.nextInt(5000));
            }
            drop.put("DONE");
        } catch (InterruptedException e) {}
    }
}
```

Similar changes are required for `Consumer`:

```
import java.util.Random;
import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
```

```
    private BlockingQueue<String> drop;

    public Consumer(BlockingQueue<String> drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        try {
            for (String message = drop.take();
                 ! message.equals("DONE");
                 message = drop.take()) {
                System.out.format("MESSAGE RECEIVED: %s%n",
                                  message);
                Thread.sleep(random.nextInt(5000));
            }
        } catch (InterruptedException e) {}
    }
}
```

For `ProducerConsumerExample`, we simply change the declaration for the `drop` object:

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;

public class ProducerConsumerExample {
    public static void main(String[] args) {
        BlockingQueue<String> drop =
            new SynchronousQueue<String> ();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

# Lesson: The Platform Environment

An application runs in a *platform environment*, defined by the underlying operating system, the Java virtual machine, the class libraries, and various configuration data supplied when the application is launched. This lesson describes some of the APIs an application uses to examine and configure its platform environment. The lesson consists of three sections:

- Configuration Utilities describes APIs used to access configuration data supplied when the application is deployed, or by the application's user.
- System Utilities describes miscellaneous APIs defined in the `System` and `Runtime` classes.
- PATH and CLASSPATH describes environment variables used to configure JDK development tools and other applications.

---

**Note:** See online version of topics in this ebook to download complete source code.

---

# Configuration Utilities

This section describes some of the configuration utilities that help an application access its startup context.

# Properties

*Properties* are configuration values managed as *key/value pairs*. In each pair, the key and value are both `String` values. The key identifies, and is used to retrieve, the value, much as a variable name is used to retrieve the variable's value. For example, an application capable of downloading files might use a property named "download.lastDirectory" to keep track of the directory used for the last download.

To manage properties, create instances of `java.util.Properties`. This class provides methods for the following:

- loading key/value pairs into a `Properties` object from a stream,
- retrieving a value from its key,
- listing the keys and their values,
- enumerating over the keys, and
- saving the properties to a stream.

For an introduction to streams, refer to the section I/O Streams in the Basic I/O lesson.

`Properties` extends `java.util.Hashtable`. Some of the methods inherited from `Hashtable` support the following actions:

- testing to see if a particular key or value is in the `Properties` object,
- getting the current number of key/value pairs,
- removing a key and its value,
- adding a key/value pair to the `Properties` list,
- enumerating over the values or the keys,
- retrieving a value by its key, and
- finding out if the `Properties` object is empty.

---

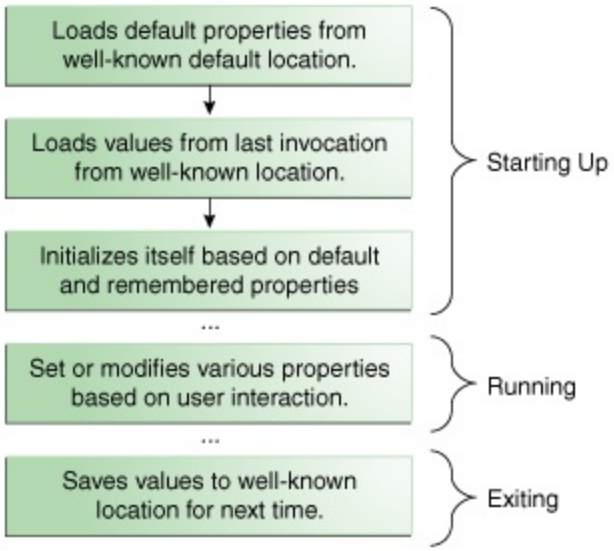
**Security Considerations:** Access to properties is subject to approval by the current security manager. The example code segments in this section are assumed to be in standalone applications, which, by default, have no security manager. The same code in an applet may not work depending on the browser in which it is running. See What Applets Can and Cannot Do in the Java Applets lesson for information about security restrictions on applets.

---

The `System` class maintains a `Properties` object that defines the configuration of the current working environment. For more about these properties, see System Properties. The remainder of this section explains how to use properties to manage application configuration.

## Properties in the Application Life Cycle

The following figure illustrates how a typical application might manage its configuration data with a `Properties` object over the course of its execution.




- `Starting Up`
  The actions given in the first three boxes occur when the application is starting up. First, the application loads the default properties from a well-known location into a `Properties` object. Normally, the default properties are stored in a file on disk along with the `.class` and other resource files for the application.
  Next, the application creates another `Properties` object and loads the properties that were saved from the last time the application was run. Many applications store properties on a per-user basis, so the properties loaded in this step are usually in a specific file in a particular directory maintained by this application in the user's home directory. Finally, the application uses the default and remembered properties to initialize itself.
  The key here is consistency. The application must always load and save properties to the same location so that it can find them the next time it's executed.
- `Running`
  During the execution of the application, the user may change some settings, perhaps in a Preferences window, and the `Properties` object is updated to reflect these changes. If the users changes are to be remembered in future sessions, they must be saved.
- `Exiting`
  Upon exiting, the application saves the properties to its well-known location, to be loaded again when the application is next started up.

## Setting Up the Properties Object

The following Java code performs the first two steps described in the previous section: loading the default properties and loading the remembered properties:

```
. . .
// create and load default properties
Properties defaultProps = new Properties();
FileInputStream in = new FileInputStream("defaultProperties");
defaultProps.load(in);
in.close();

// create application properties with default
Properties applicationProps = new Properties(defaultProps);
```

```
// now load properties
// from last invocation
in = new FileInputStream("appProperties");
applicationProps.load(in);
in.close();
. . .
```

First, the application sets up a default `Properties` object. This object contains the set of properties to use if values are not explicitly set elsewhere. Then the load method reads the default values from a file on disk named `defaultProperties`.

Next, the application uses a different constructor to create a second `Properties` object, `applicationProps`, whose default values are contained in `defaultProps`. The defaults come into play when a property is being retrieved. If the property can't be found in `applicationProps`, then its default list is searched.

Finally, the code loads a set of properties into `applicationProps` from a file named `appProperties`. The properties in this file are those that were saved from the application the last time it was invoked, as explained in the next section.

## Saving Properties

The following example writes out the application properties from the previous example using `Properties.store`. The default properties don't need to be saved each time because they never change.

```
FileOutputStream out = new FileOutputStream("appProperties");
applicationProps.store(out, "---No Comment---");
out.close();
```

The `store` method needs a stream to write to, as well as a string that it uses as a comment at the top of the output.

## Getting Property Information

Once the application has set up its `Properties` object, the application can query the object for information about various keys and values that it contains. An application gets information from a `Properties` object after start up so that it can initialize itself based on choices made by the user. The `Properties` class has several methods for getting property information:

- `contains(Object value)` and `containsKey(Object key)`
  Returns `true` if the value or the key is in the `Properties` object. `Properties` inherits these methods from `Hashtable`. Thus they accept `Object` arguments, but only `String` values should be used.
- `getProperty(String key)` and `getProperty(String key, String default)`
  Returns the value for the specified property. The second version provides for a default value. If the key is not found, the default is returned.

- `list(PrintStream s)` and `list(PrintWriter w)`
  Writes all of the properties to the specified stream or writer. This is useful for debugging.
- `elements()`, `keys()`, and `propertyNames()`
  Returns an `Enumeration` containing the keys or values (as indicated by the method name) contained in the `Properties` object. The `keys` method only returns the keys for the object itself; the `propertyNames` method returns the keys for default properties as well.
- `stringPropertyNames()`
  Like `propertyNames`, but returns a `Set<String>`, and only returns names of properties where both key and value are strings. Note that the `Set` object is not backed by the `Properties` object, so changes in one do not affect the other.
- `size()`
  Returns the current number of key/value pairs.

## Setting Properties

A user's interaction with an application during its execution may impact property settings. These changes should be reflected in the `Properties` object so that they are saved when the application exits (and calls the `store` method). The following methods change the properties in a `Properties` object:

- `setProperty(String key, String value)`
  Puts the key/value pair in the `Properties` object.
- `remove(Object key)`
  Removes the key/value pair associated with key.

---

**Note:** Some of the methods described above are defined in `Hashtable`, and thus accept key and value argument types other than `String`. Always use `String`s for keys and values, even if the method allows other types. Also do not invoke `Hashtable.set` or `Hastable.setAll` on `Properties` objects; always use `Properties.setProperty`.

---

# Command-Line Arguments

A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.

The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run. For example, suppose a Java application called `Sort` sorts lines in a file. To sort the data in a file named `friends.txt`, a user would enter:

```
java Sort friends.txt
```

When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of `String`s. In the previous example, the command-line arguments passed to the `Sort` application in an array that contains a single `String`: `"friends.txt"`.

## Echoing Command-Line Arguments

The `Echo` example displays each of its command-line arguments on a line by itself:

```
public class Echo {
    public static void main (String[] args) {
        for (String s: args) {
            System.out.println(s);
        }
    }
}
```

The following example shows how a user might run `Echo`. User input is in italics.

```
java Echo Drink Hot Java
Drink
Hot
Java
```

Note that the application displays each word — `Drink`, `Hot`, and `Java` — on a line by itself. This is because the space character separates command-line arguments. To have `Drink`, `Hot`, and `Java` interpreted as a single argument, the user would join them by enclosing them within quotation marks.

```
java Echo "Drink Hot Java"
Drink Hot Java
```

## Parsing Numeric Command-Line Arguments

If an application needs to support a numeric command-line argument, it must convert a `String` argument that represents a number, such as "34", to a numeric value. Here is a code snippet that

converts a command-line argument to an `int`:

```
int firstArg;
if (args.length > 0) {
    try {
        firstArg = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Argument" + args[0] + " must be an integer.");
        System.exit(1);
    }
}
```

`parseInt` throws a `NumberFormatException` if the format of `args[0]` isn't valid. All of the `Number` classes — `Integer`, `Float`, `Double`, and so on — have `parseXXX` methods that convert a `String` representing a number to an object of their type.

# Environment Variables

Many operating systems use *environment variables* to pass configuration information to applications. Like properties in the Java platform, environment variables are key/value pairs, where both the key and the value are strings. The conventions for setting and using environment variables vary between operating systems, and also between command line interpreters. To learn how to pass environment variables to applications on your system, refer to your system documentation.

## Querying Environment Variables

On the Java platform, an application uses `System.getenv` to retrieve environment variable values. Without an argument, `getenv` returns a read-only instance of `java.util.Map`, where the map keys are the environment variable names, and the map values are the environment variable values. This is demonstrated in the `EnvMap` example:

```
import java.util.Map;

public class EnvMap {
    public static void main (String[] args) {
        Map<String, String> env = System.getenv();
        for (String envName : env.keySet()) {
            System.out.format("%s=%s%n",
                              envName,
                              env.get(envName));
        }
    }
}
```

With a `String` argument, `getenv` returns the value of the specified variable. If the variable is not defined, `getenv` returns `null`. The `Env` example uses `getenv` this way to query specific environment variables, specified on the command line:

```
public class Env {
    public static void main (String[] args) {
        for (String env: args) {
            String value = System.getenv(env);
            if (value != null) {
                System.out.format("%s=%s%n",
                                  env, value);
            } else {
                System.out.format("%s is"
                    + " not assigned.%n", env);
            }
        }
    }
}
```

## Passing Environment Variables to New Processes

When a Java application uses a `ProcessBuilder` object to create a new process, the default set of

environment variables passed to the new process is the same set provided to the application's virtual machine process. The application can change this set using `ProcessBuilder.environment`.

## Platform Dependency Issues

There are many subtle differences between the way environment variables are implemented on different systems. For example, Windows ignores case in environment variable names, while UNIX does not. The way environment variables are used also varies. For example, Windows provides the user name in an environment variable called `USERNAME`, while UNIX implementations might provide the user name in `USER`, `LOGNAME`, or both.

To maximize portability, never refer to an environment variable when the same value is available in a system property. For example, if the operating system provides a user name, it will always be available in the system property `user.name`.

# Other Configuration Utilities

Here is a summary of some other configuration utilities.

The *Preferences API* allows applications to store and retrieve configuration data in an implementation-dependent backing store. Asynchronous updates are supported, and the same set of preferences can be safely updated by multiple threads and even multiple applications. For more information, refer to the Preferences API Guide.

An application deployed in a *JAR archive* uses a *manifest* to describe the contents of the archive. For more information, refer to the Packaging Programs in JAR Files lesson.

The configuration of a *Java Web Start application* is contained in a *JNLP file*. For more information, refer to the Java Web Start lesson.

The configuration of a *Java Plug-in applet* is partially determined by the HTML tags used to embed the applet in the web page. Depending on the applet and the browser, these tags can include `<applet>`, `<object>`, `<embed>`, and `<param>`. For more information, refer to the Java Applets lesson.

The class `java.util.ServiceLoader` provides a simple *service provider* facility. A service provider is an implementation of a *service* — a well-known set of interfaces and (usually abstract) classes. The classes in a service provider typically implement the interfaces and subclass the classes defined in the service. Service providers can be installed as extensions (see The Extension Mechanism). Providers can also be made available by adding them to the class path or by some other platform-specific means.

# System Utilities

The `System` class implements a number of system utilities. Some of these have already been covered in the previous section on Configuration Utilities. This section covers some of the other system utilities.

# Command-Line I/O Objects

`System` provides several predefined I/O objects that are useful in a Java application that is meant to be launched from the command line. These implement the Standard I/O streams provided by most operating systems, and also a console object that is useful for entering passwords. For more information, refer to I/O from the Command Line in the Basic I/O lesson.

# System Properties

In Properties, we examined the way an application can use `Properties` objects to maintain its configuration. The Java platform itself uses a `Properties` object to maintain its own configuration. The `System` class maintains a `Properties` object that describes the configuration of the current working environment. System properties include information about the current user, the current version of the Java runtime, and the character used to separate components of a file path name.

The following table describes some of the most important system properties

| Key | Meaning |
|---|---|
| `"file.separator"` | Character that separates components of a file path. This is "/" on UNIX and "\" on Windows. |
| `"java.class.path"` | Path used to find directories and JAR archives containing class files. Elements of the class path are separated by a platform-specific character specified in the `path.separator` property. |
| `"java.home"` | Installation directory for Java Runtime Environment (JRE) |
| `"java.vendor"` | JRE vendor name |
| `"java.vendor.url"` | JRE vendor URL |
| `"java.version"` | JRE version number |
| `"line.separator"` | Sequence used by operating system to separate lines in text files |
| `"os.arch"` | Operating system architecture |
| `"os.name"` | Operating system name |
| `"os.version"` | Operating system version |
| `"path.separator"` | Path separator character used in `java.class.path` |
| `"user.dir"` | User working directory |
| `"user.home"` | User home directory |
| `"user.name"` | User account name |

---

**Security consideration:** Access to system properties can be restricted by the Security Manager. This is most often an issue in applets, which are prevented from reading some system properties, and from writing *any* system properties. For more on accessing system properties in applets, refer to System Properties in the Doing More With Java Rich Internet Applications lesson.

---

# Reading System Properties

The `System` class has two methods used to read system properties: `getProperty` and `getProperties`.

The `System` class has two different versions of `getProperty`. Both retrieve the value of the property named in the argument list. The simpler of the two `getProperty` methods takes a single argument, a property key For example, to get the value of `path.separator`, use the following statement:

```
System.getProperty("path.separator");
```

The `getProperty` method returns a string containing the value of the property. If the property does not exist, this version of `getProperty` returns null.

The other version of `getProperty` requires two `String` arguments: the first argument is the key to look up and the second argument is a default value to return if the key cannot be found or if it has no value. For example, the following invocation of `getProperty` looks up the `System` property called `subliminal.message`. This is not a valid system property, so instead of returning null, this method returns the default value provided as a second argument: `"Buy StayPuft Marshmallows!"`

```
System.getProperty("subliminal.message", "Buy StayPuft Marshmallows!");
```

The last method provided by the `System` class to access property values is the `getProperties` method, which returns a `Properties` object. This object contains a complete set of system property definitions.

## Writing System Properties

To modify the existing set of system properties, use `System.setProperties`. This method takes a `Properties` object that has been initialized to contain the properties to be set. This method replaces the entire set of system properties with the new set represented by the `Properties` object.

---

**Warning:** Changing system properties is potentially dangerous and should be done with discretion. Many system properties are not reread after start-up and are there for informational purposes. Changing some properties may have unexpected side-effects.

---

The next example, `PropertiesTest`, creates a `Properties` object and initializes it from `myProperties.txt` .

```
subliminal.message=Buy StayPuft Marshmallows!
```

`PropertiesTest` then uses `System.setProperties` to install the new `Properties` objects as the current set of system properties.

```
```

```
import java.io.FileInputStream;
import java.util.Properties;

public class PropertiesTest {
    public static void main(String[] args)
        throws Exception {

        // set up new properties object
        // from file "myProperties.txt"
        FileInputStream propFile =
            new FileInputStream( "myProperties.txt");
        Properties p =
            new Properties(System.getProperties());
        p.load(propFile);

        // set the system properties
        System.setProperties(p);
        // display new properties
        System.getProperties().list(System.out);
    }
}
```

Note how `PropertiesTest` creates the `Properties` object, `p`, which is used as the argument to `setProperties`:

```
Properties p = new Properties(System.getProperties());
```

This statement initializes the new properties object, `p`, with the current set of system properties, which in the case of this small application, is the set of properties initialized by the runtime system. Then the application loads additional properties into `p` from the file `myProperties.txt` and sets the system properties to `p`. This has the effect of adding the properties listed in `myProperties.txt` to the set of properties created by the runtime system at startup. Note that an application can create `p` without any default `Properties` object, like this:

```
Properties p = new Properties();
```

Also note that the value of system properties can be overwritten! For example, if `myProperties.txt` contains the following line, the `java.vendor` system property will be overwritten:

```
java.vendor=Acme Software Company
```

In general, be careful not to overwrite system properties.

The `setProperties` method changes the set of system properties for the current running application. These changes are not persistent. That is, changing the system properties within an application will not affect future invocations of the Java interpreter for this or any other application. The runtime system re-initializes the system properties each time its starts up. If changes to system properties are to be persistent, then the application must write the values to some file before exiting and read them in again upon startup.

# The Security Manager

A *security manager* is an object that defines a security policy for an application. This policy specifies actions that are unsafe or sensitive. Any actions not allowed by the security policy cause a `SecurityException` to be thrown. An application can also query its security manager to discover which actions are allowed.

Typically, a web applet runs with a security manager provided by the browser or Java Web Start plugin. Other kinds of applications normally run without a security manager, unless the application itself defines one. If no security manager is present, the application has no security policy and acts without restrictions.

This section explains how an application interacts with an existing security manager. For more detailed information, including information on how to design a security manager, refer to the Security Guide.

## Interacting with the Security Manager

The security manager is an object of type `SecurityManager`; to obtain a reference to this object, invoke `System.getSecurityManager`.

```
SecurityManager appsm = System.getSecurityManager();
```

If there is no security manager, this method returns `null`.

Once an application has a reference to the security manager object, it can request permission to do specific things. Many classes in the standard libraries do this. For example, `System.exit`, which terminates the Java virtual machine with an exit status, invokes `SecurityManager.checkExit` to ensure that the current thread has permission to shut down the application.

The SecurityManager class defines many other methods used to verify other kinds of operations. For example, `SecurityManager.checkAccess` verifies thread accesses, and `SecurityManager.checkPropertyAccess` verifies access to the specified property. Each operation or group of operations has its own `checkXXX()` method.

In addition, the set of `checkXXX()` methods represents the set of operations that are already subject to the protection of the security manager. Typically, an application does not have to directly invoke any `checkXXX()` methods.

## Recognizing a Security Violation

Many actions that are routine without a security manager can throw a `SecurityException` when run with a security manager. This is true even when invoking a method that isn't documented as throwing `SecurityException`. For example, consider the following code used to read a file:

```
reader = new FileReader("xanadu.txt");
```

In the absence of a security manager, this statement executes without error, provided `xanadu.txt` exists and is readable. But suppose this statement is inserted in a web applet, which typically runs under a security manager that does not allow file input. The following error messages might result:

```
appletviewer fileApplet.html
Exception in thread "AWT-EventQueue-1" java.security.AccessControlException: access denied (java.io.FilePermission
characteroutput.txt write)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
        at java.security.AccessController.checkPermission(AccessController.java:546)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
        at java.lang.SecurityManager.checkWrite(SecurityManager.java:962)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:169)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:70)
        at java.io.FileWriter.<init>(FileWriter.java:46)
...
```

Note that the specific exception thrown in this case, `java.security.AccessControlException`, is a subclass of `SecurityException`.

# Miscellaneous Methods in System

This section describes some of the methods in `System` that aren't covered in the previous sections.

The `arrayCopy` method efficiently copies data between arrays. For more information, refer to Arrays in the Language Basics lesson.

The `currentTimeMillis` and `nanoTime` methods are useful for measuring time intervals during execution of an application. To measure a time interval in milliseconds, invoke `currentTimeMillis` twice, at the beginning and end of the interval, and subtract the first value returned from the second. Similarly, invoking `nanoTime` twice measures an interval in nanoseconds.

---

**Note:** The accuracy of both `currentTimeMillis` and `nanoTime` is limited by the time services provided by the operating system. Do not assume that `currentTimeMillis` is accurate to the nearest millisecond or that `nanoTime` is accurate to the nearest nanosecond. Also, neither `currentTimeMillis` nor `nanoTime` should be used to determine the current time. Use a high-level method, such as `java.util.Calendar.getInstance`.

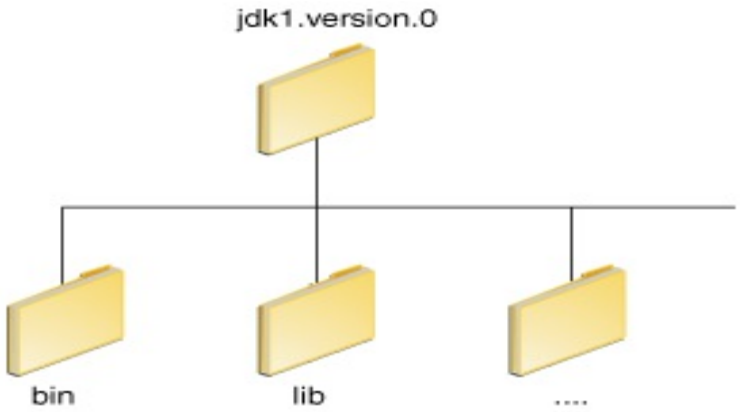
---

The `exit` method causes the Java virtual machine to shut down, with an integer exit status specified by the argument. The exit status is available to the process that launched the application. By convention, an exit status of `0` indicates normal termination of the application, while any other value is an error code.

# PATH and CLASSPATH

This section explains how to use the `PATH` and `CLASSPATH` environment variables on Microsoft Windows, Solaris, and Linux. Consult the installation instructions included with your installation of the Java Development Kit (JDK) software bundle for current information.

After installing the software, the JDK directory will have the structure shown below.




The `bin` directory contains both the compiler and the launcher.

## Update the PATH Environment Variable (Microsoft Windows)

You can run Java applications just fine without setting the `PATH` environment variable. Or, you can optionally set it as a convenience.

Set the `PATH` environment variable if you want to be able to conveniently run the executables (`javac.exe`, `java.exe`, `javadoc.exe`, and so on) from any directory without having to type the full path of the command. If you do not set the `PATH` variable, you need to specify the full path to the executable every time you run it, such as:

```
C:\Java\jdk1.7.0\bin\javac MyClass.java
```

The `PATH` environment variable is a series of directories separated by semicolons (`;`). Microsoft Windows looks for programs in the `PATH` directories in order, from left to right. You should have only one `bin` directory for the JDK in the path at a time (those following the first are ignored), so if one is already present, you can update that particular entry.

The following is an example of a `PATH` environment variable:

```
C:\Java\jdk1.7.0\bin;C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem
```

It is useful to set the `PATH` environment variable permanently so it will persist after rebooting. To make a permanent change to the `PATH` variable, use the **System** icon in the Control Panel. The precise procedure varies depending on the version of Windows:

**Windows XP**

**1.** Select **Start**, select **Control Panel**. double click **System**, and select the **Advanced** tab.
**2.** Click **Environment Variables**. In the section **System Variables**, find the `PATH` environment variable and select it. Click **Edit**. If the `PATH` environment variable does not exist, click `New`.
**3.** In the **Edit System Variable** (or **New System Variable**) window, specify the value of the `PATH` environment variable. Click **OK**. Close all remaining windows by clicking **OK**.

**Windows Vista:**

**1.** From the desktop, right click the **My Computer** icon.
**2.** Choose **Properties** from the context menu.
**3.** Click the **Advanced** tab (**Advanced system settings** link in Vista).
**4.** Click **Environment Variables**. In the section **System Variables**, find the `PATH` environment variable and select it. Click **Edit**. If the `PATH` environment variable does not exist, click `New`.
**5.** In the **Edit System Variable** (or **New System Variable**) window, specify the value of the `PATH` environment variable. Click **OK**. Close all remaining windows by clicking **OK**.

**Windows 7:**

**1.** From the desktop, right click the **Computer** icon.
**2.** Choose **Properties** from the context menu.
**3.** Click the **Advanced system settings** link.
**4.** Click **Environment Variables**. In the section **System Variables**, find the `PATH` environment variable and select it. Click **Edit**. If the `PATH` environment variable does not exist, click `New`.
**5.** In the **Edit System Variable** (or **New System Variable**) window, specify the value of the `PATH` environment variable. Click **OK**. Close all remaining windows by clicking **OK**.

---

**Note:** You may see a `PATH` environment variable similar to the following when editing it from the Control Panel:

```
%JAVA_HOME%\bin;%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem
```

Variables enclosed in percentage signs (`%`) are existing environment variables. If one of these variables is listed in the **Environment Variables** window from the Control Panel (such as `JAVA_HOME`), then you can edit its value. If it does not appear, then it is a special environment variable that the operating system has defined. For example, `SystemRoot` is the location of the Microsoft Windows system folder. To obtain the value of a environment variable, enter the following at a command prompt. (This example obtains the value of the `SystemRoot` environment variable):

```
echo %SystemRoot%
```

---

## Update the PATH Variable (Solaris and Linux)

You can run the JDK just fine without setting the `PATH` variable, or you can optionally set it as a convenience. However, you should set the path variable if you want to be able to run the executables (`javac`, `java`, `javadoc`, and so on) from any directory without having to type the full path of the command. If you do not set the `PATH` variable, you need to specify the full path to the executable every time you run it, such as:

```
% /usr/local/jdk1.7.0/bin/javac MyClass.java
```

To find out if the path is properly set, execute:

```
% java -version
```

This will print the version of the `java` tool, if it can find it. If the version is old or you get the error **java: Command not found**, then the path is not properly set.

To set the path permanently, set the path in your startup file.

For C shell (`csh`), edit the startup file (`~/.cshrc`):

```
set path=(/usr/local/jdk1.7.0/bin $path)
```

For `bash`, edit the startup file (`~/.bashrc`):

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
export PATH
```

For `ksh`, the startup file is named by the environment variable, `ENV`. To set the path:

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
export PATH
```

For `sh`, edit the profile file (`~/.profile`):

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
export PATH
```

Then load the startup file and verify that the path is set by repeating the `java` command:

For C shell (`csh`):

```
% source ~/.cshrc
% java -version
```

For `ksh`, `bash`, or `sh`:

```
% . /.profile
% java -version
```

## Checking the CLASSPATH variable (All platforms)

The `CLASSPATH` variable is one way to tell applications, including the JDK tools, where to look for user classes. (Classes that are part of the JRE, JDK platform, and extensions should be defined through other means, such as the bootstrap class path or the extensions directory.)

The preferred way to specify the class path is by using the `-cp` command line switch. This allows the `CLASSPATH` to be set individually for each application without affecting other applications. *Setting the `CLASSPATH` can be tricky and should be performed with care.*

The default value of the class path is ".", meaning that only the current directory is searched. Specifying either the CLASSPATH variable or the `-cp` command line switch overrides this value.

To check whether `CLASSPATH` is set on Microsoft Windows NT/2000/XP, execute the following:

```
C:> echo %CLASSPATH%
```

On Solaris or Linux, execute the following:

```
% echo $CLASSPATH
```

If `CLASSPATH` is not set you will get a **CLASSPATH: Undefined variable** error (Solaris or Linux) or simply **%CLASSPATH%** (Microsoft Windows NT/2000/XP).

To modify the `CLASSPATH`, use the same procedure you used for the `PATH` variable.

Class path wildcards allow you to include an entire directory of `.jar` files in the class path without explicitly naming them individually. For more information, including an explanation of class path wildcards, and a detailed description on how to clean up the `CLASSPATH` environment variable, see the Setting the Class Path technical note.

# Questions and Exercises: The Platform Environment

## Questions

1. A programmer installs a new library contained in a .jar file. In order to access the library from his code, he sets the CLASSPATH environment variable to point to the new .jar file. Now he finds that he gets an error message when he tries to launch simple applications:

```
java Hello
Exception in thread "main" java.lang.NoClassDefFoundError: Hello
```

In this case, the `Hello` class is compiled into a .class file in the current directory — yet the `java` command can't seem to find it. What's going wrong?

## Exercises

1. Write an application, `PersistentEcho`, with the following features:

- If `PersistentEcho` is run with command line arguments, it prints out those arguments. It also saves the string printed out to a property, and saves the property to a file called `PersistentEcho.txt`
- If `PersistentEcho` is run with no command line arguments, it looks for an environment variable called PERSISTENTECHO. If that variable exists, `PersistentEcho` prints out its value, and also saves the value in the same way it does for command line arguments.
- If `PersistentEcho` is run with no command line arguments, and the PERSISTENTECHO environment variable is not defined, it retrieves the property value from `PersistentEcho.txt` and prints that out.

Check your answers.

## Questions

**Question 1.** A programmer installs a new library contained in a .jar file. In order to access the library from his code, he sets the CLASSPATH environment variable to point to the new .jar file. Now he finds that he gets an error message when he tries to launch simple applications:

```
java Hello
Exception in thread "main" java.lang.NoClassDefFoundError: Hello
```

In this case, the `Hello` class is compiled into a .class file in the current directory — yet the `java` command can't seem to find it. What's going wrong?

**Answer 1.** A class is only found if it appears in the class path. By default, the class path consists of the current directory. If the CLASSPATH environment variable is set, and doesn't include the current directory, the launcher can no longer find classes in the current directory. The solution is to change the CLASSPATH variable to include the current directory. For example, if the CLASSPATH value is `c:\java\newLibrary.jar` (Windows) or `/home/me/newLibrary.jar` (UNIX or Linux) it needs to be changed to `.;c:\java\newLibrary.jar` or `.:/home/me/newLibrary.jar`.

## Exercises

### Exercise 1.

Write an application, `PersistentEcho`, with the following features:

- If `PersistentEcho` is run with command line arguments, it prints out those arguments. It also saves the string printed out to a property, and saves the property to a file called `PersistentEcho.txt`
- If `PersistentEcho` is run with no command line arguments, it looks for an environment variable called PERSISTENTECHO. If that variable exists, `PersistentEcho` prints out its value, and also saves the value in the same way it does for command line arguments.
- If `PersistentEcho` is run with no command line arguments, and the PERSISTENTECHO environment variable is not defined, it retrieves the property value from `PersistentEcho.txt` and prints that out.

### Answer 1.

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class PersistentEcho {
    public static void main (String[] args) {
        String argString = "";
        boolean notProperty = true;

        // Are there arguments?
```

```
        // If so retrieve them.
        if (args.length > 0) {
            for (String arg: args) {
                argString += arg + " ";
            }
            argString = argString.trim();
        }
        // No arguments, is there
        // an environment variable?
        // If so, //retrieve it.
        else if ((argString = System.getenv("PERSISTENTECHO")) != null) {}
        // No environment variable
        // either. Retrieve property value.
        else {
            notProperty = false;
            // Set argString to null.
            // If it's still null after
            // we exit the try block,
            // we've failed to retrieve
            // the property value.
            argString = null;
            FileInputStream fileInputStream = null;
            try {
                fileInputStream =
                    new FileInputStream("PersistentEcho.txt");
                Properties inProperties
                    = new Properties();
                inProperties.load(fileInputStream);
                argString = inProperties.getProperty("argString");
            } catch (IOException e) {
                System.err.println("Can't read property file.");
                System.exit(1);
            } finally {
                if (fileInputStream != null) {
                    try {
                        fileInputStream.close();
                    } catch(IOException e) {};
                }
            }
        }
        if (argString == null) {
            System.err.println("Couldn't find argString property");
            System.exit(1);
        }

        // Somehow, we got the
        // value. Echo it already!
        System.out.println(argString);

        // If we didn't retrieve the
        // value from the property,
        // save it //in the property.
        if (notProperty) {
            Properties outProperties =
                new Properties();
            outProperties.setProperty("argString",
                                      argString);
            FileOutputStream fileOutputStream = null;
            try {
                fileOutputStream =
                    new FileOutputStream("PersistentEcho.txt");
                outProperties.store(fileOutputStream,
                        "PersistentEcho properties");
            } catch (IOException e) {}
            finally {
                if (fileOutputStream != null) {
                    try {
                        fileOutputStream.close();
                    } catch(IOException e) {};
```

```
                    }
                }
            }
        }
}
```

# Lesson: Regular Expressions

This lesson explains how to use the `java.util.regex` API for pattern matching with regular expressions. Although the syntax accepted by this package is similar to the Perl programming language, knowledge of Perl is not a prerequisite. This lesson starts with the basics, and gradually builds to cover more advanced techniques.



---

**Note:** See online version of topics in this ebook to download complete source code.

---

# Introduction

## What Are Regular Expressions?

*Regular expressions* are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used to search, edit, or manipulate text and data. You must learn a specific syntax to create regular expressions — one that goes beyond the normal syntax of the Java programming language. Regular expressions vary in complexity, but once you understand the basics of how they're constructed, you'll be able to decipher (or create) any regular expression.

This trail teaches the regular expression syntax supported by the `java.util.regex` API and presents several working examples to illustrate how the various objects interact. In the world of regular expressions, there are many different flavors to choose from, such as grep, Perl, Tcl, Python, PHP, and awk. The regular expression syntax in the `java.util.regex` API is most similar to that found in Perl.

## How Are Regular Expressions Represented in This Package?

The `java.util.regex` package primarily consists of three classes: `Pattern`, `Matcher`, and `PatternSyntaxException`.

- A `Pattern` object is a compiled representation of a regular expression. The `Pattern` class provides no public constructors. To create a pattern, you must first invoke one of its `public static compile` methods, which will then return a `Pattern` object. These methods accept a regular expression as the first argument; the first few lessons of this trail will teach you the required syntax.
- A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string. Like the `Pattern` class, `Matcher` defines no public constructors. You obtain a `Matcher` object by invoking the `matcher` method on a `Pattern` object.
- A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.

The last few lessons of this trail explore each class in detail. But first, you must understand how regular expressions are actually constructed. Therefore, the next section introduces a simple test harness that will be used repeatedly to explore their syntax.

# Test Harness

This section defines a reusable test harness, `RegexTestHarness.java` , for exploring the regular expression constructs supported by this API. The command to run this code is `java RegexTestHarness`; no command-line arguments are accepted. The application loops repeatedly, prompting the user for a regular expression and input string. Using this test harness is optional, but you may find it convenient for exploring the test cases discussed in the following pages.

```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTestHarness {

    public static void main(String[] args){
        Console console = System.console();
        if (console == null) {
            System.err.println("No console.");
            System.exit(1);
        }
        while (true) {

            Pattern pattern =
            Pattern.compile(console.readLine("%nEnter your regex: "));

            Matcher matcher =
            pattern.matcher(console.readLine("Enter input string to search: "));

            boolean found = false;
            while (matcher.find()) {
                console.format("I found the text" +
                    " \"%s\" starting at " +
                    "index %d and ending at index %d.%n",
                    matcher.group(),
                    matcher.start(),
                    matcher.end());
                found = true;
            }
            if(!found){
                console.format("No match found.%n");
            }
        }
    }
}
```

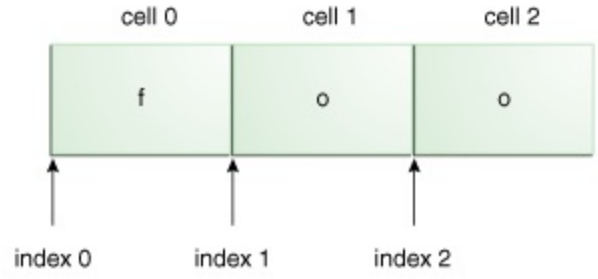
Before continuing to the next section, save and compile this code to ensure that your development environment supports the required packages.

## String Literals

The most basic form of pattern matching supported by this API is the match of a string literal. For example, if the regular expression is `foo` and the input string is `foo`, the match will succeed because the strings are identical. Try this out with the test harness:

```
Enter your regex: foo
Enter input string to search: foo
I found the text foo starting at index 0 and ending at index 3.
```

This match was a success. Note that while the input string is 3 characters long, the start index is 0 and the end index is 3. By convention, ranges are inclusive of the beginning index and exclusive of the end index, as shown in the following figure:




The string literal foo, with numbered cells and index values.

Each character in the string resides in its own *cell*, with the index positions pointing between each cell. The string "foo" starts at index 0 and ends at index 3, even though the characters themselves only occupy cells 0, 1, and 2.

With subsequent matches, you'll notice some overlap; the start index for the next match is the same as the end index of the previous match:

```
Enter your regex: foo
Enter input string to search: foofoofoo
I found the text foo starting at index 0 and ending at index 3.
I found the text foo starting at index 3 and ending at index 6.
I found the text foo starting at index 6 and ending at index 9.
```

## Metacharacters

This API also supports a number of special characters that affect the way a pattern is matched. Change the regular expression to `cat.` and the input string to `cats`. The output will appear as follows:

```
Enter your regex: cat.
Enter input string to search: cats
I found the text cats starting at index 0 and ending at index 4.
```

The match still succeeds, even though the dot "." is not present in the input string. It succeeds because the dot is a *metacharacter* — a character with special meaning interpreted by the matcher. The metacharacter "." means "any character" which is why the match succeeds in this example.

The metacharacters supported by this API are: `<([{\^-=$!|]})?*+.>`

---

**Note:** In certain situations the special characters listed above will *not* be treated as metacharacters. You'll encounter this as you learn more about how regular expressions are constructed. You can, however, use this list to check whether or not a specific character will ever be considered a metacharacter. For example, the characters `@` and `#` never carry a special meaning.

---

There are two ways to force a metacharacter to be treated as an ordinary character:

- precede the metacharacter with a backslash, or
- enclose it within `\Q` (which starts the quote) and `\E` (which ends it).

When using this technique, the `\Q` and `\E` can be placed at any location within the expression, provided that the `\Q` comes first.

# Character Classes

If you browse through the `Pattern` class specification, you'll see tables summarizing the supported regular expression constructs. In the "Character Classes" section you'll find the following:

| Construct | Description |
|---|---|
| `[abc]` | a, b, or c (simple class) |
| `[^abc]` | Any character except a, b, or c (negation) |
| `[a-zA-Z]` | a through z, or A through Z, inclusive (range) |
| `[a-d[m-p]]` | a through d, or m through p: [a-dm-p] (union) |
| `[a-z&&[def]]` | d, e, or f (intersection) |
| `[a-z&&[^bc]]` | a through z, except for b and c: [ad-z] (subtraction) |
| `[a-z&&[^m-p]]` | a through z, and not m through p: [a-lq-z] (subtraction) |

The left-hand column specifies the regular expression constructs, while the right-hand column describes the conditions under which each construct will match.

---

**Note:** The word "class" in the phrase "character class" does not refer to a `.class` file. In the context of regular expressions, a *character class* is a set of characters enclosed within square brackets. It specifies the characters that will successfully match a single character from a given input string.

---

## Simple Classes

The most basic form of a character class is to simply place a set of characters side-by-side within square brackets. For example, the regular expression `[bcr]at` will match the words "bat", "cat", or "rat" because it defines a character class (accepting either "b", "c", or "r") as its first character.

```
Enter your regex: [bcr]at
Enter input string to search: bat
I found the text "bat" starting at index 0 and ending at index 3.

Enter your regex: [bcr]at
Enter input string to search: cat
I found the text "cat" starting at index 0 and ending at index 3.

Enter your regex: [bcr]at
Enter input string to search: rat
I found the text "rat" starting at index 0 and ending at index 3.

Enter your regex: [bcr]at
Enter input string to search: hat
No match found.
```

In the above examples, the overall match succeeds only when the first letter matches one of the

characters defined by the character class.

### Negation

To match all characters *except* those listed, insert the "^" metacharacter at the beginning of the character class. This technique is known as *negation*.

```
Enter your regex: [^bcr]at
Enter input string to search: bat
No match found.

Enter your regex: [^bcr]at
Enter input string to search: cat
No match found.

Enter your regex: [^bcr]at
Enter input string to search: rat
No match found.

Enter your regex: [^bcr]at
Enter input string to search: hat
I found the text "hat" starting at index 0 and ending at index 3.
```

The match is successful only if the first character of the input string does *not* contain any of the characters defined by the character class.

### Ranges

Sometimes you'll want to define a character class that includes a range of values, such as the letters "a through h" or the numbers "1 through 5". To specify a range, simply insert the "-" metacharacter between the first and last character to be matched, such as `[1-5]` or `[a-h]`. You can also place different ranges beside each other within the class to further expand the match possibilities. For example, `[a-zA-Z]` will match any letter of the alphabet: a to z (lowercase) or A to Z (uppercase).

Here are some examples of ranges and negation:

```
Enter your regex: [a-c]
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: [a-c]
Enter input string to search: b
I found the text "b" starting at index 0 and ending at index 1.

Enter your regex: [a-c]
Enter input string to search: c
I found the text "c" starting at index 0 and ending at index 1.

Enter your regex: [a-c]
Enter input string to search: d
No match found.

Enter your regex: foo[1-5]
Enter input string to search: foo1
I found the text "foo1" starting at index 0 and ending at index 4.

Enter your regex: foo[1-5]
```

```
Enter input string to search: foo5
I found the text "foo5" starting at index 0 and ending at index 4.

Enter your regex: foo[1-5]
Enter input string to search: foo6
No match found.

Enter your regex: foo[^1-5]
Enter input string to search: foo1
No match found.

Enter your regex: foo[^1-5]
Enter input string to search: foo6
I found the text "foo6" starting at index 0 and ending at index 4.
```

### Unions

You can also use *unions* to create a single character class comprised of two or more separate character classes. To create a union, simply nest one class inside the other, such as `[0-4[6-8]]`. This particular union creates a single character class that matches the numbers 0, 1, 2, 3, 4, 6, 7, and 8.

```
Enter your regex: [0-4[6-8]]
Enter input string to search: 0
I found the text "0" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]
Enter input string to search: 5
No match found.

Enter your regex: [0-4[6-8]]
Enter input string to search: 6
I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]
Enter input string to search: 8
I found the text "8" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]
Enter input string to search: 9
No match found.
```

### Intersections

To create a single character class matching only the characters common to all of its nested classes, use `&&`, as in `[0-9&&[345]]`. This particular intersection creates a single character class matching only the numbers common to both character classes: 3, 4, and 5.

```
Enter your regex: [0-9&&[345]]
Enter input string to search: 3
I found the text "3" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]
Enter input string to search: 4
I found the text "4" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]
Enter input string to search: 5
I found the text "5" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]
```

```
Enter input string to search: 2
No match found.

Enter your regex: [0-9&&[345]]
Enter input string to search: 6
No match found.
```

And here's an example that shows the intersection of two ranges:

```
Enter your regex: [2-8&&[4-6]]
Enter input string to search: 3
No match found.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 4
I found the text "4" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 5
I found the text "5" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 6
I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 7
No match found.
```

### Subtraction

Finally, you can use *subtraction* to negate one or more nested character classes, such as `[0-9&&[^345]]`. This example creates a single character class that matches everything from 0 to 9, *except* the numbers 3, 4, and 5.

```
Enter your regex: [0-9&&[^345]]
Enter input string to search: 2
I found the text "2" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 3
No match found.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 4
No match found.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 5
No match found.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 6
I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 9
I found the text "9" starting at index 0 and ending at index 1.
```

Now that we've covered how character classes are created, You may want to review the Character Classes table before continuing with the next section.

# Predefined Character Classes

The `Pattern` API contains a number of useful *predefined character classes*, which offer convenient shorthands for commonly used regular expressions:

| Construct | Description |
|---|---|
| . | Any character (may or may not match line terminators) |
| `\d` | A digit: `[0-9]` |
| `\D` | A non-digit: `[^0-9]` |
| `\s` | A whitespace character: `[ \t\n\x0B\f\r]` |
| `\S` | A non-whitespace character: `[^\s]` |
| `\w` | A word character: `[a-zA-Z_0-9]` |
| `\W` | A non-word character: `[^\w]` |

In the table above, each construct in the left-hand column is shorthand for the character class in the right-hand column. For example, `\d` means a range of digits (0-9), and `\w` means a word character (any lowercase letter, any uppercase letter, the underscore character, or any digit). Use the predefined classes whenever possible. They make your code easier to read and eliminate errors introduced by malformed character classes.

Constructs beginning with a backslash are called *escaped constructs*. We previewed escaped constructs in the String Literals section where we mentioned the use of backslash and `\Q` and `\E` for quotation. If you are using an escaped construct within a string literal, you must precede the backslash with another backslash for the string to compile. For example:

```
private final String REGEX = "\\d"; // a single digit
```

In this example `\d` is the regular expression; the extra backslash is required for the code to compile. The test harness reads the expressions directly from the `Console`, however, so the extra backslash is unnecessary.

The following examples demonstrate the use of predefined character classes.

```
Enter your regex: .
Enter input string to search: @
I found the text "@" starting at index 0 and ending at index 1.

Enter your regex: .
Enter input string to search: 1
I found the text "1" starting at index 0 and ending at index 1.

Enter your regex: .
Enter input string to search: a
```

```
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \d
Enter input string to search: 1
I found the text "1" starting at index 0 and ending at index 1.

Enter your regex: \d
Enter input string to search: a
No match found.

Enter your regex: \D
Enter input string to search: 1
No match found.

Enter your regex: \D
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \s
Enter input string to search:
I found the text " " starting at index 0 and ending at index 1.

Enter your regex: \s
Enter input string to search: a
No match found.

Enter your regex: \S
Enter input string to search:
No match found.

Enter your regex: \S
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \w
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \w
Enter input string to search: !
No match found.

Enter your regex: \W
Enter input string to search: a
No match found.

Enter your regex: \W
Enter input string to search: !
I found the text "!" starting at index 0 and ending at index 1.
```

In the first three examples, the regular expression is simply `.` (the "dot" metacharacter) that indicates "any character." Therefore, the match is successful in all three cases (a randomly selected `@` character, a digit, and a letter). The remaining examples each use a single regular expression construct from the Predefined Character Classes table. You can refer to this table to figure out the logic behind each match:

- `\d` matches all digits
- `\s` matches spaces
- `\w` matches word characters

Alternatively, a capital letter means the opposite:

- `\D` matches non-digits
- `\S` matches non-spaces
- `\W` matches non-word characters

# Quantifiers

*Quantifiers* allow you to specify the number of occurrences to match against. For convenience, the three sections of the Pattern API specification describing greedy, reluctant, and possessive quantifiers are presented below. At first glance it may appear that the quantifiers `X?`, `X??` and `X?+` do exactly the same thing, since they all promise to match "`X`, once or not at all". There are subtle implementation differences which will be explained near the end of this section.

| Greedy | Reluctant | Possessive | Meaning |
|---|---|---|---|
| `X?` | `X??` | `X?+` | *X*, once or not at all |
| `X*` | `X*?` | `X*+` | *X*, zero or more times |
| `X+` | `X+?` | `X++` | *X*, one or more times |
| `X{n}` | `X{n}?` | `X{n}+` | *X*, exactly *n* times |
| `X{n,}` | `X{n,}?` | `X{n,}+` | *X*, at least *n* times |
| `X{n,m}` | `X{n,m}?` | `X{n,m}+` | *X*, at least *n* but not more than *m* times |

Let's start our look at greedy quantifiers by creating three different regular expressions: the letter "a" followed by either `?`, `*`, or `+`. Let's see what happens when these expressions are tested against an empty input string `""`:

```
Enter your regex: a?
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.

Enter your regex: a*
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.

Enter your regex: a+
Enter input string to search:
No match found.
```

## Zero-Length Matches

In the above example, the match is successful in the first two cases because the expressions `a?` and `a*` both allow for zero occurrences of the letter `a`. You'll also notice that the start and end indices are both zero, which is unlike any of the examples we've seen so far. The empty input string `""` has no length, so the test simply matches nothing at index 0. Matches of this sort are known as a *zero-length matches*. A zero-length match can occur in several cases: in an empty input string, at the beginning of an input string, after the last character of an input string, or in between any two characters of an input string. Zero-length matches are easily identifiable because they always start and end at the same index position.

Let's explore zero-length matches with a few more examples. Change the input string to a single letter

"a" and you'll notice something interesting:

```
Enter your regex: a?
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.

Enter your regex: a*
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.

Enter your regex: a+
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.
```

All three quantifiers found the letter "a", but the first two also found a zero-length match at index 1; that is, after the last character of the input string. Remember, the matcher sees the character "a" as sitting in the cell between index 0 and index 1, and our test harness loops until it can no longer find a match. Depending on the quantifier used, the presence of "nothing" at the index after the last character may or may not trigger a match.

Now change the input string to the letter "a" five times in a row and you'll get the following:

```
Enter your regex: a?
Enter input string to search: aaaaa
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 1 and ending at index 2.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "a" starting at index 3 and ending at index 4.
I found the text "a" starting at index 4 and ending at index 5.
I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a*
Enter input string to search: aaaaa
I found the text "aaaaa" starting at index 0 and ending at index 5.
I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a+
Enter input string to search: aaaaa
I found the text "aaaaa" starting at index 0 and ending at index 5.
```

The expression `a?` finds an individual match for each character, since it matches when "a" appears zero or one times. The expression `a*` finds two separate matches: all of the letter "a"'s in the first match, then the zero-length match after the last character at index 5. And finally, `a+` matches all occurrences of the letter "a", ignoring the presence of "nothing" at the last index.

At this point, you might be wondering what the results would be if the first two quantifiers encounter a letter other than "a". For example, what happens if it encounters the letter "b", as in "ababaaaab"?

Let's find out:

```
Enter your regex: a?
Enter input string to search: abababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "" starting at index 3 and ending at index 3.
I found the text "a" starting at index 4 and ending at index 5.
I found the text "a" starting at index 5 and ending at index 6.
I found the text "a" starting at index 6 and ending at index 7.
I found the text "a" starting at index 7 and ending at index 8.
I found the text "" starting at index 8 and ending at index 8.
I found the text "" starting at index 9 and ending at index 9.

Enter your regex: a*
Enter input string to search: abababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "" starting at index 3 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
I found the text "" starting at index 8 and ending at index 8.
I found the text "" starting at index 9 and ending at index 9.

Enter your regex: a+
Enter input string to search: abababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
```

Even though the letter "b" appears in cells 1, 3, and 8, the output reports a zero-length match at those locations. The regular expression `a?` is not specifically looking for the letter "b"; it's merely looking for the presence (or lack thereof) of the letter "a". If the quantifier allows for a match of "a" zero times, anything in the input string that's not an "a" will show up as a zero-length match. The remaining a's are matched according to the rules discussed in the previous examples.

To match a pattern exactly *n* number of times, simply specify the number inside a set of braces:

```
Enter your regex: a{3}
Enter input string to search: aa
No match found.

Enter your regex: a{3}
Enter input string to search: aaa
I found the text "aaa" starting at index 0 and ending at index 3.

Enter your regex: a{3}
Enter input string to search: aaaa
I found the text "aaa" starting at index 0 and ending at index 3.
```

Here, the regular expression `a{3}` is searching for three occurrences of the letter "a" in a row. The first test fails because the input string does not have enough a's to match against. The second test contains exactly 3 a's in the input string, which triggers a match. The third test also triggers a match because there are exactly 3 a's at the beginning of the input string. Anything following that is irrelevant to the first match. If the pattern should appear again after that point, it would trigger subsequent matches:

```
Enter your regex: a{3}
Enter input string to search: aaaaaaaaa
I found the text "aaa" starting at index 0 and ending at index 3.
I found the text "aaa" starting at index 3 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
```

To require a pattern to appear at least *n* times, add a comma after the number:

```
Enter your regex: a{3,}
Enter input string to search: aaaaaaaaa
I found the text "aaaaaaaaa" starting at index 0 and ending at index 9.
```

With the same input string, this test finds only one match, because the 9 a's in a row satisfy the need for "at least" 3 a's.

Finally, to specify an upper limit on the number of occurances, add a second number inside the braces:

```
Enter your regex: a{3,6} // find at least 3 (but no more than 6) a's in a row
Enter input string to search: aaaaaaaaa
I found the text "aaaaaa" starting at index 0 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
```

Here the first match is forced to stop at the upper limit of 6 characters. The second match includes whatever is left over, which happens to be three a's — the mimimum number of characters allowed for this match. If the input string were one character shorter, there would not be a second match since only two a's would remain.

## Capturing Groups and Character Classes with Quantifiers

Until now, we've only tested quantifiers on input strings containing one character. In fact, quantifiers can only attach to one character at a time, so the regular expression "abc+" would mean "a, followed by b, followed by c one or more times". It would not mean "abc" one or more times. However, quantifiers can also attach to Character Classes and Capturing Groups, such as `[abc]+` (a or b or c, one or more times) or `(abc)+` (the group "abc", one or more times).

Let's illustrate by specifying the group `(dog)`, three times in a row.

```
Enter your regex: (dog){3}
Enter input string to search: dogdogdogdogdogdog
I found the text "dogdogdog" starting at index 0 and ending at index 9.
I found the text "dogdogdog" starting at index 9 and ending at index 18.

Enter your regex: dog{3}
Enter input string to search: dogdogdogdogdogdog
No match found.
```

Here the first example finds three matches, since the quantifier applies to the entire capturing group. Remove the parentheses, however, and the match fails because the quantifier `{3}` now applies only to the letter "g".

Similarly, we can apply a quantifier to an entire character class:

```
Enter your regex: [abc]{3}
Enter input string to search: abccabaaaccbbbc
I found the text "abc" starting at index 0 and ending at index 3.
I found the text "cab" starting at index 3 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
I found the text "ccb" starting at index 9 and ending at index 12.
I found the text "bbc" starting at index 12 and ending at index 15.

Enter your regex: abc{3}
Enter input string to search: abccabaaaccbbbc
No match found.
```

Here the quantifier `{3}` applies to the entire character class in the first example, but only to the letter "c" in the second.

## Differences Among Greedy, Reluctant, and Possessive Quantifiers

There are subtle differences among greedy, reluctant, and possessive quantifiers.

Greedy quantifiers are considered "greedy" because they force the matcher to read in, or *eat*, the entire input string prior to attempting the first match. If the first match attempt (the entire input string) fails, the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left to back off from. Depending on the quantifier used in the expression, the last thing it will try matching against is 1 or 0 characters.

The reluctant quantifiers, however, take the opposite approach: They start at the beginning of the input string, then reluctantly eat one character at a time looking for a match. The last thing they try is the entire input string.

Finally, the possessive quantifiers always eat the entire input string, trying once (and only once) for a match. Unlike the greedy quantifiers, possessive quantifiers never back off, even if doing so would allow the overall match to succeed.

To illustrate, consider the input string `xfooxxxxxxfoo`.

```
Enter your regex: .*foo  // greedy quantifier
Enter input string to search: xfooxxxxxxfoo
I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Enter your regex: .*?foo  // reluctant quantifier
Enter input string to search: xfooxxxxxxfoo
I found the text "xfoo" starting at index 0 and ending at index 4.
I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

Enter your regex: .*+foo // possessive quantifier
```

```
Enter input string to search: xfooxxxxxfoo
No match found.
```

The first example uses the greedy quantifier `.*` to find "anything", zero or more times, followed by the letters `"f" "o" "o"`. Because the quantifier is greedy, the `.*` portion of the expression first eats the entire input string. At this point, the overall expression cannot succeed, because the last three letters (`"f" "o" "o"`) have already been consumed. So the matcher slowly backs off one letter at a time until the rightmost occurrence of "foo" has been regurgitated, at which point the match succeeds and the search ends.

The second example, however, is reluctant, so it starts by first consuming "nothing". Because "foo" doesn't appear at the beginning of the string, it's forced to swallow the first letter (an "x"), which triggers the first match at 0 and 4. Our test harness continues the process until the input string is exhausted. It finds another match at 4 and 13.

The third example fails to find a match because the quantifier is possessive. In this case, the entire input string is consumed by `.*+`, leaving nothing left over to satisfy the "foo" at the end of the expression. Use a possessive quantifier for situations where you want to seize all of something without ever backing off; it will outperform the equivalent greedy quantifier in cases where the match is not immediately found.

# Capturing Groups

In the previous section, we saw how quantifiers attach to one character, character class, or capturing group at a time. But until now, we have not discussed the notion of capturing groups in any detail.

*Capturing groups* are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression `(dog)` creates a single group containing the letters `"d" "o"` and `"g"`. The portion of the input string that matches the capturing group will be saved in memory for later recall via backreferences (as discussed below in the section, Backreferences).

## Numbering

As described in the `Pattern` API, capturing groups are numbered by counting their opening parentheses from left to right. In the expression `((A)(B(C)))`, for example, there are four such groups:

1. `((A)(B(C)))`
2. `(A)`
3. `(B(C))`
4. `(C)`

To find out how many groups are present in the expression, call the `groupCount` method on a matcher object. The `groupCount` method returns an `int` showing the number of capturing groups present in the matcher's pattern. In this example, `groupCount` would return the number `4`, showing that the pattern contains 4 capturing groups.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by `groupCount`. Groups beginning with `(?` are pure, *non-capturing groups* that do not capture text and do not count towards the group total. (You'll see examples of non-capturing groups later in the section Methods of the Pattern Class.)

It's important to understand how groups are numbered because some `Matcher` methods accept an `int` specifying a particular group number as a parameter:

- `public int start(int group)`: Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end (int group)`: Returns the index of the last character, plus one, of the subsequence captured by the given group during the previous match operation.
- `public String group (int group)`: Returns the input subsequence captured by the given group during the previous match operation.

## Backreferences

The section of the input string matching the capturing group(s) is saved in memory for later recall via *backreference*. A backreference is specified in the regular expression as a backslash (`\`) followed by

a digit indicating the number of the group to be recalled. For example, the expression `(\d\d)` defines one capturing group matching two digits in a row, which can be recalled later in the expression via the backreference `\1`.

To match any 2 digits, followed by the exact same two digits, you would use `(\d\d)\1` as the regular expression:

```
Enter your regex: (\d\d)\1
Enter input string to search: 1212
I found the text "1212" starting at index 0 and ending at index 4.
```

If you change the last two digits the match will fail:

```
Enter your regex: (\d\d)\1
Enter input string to search: 1234
No match found.
```

For nested capturing groups, backreferencing works in exactly the same way: Specify a backslash followed by the number of the group to be recalled.

# Boundary Matchers

Until now, we've only been interested in whether or not a match is found *at some location* within a particular input string. We never cared about *where* in the string the match was taking place.

You can make your pattern matches more precise by specifying such information with *boundary matchers*. For example, maybe you're interested in finding a particular word, but only if it appears at the beginning or end of a line. Or maybe you want to know if the match is taking place on a word boundary, or at the end of the previous match.

The following table lists and explains all the boundary matchers.

| Boundary Construct | Description |
|---|---|
| `^` | The beginning of a line |
| `$` | The end of a line |
| `\b` | A word boundary |
| `\B` | A non-word boundary |
| `\A` | The beginning of the input |
| `\G` | The end of the previous match |
| `\Z` | The end of the input but for the final terminator, if any |
| `\z` | The end of the input |

The following examples demonstrate the use of boundary matchers `^` and `$`. As noted above, `^` matches the beginning of a line, and `$` matches the end.

```
Enter your regex: ^dog$
Enter input string to search: dog
I found the text "dog" starting at index 0 and ending at index 3.

Enter your regex: ^dog$
Enter input string to search:       dog
No match found.

Enter your regex: \s*dog$
Enter input string to search:             dog
I found the text "            dog" starting at index 0 and ending at index 15.

Enter your regex: ^dog\w*
Enter input string to search: dogblahblah
I found the text "dogblahblah" starting at index 0 and ending at index 11.
```

The first example is successful because the pattern occupies the entire input string. The second example fails because the input string contains extra whitespace at the beginning. The third example specifies an expression that allows for unlimited white space, followed by "dog" on the end of the line. The fourth example requires "dog" to be present at the beginning of a line followed by an

unlimited number of word characters.

To check if a pattern begins and ends on a word boundary (as opposed to a substring within a longer string), just use `\b` on either side; for example, `\bdog\b`

```
Enter your regex: \bdog\b
Enter input string to search: The dog plays in the yard.
I found the text "dog" starting at index 4 and ending at index 7.

Enter your regex: \bdog\b
Enter input string to search: The doggie plays in the yard.
No match found.
```

To match the expression on a non-word boundary, use `\B` instead:

```
Enter your regex: \bdog\B
Enter input string to search: The dog plays in the yard.
No match found.

Enter your regex: \bdog\B
Enter input string to search: The doggie plays in the yard.
I found the text "dog" starting at index 4 and ending at index 7.
```

To require the match to occur only at the end of the previous match, use `\G`:

```
Enter your regex: dog
Enter input string to search: dog dog
I found the text "dog" starting at index 0 and ending at index 3.
I found the text "dog" starting at index 4 and ending at index 7.

Enter your regex: \Gdog
Enter input string to search: dog dog
I found the text "dog" starting at index 0 and ending at index 3.
```

Here the second example finds only one match, because the second occurrence of "dog" does not start at the end of the previous match.

# Methods of the Pattern Class

Until now, we've only used the test harness to create `Pattern` objects in their most basic form. This section explores advanced techniques such as creating patterns with flags and using embedded flag expressions. It also explores some additional useful methods that we haven't yet discussed.

## Creating a Pattern with Flags

The `Pattern` class defines an alternate `compile` method that accepts a set of flags affecting the way the pattern is matched. The flags parameter is a bit mask that may include any of the following public static fields:

- `Pattern.CANON_EQ` Enables canonical equivalence. When this flag is specified, two characters will be considered to match if, and only if, their full canonical decompositions match. The expression `"a\u030A"`, for example, will match the string `"\u00E5"` when this flag is specified. By default, matching does not take canonical equivalence into account. Specifying this flag may impose a performance penalty.
- `Pattern.CASE_INSENSITIVE` Enables case-insensitive matching. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the UNICODE_CASE flag in conjunction with this flag. Case-insensitive matching can also be enabled via the embedded flag expression `(?i)`. Specifying this flag may impose a slight performance penalty.
- `Pattern.COMMENTS` Permits whitespace and comments in the pattern. In this mode, whitespace is ignored, and embedded comments starting with `#` are ignored until the end of a line. Comments mode can also be enabled via the embedded flag expression `(?x)`.
- `Pattern.DOTALL` Enables dotall mode. In dotall mode, the expression `.` matches any character, including a line terminator. By default this expression does not match line terminators. Dotall mode can also be enabled via the embedded flag expression `(?s)`. (The s is a mnemonic for "single-line" mode, which is what this is called in Perl.)
- `Pattern.LITERAL` Enables literal parsing of the pattern. When this flag is specified then the input string that specifies the pattern is treated as a sequence of literal characters. Metacharacters or escape sequences in the input sequence will be given no special meaning. The flags `CASE_INSENSITIVE` and `UNICODE_CASE` retain their impact on matching when used in conjunction with this flag. The other flags become superfluous. There is no embedded flag character for enabling literal parsing.
- `Pattern.MULTILINE` Enables multiline mode. In multiline mode the expressions `^` and `$` match just after or just before, respectively, a line terminator or the end of the input sequence. By default these expressions only match at the beginning and the end of the entire input sequence. Multiline mode can also be enabled via the embedded flag expression `(?m)`.
- `Pattern.UNICODE_CASE` Enables Unicode-aware case folding. When this flag is specified then case-insensitive matching, when enabled by the `CASE_INSENSITIVE` flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case folding can also be enabled via the embedded flag expression `(?u)`. Specifying this flag may impose a performance penalty.

- `Pattern.UNIX_LINES` Enables UNIX lines mode. In this mode, only the `'\n'` line terminator is recognized in the behavior of `.`, `^`, and `$`. UNIX lines mode can also be enabled via the embedded flag expression `(?d)`.

In the following steps we will modify the test harness, `RegexTestHarness.java` to create a pattern with case-insensitive matching.

First, modify the code to invoke the alternate version of `compile`:

```
Pattern pattern =
Pattern.compile(console.readLine("%nEnter your regex: "),
Pattern.CASE_INSENSITIVE);
```

Then compile and run the test harness to get the following results:

```

Enter your regex: dog
Enter input string to search: DoGDOg
I found the text "DoG" starting at index 0 and ending at index 3.
I found the text "DOg" starting at index 3 and ending at index 6.
```

As you can see, the string literal "dog" matches both occurences, regardless of case. To compile a pattern with multiple flags, separate the flags to be included using the bitwise OR operator "`|`". For clarity, the following code samples hardcode the regular expression instead of reading it from the `Console`:

```

pattern = Pattern.compile("[az]$", Pattern.MULTILINE | Pattern.UNIX_LINES);
```

You could also specify an `int` variable instead:

```

final int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;
Pattern pattern = Pattern.compile("aa", flags);
```

## Embedded Flag Expressions

It's also possible to enable various flags using *embedded flag expressions*. Embedded flag expressions are an alternative to the two-argument version of `compile`, and are specified in the regular expression itself. The following example uses the original test harness, `RegexTestHarness.java` with the embedded flag expression `(?i)` to enable case-insensitive matching.

```

Enter your regex: (?i)foo
Enter input string to search: FOOfooFoOfoO
I found the text "FOO" starting at index 0 and ending at index 3.
I found the text "foo" starting at index 3 and ending at index 6.
```

```
I found the text "FoO" starting at index 6 and ending at index 9.
I found the text "foO" starting at index 9 and ending at index 12.
```

Once again, all matches succeed regardless of case.

The embedded flag expressions that correspond to `Pattern`'s publicly accessible fields are presented in the following table:

| Constant | Equivalent Embedded Flag Expression |
|---|---|
| `Pattern.CANON_EQ` | None |
| `Pattern.CASE_INSENSITIVE` | `(?i)` |
| `Pattern.COMMENTS` | `(?x)` |
| `Pattern.MULTILINE` | `(?m)` |
| `Pattern.DOTALL` | `(?s)` |
| `Pattern.LITERAL` | None |
| `Pattern.UNICODE_CASE` | `(?u)` |
| `Pattern.UNIX_LINES` | `(?d)` |

## Using the `matches(String,CharSequence)` Method

The `Pattern` class defines a convenient `matches` method that allows you to quickly check if a pattern is present in a given input string. As with all public static methods, you should invoke `matches` by its class name, such as `Pattern.matches("\\d","1");`. In this example, the method returns `true`, because the digit "1" matches the regular expression `\d`.

## Using the `split(String)` Method

The `split` method is a great tool for gathering the text that lies on either side of the pattern that's been matched. As shown below in `SplitDemo.java`, the `split` method could extract the words "`one two three four five`" from the string "`one:two:three:four:five`":

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitDemo {

    private static final String REGEX = ":";
    private static final String INPUT =
        "one:two:three:four:five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

```
OUTPUT:

one
two
three
four
five
```

For simplicity, we've matched a string literal, the colon (`:`) instead of a complex regular expression. Since we're still using `Pattern` and `Matcher` objects, you can use split to get the text that falls on either side of any regular expression. Here's the same example, `SplitDemo2.java`, modified to split on digits instead:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitDemo2 {

    private static final String REGEX = "\\d";
    private static final String INPUT =
        "one9two4three7four1five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

```
OUTPUT:

one
two
three
four
five
```

## Other Utility Methods

You may find the following methods to be of some use as well:

- `public static String quote(String s)` Returns a literal pattern `String` for the specified `String`. This method produces a `String` that can be used to create a `Pattern` that would match `String s` as if it were a literal pattern. Metacharacters or escape sequences in the input sequence will be given no special meaning.
- `public String toString()` Returns the `String` representation of this pattern. This is the regular expression from which this pattern was compiled.

## Pattern Method Equivalents in `java.lang.String`

Regular expression support also exists in `java.lang.String` through several methods that mimic the

behavior of `java.util.regex.Pattern`. For convenience, key excerpts from their API are presented below.

- `public boolean matches(String regex)`: Tells whether or not this string matches the given regular expression. An invocation of this method of the form *`str`*`.matches(`*`regex`*`)` yields exactly the same result as the expression `Pattern.matches(`*`regex`*`, `*`str`*`)`.
- `public String[] split(String regex, int limit)`: Splits this string around matches of the given regular expression. An invocation of this method of the form *`str`*`.split(`*`regex`*`, `*`n`*`)` yields the same result as the expression `Pattern.compile(`*`regex`*`).split(`*`str`*`, `*`n`*`)`
- `public String[] split(String regex)`: Splits this string around matches of the given regular expression. This method works the same as if you invoked the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are not included in the resulting array.

There is also a replace method, that replaces one `CharSequence` with another:

- `public String replace(CharSequence target,CharSequence replacement)`: Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

# Methods of the Matcher Class

This section describes some additional useful methods of the `Matcher` class. For convenience, the methods listed below are grouped according to functionality.

## Index Methods

*Index methods* provide useful index values that show precisely where the match was found in the input string:

- `public int start()`: Returns the start index of the previous match.
- `public int start(int group)`: Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end()`: Returns the offset after the last character matched.
- `public int end(int group)`: Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

## Study Methods

*Study methods* review the input string and return a boolean indicating whether or not the pattern is found.

- `public boolean lookingAt()`: Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
- `public boolean find()`: Attempts to find the next subsequence of the input sequence that matches the pattern.
- `public boolean find(int start)`: Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
- `public boolean matches()`: Attempts to match the entire region against the pattern.

## Replacement Methods

*Replacement methods* are useful methods for replacing text in an input string.

- `public Matcher appendReplacement(StringBuffer sb, String replacement)`: Implements a non-terminal append-and-replace step.
- `public StringBuffer appendTail(StringBuffer sb)`: Implements a terminal append-and-replace step.
- `public String replaceAll(String replacement)`: Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
- `public String replaceFirst(String replacement)`: Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
- `public static String quoteReplacement(String s)`: Returns a literal replacement `String` for the specified `String`. This method produces a `String` that will work as a literal replacement `s` in the `appendReplacement` method of the `Matcher` class. The `String` produced will match the sequence of characters in `s` treated as a literal sequence. Slashes (`'\'`) and dollar

signs (`'$'`) will be given no special meaning.

## Using the `start` and `end` Methods

Here's an example, `MatcherDemo.java`, that counts the number of times the word "dog" appears in the input string.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MatcherDemo {

    private static final String REGEX =
        "\\bdog\\b";
    private static final String INPUT =
        "dog dog dog doggie dogg";

    public static void main(String[] args) {
       Pattern p = Pattern.compile(REGEX);
       //  get a matcher object
       Matcher m = p.matcher(INPUT);
       int count = 0;
       while(m.find()) {
           count++;
           System.out.println("Match number "
                              + count);
           System.out.println("start(): "
                              + m.start());
           System.out.println("end(): "
                              + m.end());
      }
   }
}
```

```
OUTPUT:

Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
```

You can see that this example uses word boundaries to ensure that the letters `"d" "o" "g"` are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred. The `start` method returns the start index of the subsequence captured by the given group during the previous match operation, and `end` returns the index of the last character matched, plus one.

## Using the `matches` and `lookingAt` Methods

The `matches` and `lookingAt` methods both attempt to match an input sequence against a pattern. The difference, however, is that `matches` requires the entire input sequence to be matched, while

`lookingAt` does not. Both methods always start at the beginning of the input string. Here's the full code, `MatchesLooking.java`:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT =
        "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] args) {

        // Initialize
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "
                           + REGEX);
        System.out.println("Current INPUT is: "
                           + INPUT);

        System.out.println("lookingAt(): "
            + matcher.lookingAt());
        System.out.println("matches(): "
            + matcher.matches());
    }
}
```

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

## Using `replaceFirst(String)` and `replaceAll(String)`

The `replaceFirst` and `replaceAll` methods replace text that matches a given regular expression. As their names indicate, `replaceFirst` replaces the first occurrence, and `replaceAll` replaces all occurences. Here's the `ReplaceDemo.java` code:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class ReplaceDemo {

    private static String REGEX = "dog";
    private static String INPUT =
        "The dog says meow. All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
```

```
        System.out.println(INPUT);
    }
}
```

```
OUTPUT: The cat says meow. All cats say meow.
```

In this first version, all occurrences of `dog` are replaced with `cat`. But why stop here? Rather than replace a simple literal like `dog`, you can replace text that matches *any* regular expression. The API for this method states that "given the regular expression `a*b`, the input `aabfooaabfooabfoob`, and the replacement string `-`, an invocation of this method on a matcher for that expression would yield the string `-foo-foo-foo-`."

Here's the `ReplaceDemo2.java` code:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class ReplaceDemo2 {

    private static String REGEX = "a*b";
    private static String INPUT =
        "aabfooaabfooabfoob";
    private static String REPLACE = "-";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}
```

```
OUTPUT: -foo-foo-foo-
```

To replace only the first occurrence of the pattern, simply call `replaceFirst` instead of `replaceAll`. It accepts the same parameter.

## Using `appendReplacement(StringBuffer,String)` and `appendTail(StringBuffer)`

The `Matcher` class also provides `appendReplacement` and `appendTail` methods for text replacement. The following example, `RegexDemo.java`, uses these two methods to achieve the same effect as `replaceAll`.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexDemo {

    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
```

```
    private static String REPLACE = "-";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT); // get a matcher object
        StringBuffer sb = new StringBuffer();
        while(m.find()){
            m.appendReplacement(sb,REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

```
OUTPUT: -foo-foo-foo-
```

## Matcher Method Equivalents in `java.lang.String`

For convenience, the `String` class mimics a couple of `Matcher` methods as well:

- `public String replaceFirst(String regex, String replacement)`: Replaces the first substring of this string that matches the given regular expression with the given replacement. An invocation of this method of the form *`str`*`.replaceFirst(`*`regex, repl`*`)` yields exactly the same result as the expression `Pattern.compile(`*`regex`*`).matcher(`*`str`*`).replaceFirst(`*`repl`*`)`
- `public String replaceAll(String regex, String replacement)`: Replaces each substring of this string that matches the given regular expression with the given replacement. An invocation of this method of the form *`str`*`.replaceAll(`*`regex, repl`*`)` yields exactly the same result as the expression `Pattern.compile(`*`regex`*`).matcher(`*`str`*`).replaceAll(`*`repl`*`)`

# Methods of the PatternSyntaxException Class

A `PatternSyntaxException` is an unchecked exception that indicates a syntax error in a regular expression pattern. The `PatternSyntaxException` class provides the following methods to help you determine what went wrong:

- `public String getDescription()`: Retrieves the description of the error.
- `public int getIndex()`: Retrieves the error index.
- `public String getPattern()`: Retrieves the erroneous regular expression pattern.
- `public String getMessage()`: Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular-expression pattern, and a visual indication of the error index within the pattern.

The following source code, `RegexTestHarness2.java`, updates our test harness to check for malformed regular expressions:

```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.regex.PatternSyntaxException;

public class RegexTestHarness2 {

    public static void main(String[] args){
        Pattern pattern = null;
        Matcher matcher = null;

        Console console = System.console();
        if (console == null) {
            System.err.println("No console.");
            System.exit(1);
        }
        while (true) {
            try{
                pattern =
                Pattern.compile(console.readLine("%nEnter your regex: "));

                matcher =
                pattern.matcher(console.readLine("Enter input string to search: "));
            }
            catch(PatternSyntaxException pse){
                console.format("There is a problem" +
                               " with the regular expression!%n");
                console.format("The pattern in question is: %s%n",
                               pse.getPattern());
                console.format("The description is: %s%n",
                               pse.getDescription());
                console.format("The message is: %s%n",
                               pse.getMessage());
                console.format("The index is: %s%n",
                               pse.getIndex());
                System.exit(0);
            }
            boolean found = false;
            while (matcher.find()) {
                console.format("I found the text" +
                    " \"%s\" starting at " +
                    "index %d and ending at index %d.%n",
                    matcher.group(),
```

```
                    matcher.start(),
                    matcher.end());
                found = true;
            }
            if(!found){
                console.format("No match found.%n");
            }
        }
    }
}
```

To run this test, enter `?i)foo` as the regular expression. This mistake is a common scenario in which the programmer has forgotten the opening parenthesis in the embedded flag expression `(?i)`. Doing so will produce the following results:

```
Enter your regex: ?i)
There is a problem with the regular expression!
The pattern in question is: ?i)
The description is: Dangling meta character '?'
The message is: Dangling meta character '?' near index 0
?i)
^
The index is: 0
```

From this output, we can see that the syntax error is a dangling metacharacter (the question mark) at index 0. A missing opening parenthesis is the culprit.

# Unicode Support

As of the JDK 7 release, Regular Expression pattern matching has expanded functionality to support Unicode 6.0.



## Matching a Specific Code Point

You can match a specific Unicode code point using an escape sequence of the form `\uFFFF`, where `FFFF` is the hexidecimal value of the code point you want to match. For example, `\u6771` matches the Han character for east.

Alternatively, you can specify a code point using Perl-style hex notation, `\x{...}`. For example:

```
String hexPattern = "\x{" + Integer.toHexString(codePoint) + "}";
```

## Unicode Character Properties

Each Unicode character, in addition to its value, has certain attributes, or properties. You can match a single character belonging to a particular category with the expression `\p{prop}`. You can match a single character *not* belonging to a particular category with the expression `\P{prop}`.

The three supported property types are scripts, blocks, and a "general" category.

### Scripts

To determine if a code point belongs to a specific script, you can either use the `script` keyword, or the `sc` short form, for example, `\p{script=Hiragana}`. Alternatively, you can prefix the script name with the string `Is`, such as `\p{IsHiragana}`.

Valid script names supported by `Pattern` are those accepted by `UnicodeScript.forName`.

### Blocks

A block can be specified using the `block` keyword, or the `blk` short form, for example, `\p{block=Mongolian}`. Alternatively, you can prefix the block name with the string `In`, such as `\p{InMongolian}`.

Valid block names supported by `Pattern` are those accepted by `UnicodeBlock.forName`.

### General Category

Categories can be specified with optional prefix `Is`. For example, `IsL` matches the category of Unicode letters. Categories can also be specified by using the `general_category` keyword, or the short form `gc`. For example, an uppercase letter can be matched using `general_category=Lu` or

`gc=Lu`.

Supported categories are those of The Unicode Standard in the version specified by the `Character` class.

# Additional Resources

Now that you've completed this lesson on regular expressions, you'll probably find that your main references will be the API documentation for the following classes: `Pattern`, `Matcher`, and `PatternSyntaxException`.

For a more precise description of the behavior of regular expression constructs, we recommend reading the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

# Questions and Exercises: Regular Expressions

## Questions

**1.** What are the three public classes in the `java.util.regex` package? Describe the purpose of each.
**2.** Consider the string literal `"foo"`. What is the start index? What is the end index? Explain what these numbers mean.
**3.** What is the difference between an ordinary character and a metacharacter? Give an example of each.
**4.** How do you force a metacharacter to act like an ordinary character?
**5.** What do you call a set of characters enclosed in square brackets? What is it for?
**6.** Here are three predefined character classes: `\d`, `\s`, and `\w`. Describe each one, and rewrite it using square brackets.
**7.** For each of `\d`, `\s`, and `\w`, write *two* simple expressions that match the *opposite* set of characters.
**8.** Consider the regular expression `(dog){3}`. Identify the two subexpressions. What string does the expression match?

## Exercises

**1.** Use a backreference to write an expression that will match a person's name only if that person's first name and last name are the same.

Check your answers.

## Questions

**1.** Question: What are the three public classes in the `java.util.regex` package? Describe the purpose of each. Answer:

- `Pattern` instances are compiled representations of regular expressions.
- `Matcher` instances are engines that interpret patterns and perform match operations against input strings.
- `PatternSyntaxException` defines an unchecked exception indicating a syntax error in a regular expression.

**2.** Question: Consider the string literal `"foo"`. What is the start index? What is the end index? Explain what these numbers mean. Answer: Each character in the string resides in its own cell. Index positions point between cells. The string `"foo"` starts at index 0 and ends at index 3, even though the characters only occupy cells 0, 1, and 2.

**3.** Question: What is the difference between an ordinary character and a metacharacter? Give an example of each. Answer: An ordinary character in a regular expression matches itself. A metacharacter is a special character that affects the way a pattern is matched. The letter `A` is an ordinary character. The punctuation mark `.` is a metacharacter that matches any single character.

**4.** Question: How do you force a metacharacter to act like an ordinary character? Answer: There are two ways:

- Precede the metacharacter with a backslash (`\`);
- Enclose the metacharacter within the quote expressions, `\Q` (at the beginning) and `\E` (at the end).

**5.** Question: What do you call a set of characters enclosed in square brackets? What is it for? Answer: This is a character class. It matches any single character that is in the class of characters specified by the expression between the brackets.

**6.** Question: Here are three predefined character classes: `\d`, `\s`, and `\w`. Describe each one, and rewrite it using square brackets. Answer:

| | |
|---|---|
| `\d` Matches any digit. | `[0-9]` |
| `\s` Matches any white space character. | `[ \t\n-x0B\f\r]` |
| `\w` Matches any word character. | `[a-zA-Z_0-9]` |

**7.** Question: For each of `\d`, `\s`, and `\w`, write *two* simple expressions that match the *opposite* set of characters. Answer:

```
\d \D [^\d]
\s \S [^\s]
\w \W [^\w]
```

**8.** Question: Consider the regular expression `(dog){3}`. Identify the two subexpressions. What string does the expression match? Answer: The expression consists of a capturing group, `(dog)`, followed

by a greedy quantifier `{3}`. It matches the string "dogdogdog".

## Exercises

**a.** Exercise: Use a backreference to write an expression that will match a person's name only if that person's first name and last name are the same. Solution: `([A-Z][a-zA-Z]*)\s\1`

# Essential Classes: End of Trail

You have reached the end of the "Essential Classes" trail.

If you have comments or suggestions about this trail, use our feedback page to tell us about it.

 Creating a GUI With JFC/Swing: Once you know how to create applications or applets, follow this trail to learn how to create their user interfaces.

 Collections: Using the classes and interfaces in the collections framework you can group objects together into a single object.

 Internationalization: Essential if you want to create a program that can be used by people all over the world. Furthermore, you can use the information about formatting dates, numbers, and strings in any program.