Basic I/O in Java

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.

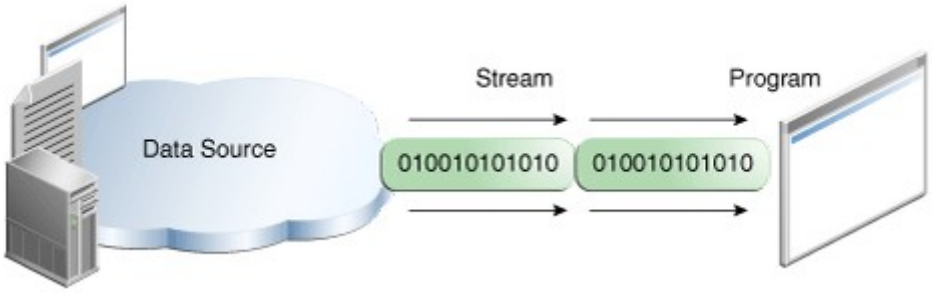
Security consideration: Some I/O operations are subject to approval by the current security manager. The example programs contained in these lessons are standalone applications, which by default have no security manager. To work in an applet, most of these examples would have to be modified. See What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

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.

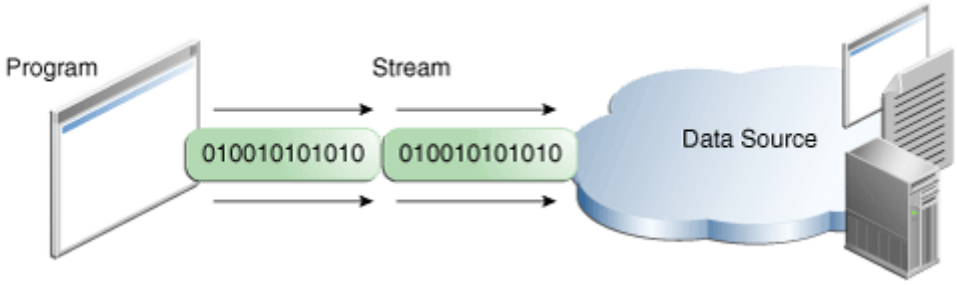
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

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`.

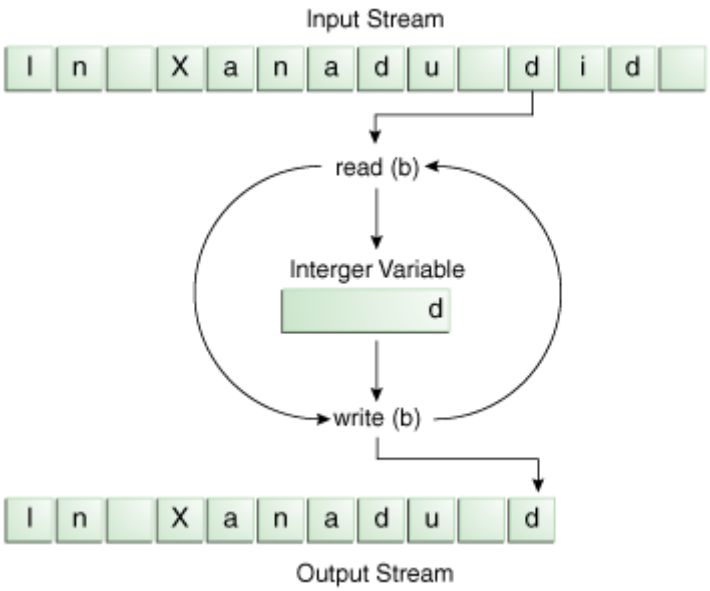
There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, `FileInputStream` and `FileOutputStream`. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

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.

Notice that `read()` returns an `int` value. If the input is a stream of bytes, why doesn't `read()` return a `byte` value? Using a `int` as a return type allows `read()` to use -1 to indicate that it has reached the end of the stream.

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

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`.

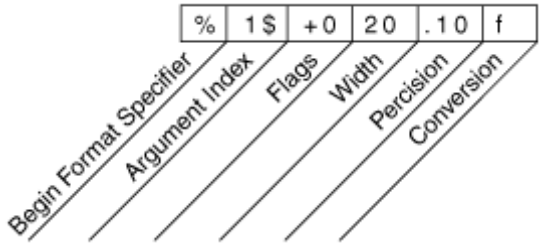
In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, `Format`, that uses every possible kind of element.

```
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:
    1. Invoke `Console.readPassword` twice to prompt for and read a new password.

    2. If the user entered the same password both times, invoke `change` to change it. (Again, `change` is a dummy method.)
    3. 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	Sample Value
1	`double`	Item price	`DataOutputStream.writeDouble`	`DataInputStream.readDouble`	`19.99`
2	`int`	Unit count	`DataOutputStream.writeInt`	`DataInputStream.readInt`	`12`
3	`String`	Item description	`DataOutputStream.writeUTF`	`DataInputStream.readUTF`	`"Java T-Shirt"`

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`.

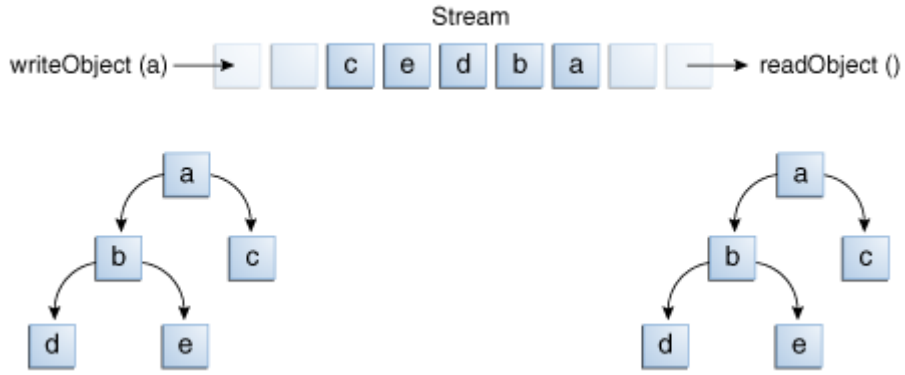
The object stream classes are `ObjectInputStream` and `ObjectOutputStream`. These classes implement `ObjectInput` and `ObjectOutput`, which are subinterfaces of `DataInput` and `DataOutput`. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. 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.



I/O of multiple referred-to objects

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.

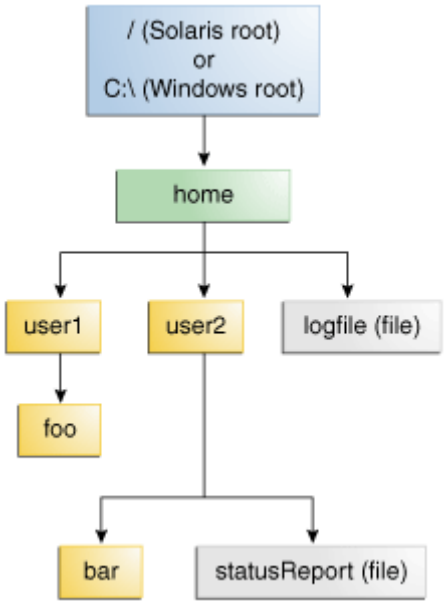
What Is a Path? (And Other File System Facts)

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.

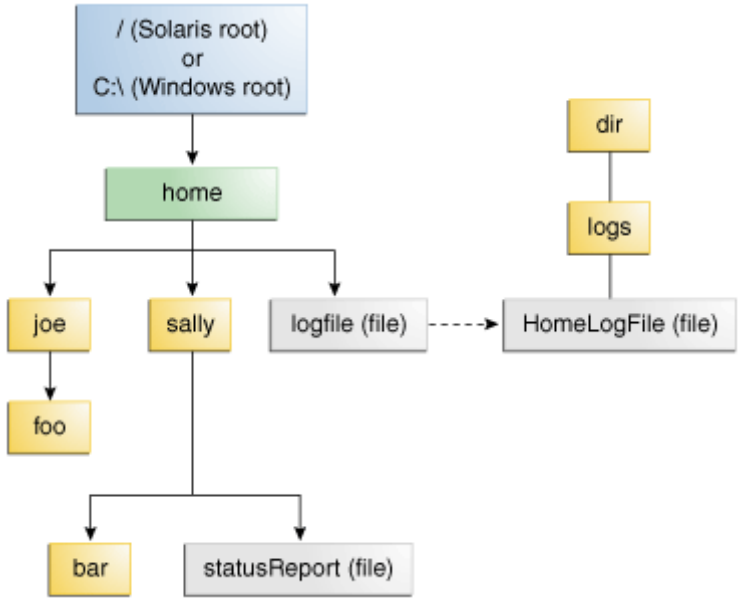
A relative path needs to be combined with another path in order to access a file. For example, `joe/foo` is a relative path. Without more information, a program cannot reliably locate the `joe/foo` directory in the file system.

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");
```

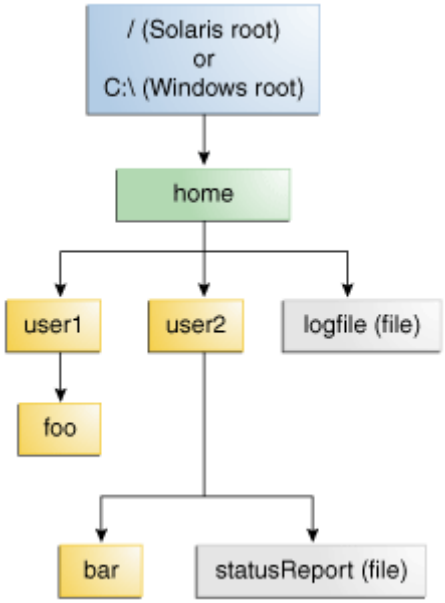
The following example creates `/u/joe/logs/foo.log` assuming your home directory is `/u/joe`, or `C:\joe\logs\foo.log` if you are on Windows.

```
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
`toString`	`/home/joe/foo`	`C:\home\joe\foo`	Returns the string representation of the `Path`. If the path was created using `Filesystems.getDefault().getPath(String)` or `Paths.get` (the latter is a convenience method 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 including a root element) as specified by the beginning and 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(true);
} 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, String, LinkOption...)`	Reads a file's attributes as a bulk operation. The `String` parameter 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

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 = file
    .getFileAttributeView(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.

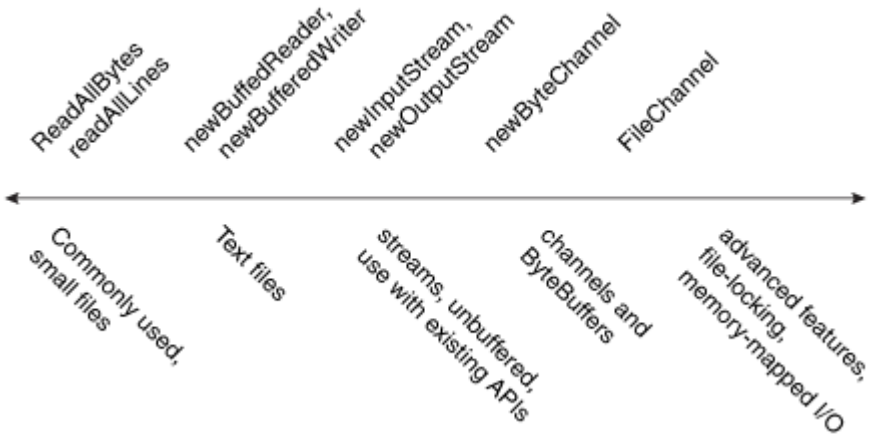
The following code snippet prints the space usage for the file store where a particular file resides:

```
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.*;

Path logfile = ...;

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

try (OutputStream out = new BufferedOutputStream(
                  logfile.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

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:

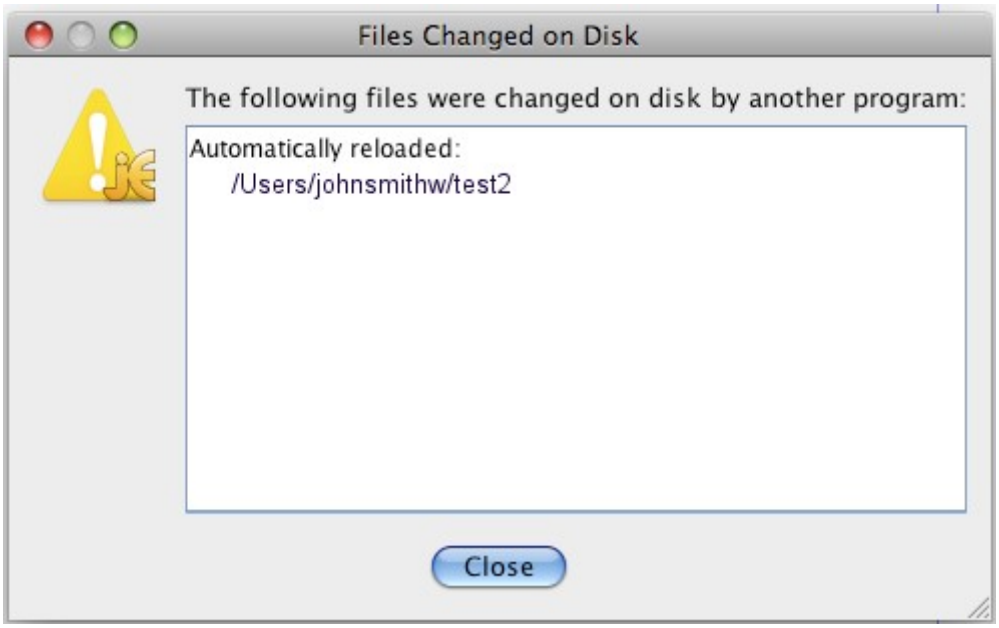
```
/**
 * Sample code that finds files that match the specified glob pattern.
 * For more information on what constitutes a glob pattern, see
 * http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob
 *
 * The file or directories that match the pattern are printed to
 * standard out.  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 don't 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
`File.createTempFile`	`Files.createTempFile(Path, String, FileAttributes<?>)`, `Files.createTempFile(Path, String, String, FileAttributes<?>)`	Creating Files Creating and Writing a File by Using Stream I/O 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`) 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