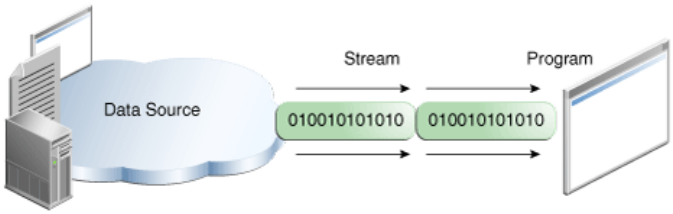
Basic I/O

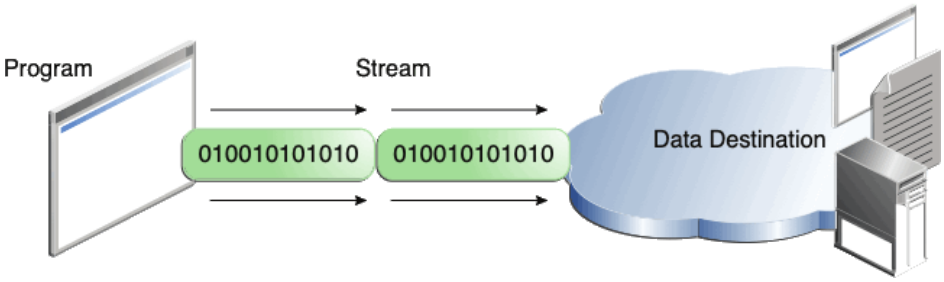
I/O Streams

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:



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



Coming up, 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.
```

I/O Streams

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.

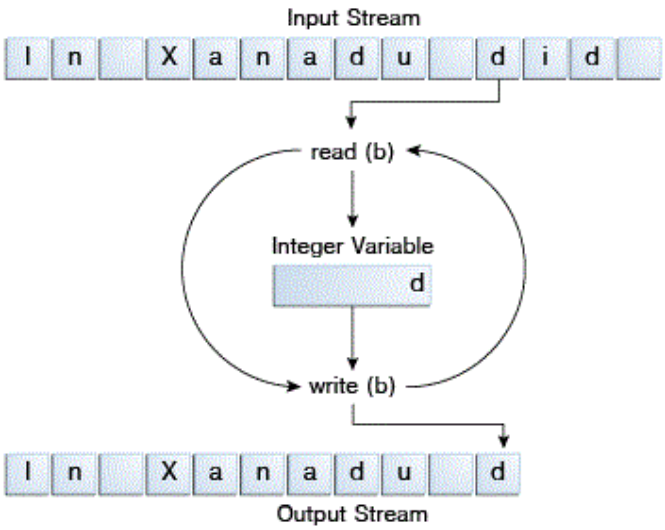
There's many byte stream classes.

How byte streams work: file I/O byte streams; FileInputStream & FileOutputStream. Other kinds of byte streams are used in similar ways; construction differs.

Using Byte Streams

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.

```
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();
            }
        }
    }
}
```



I/O Streams

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.

I/O 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.

I/O Streams

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.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();
            }
        }
    }
}
```

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.

I/O Streams

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"));
```

4 buffered stream classes used to wrap unbuffered streams:

- BufferedInputStream & BufferedOutputStream create buffered byte streams, while
- BufferedReader & BufferedWriter create buffered character streams.

Flushing Buffered Streams

Writing out a buffer at critical points, without waiting for it to fill.

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.

I/O Streams

Scanning & Formatting

Programming I/O often involves translating *to and from* the neatly formatted data humans like to work with. The Java platform provides 2 APIs to assist.

1. The **scanner** API breaks input into individual tokens associated with bits of data.
2. The **formatting** API assembles data into nicely formatted, human-readable form.

Scanning

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

Breaking Input into Tokens

Default: Scanner uses white space to separate tokens.
(White space characters include blanks, tabs, & line terminators.)

isWhitespace

`public static boolean isWhitespace(char ch)`

Determines if the specified character is white space according to Java. A character is a Java whitespace character if and only if it satisfies one of the following criteria:

- It is a Unicode space character (`SPACE_SEPARATOR`, `LINE_SEPARATOR`, or `PARAGRAPH_SEPARATOR`) but is not also a non-breaking space (`'\u00A0'`, `'\u2007'`, `'\u202F'`).
- It is `'\t'`, U+0009 HORIZONTAL TABULATION.
- It is `'\n'`, U+000A LINE FEED.
- It is `'\u000B'`, U+000B VERTICAL TABULATION.
- It is `'\f'`, U+000C FORM FEED.
- It is `'\r'`, U+000D CARRIAGE RETURN.
- It is `'\u001C'`, U+001C FILE SEPARATOR.
- It is `'\u001D'`, U+001D GROUP SEPARATOR.
- It is `'\u001E'`, U+001E RECORD SEPARATOR.
- It is `'\u001F'`, U+001F UNIT SEPARATOR.

Module java.base
Package java.lang
Class Character

java.lang.Object
    java.lang.Character

All Implemented Interfaces:
`Serializable`, `Comparable<Character>`

Note: This method cannot handle supplementary characters. To support all Unicode characters, including supplementary characters, use the `isWhitespace(int)` method.

Parameters:
ch - the character to be tested.

Returns:
`true` if the character is a Java whitespace character; `false` otherwise.

Full List: documentation for Character.isWhitespace

Scanning in action:
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:
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
...
```

Using a different token separator – invoke ***useDelimiter()***, specifying a regular expression. Example, wanting the token separator to be a comma, optionally followed by white space. Invoke, `s.useDelimiter(",\\s*");`

I/O Streams

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.

Scanner Class

Module `java.base`
Package `java.util`

Class Scanner
`java.lang.Object`
`java.util.Scanner`

All Implemented Interfaces
`Closeable, AutoCloseable, Iterator<String>`

```
public final class Scanner
extends Object
implements Iterator<>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.
A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace.
The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System.in:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Another example:
this code allows long types to be assigned from entries in a file myNumbers

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";
    Scanner s = new
Scanner(input).useDelimiter("\\s*fish\\s*");
    System.out.println(s.nextInt());
    System.out.println(s.nextInt());
    System.out.println(s.next());
    System.out.println(s.next());
    s.close();
```

prints the following output:

```
1
2
red
blue
```

The same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

```
String input = "1 fish 2 fish red fish blue fish";
    Scanner s = new Scanner(input);
    s.findInLine("(\\d+) fish (\\d+) fish (\\w+)
fish (\\w+)");
    MatchResult result = s.match();
    for (int i=1; i<=result.groupCount(); i++)
        System.out.println(result.group(i));
    s.close();
```

The default whitespace delimiter used by a scanner is as recognized by Character.isWhitespace(). The reset() method will reset the value of the scanner's delimiter to the default whitespace delimiter regardless of whether it was previously changed.

A scanning operation may block waiting for input.

The next() and hasNext() methods and their companion methods (such as nextInt() and hasNextInt()) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both hasNext() and next() methods may block waiting for further input. Whether a hasNext() method blocks has no connection to whether or not its associated next() method will block. The tokens() method may also block waiting for input.

The findInLine(), findWithinHorizon(), skip(), and findAll() methods operate independently of the delimiter pattern. These methods will attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant. These methods may block waiting for more input.

When a scanner throws an InputMismatchException, the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method.

Depending upon the type of delimiting pattern, empty tokens may be returned. For example, the pattern "\\s+" will return no empty tokens since it matches multiple instances of the delimiter. The delimiting pattern "\\s" could return empty tokens since it only passes one space at a time.

A scanner can read text from any object which implements the Readable interface. If an invocation of the underlying readable's read() method throws an IOException then the scanner assumes that the end of the input has been reached. The most recent IOException thrown by the underlying readable can be retrieved via the ioException() method.

When a Scanner is closed, it will close its input source if the source implements the Closeable interface.

A Scanner is not safe for multithreaded use without external synchronization.

Unless otherwise mentioned, passing a null parameter into any method of a Scanner will cause a NullPointerException to be thrown.

A scanner will default to interpreting numbers as decimal unless a different radix has been set by using the useRadix(int) method. The reset() method will reset the value of the scanner's radix to 10 regardless of whether it was previously changed.

View more via Oracle Docs

I/O Streams

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 & character stream objects, instances of PrintStream and PrintWriter implement a standard set of writemethods 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.

- **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.
For a complete description, see format string syntax in the API specification.

I/O Streams

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);
    }
}
```

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 3 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.

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.

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

In Java, 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.

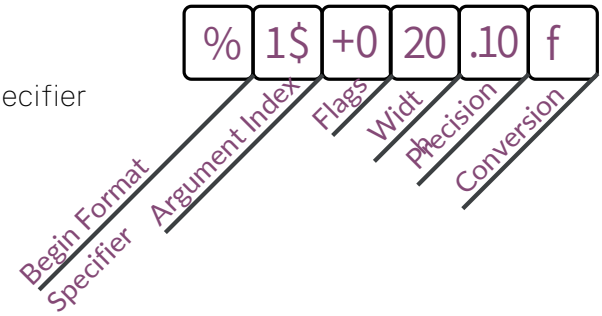
In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, Format, that uses every possible kind of element.

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

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. From right-left, 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 Streams

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.

1 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);
```

2 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.

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:
6. Invoke Console.readPassword twice to prompt for and read a new password.
7. If the user entered the same password both times, invoke change to change it. (Again, change is a dummy method.)
8. Overwrite both passwords with blanks.
9. Overwrite the old password with blanks.

I/O Streams

Password Example:

```
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.
    }
}
```

I/O Streams

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.

DataStreams example demonstrates data streams by writing out a set of data records then reading them in again. Each record consists of 3 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"

Program defines some constants containing the name of the data file & 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.

I/O Streams

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.

I/O Streams

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.

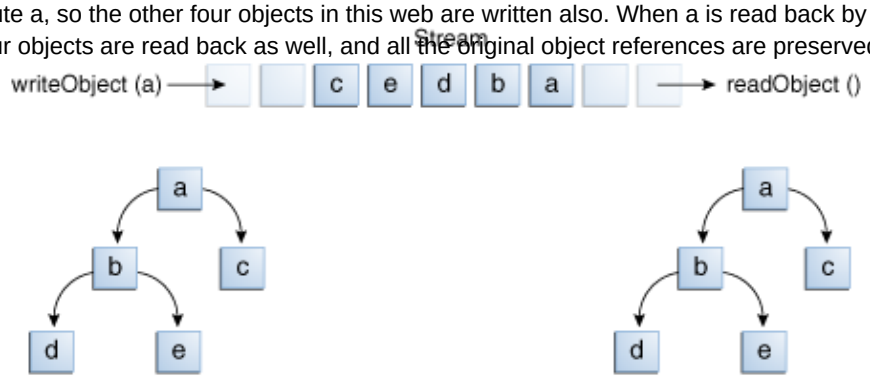
The object stream classes are ObjectInputStream and ObjectOutputStream. These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. So an object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

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

Output and Input of Complex Objects

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

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



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

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

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

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

This results in two variables, ob1 and ob2, that are references to a single object. However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

File I/O(Feat. 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.

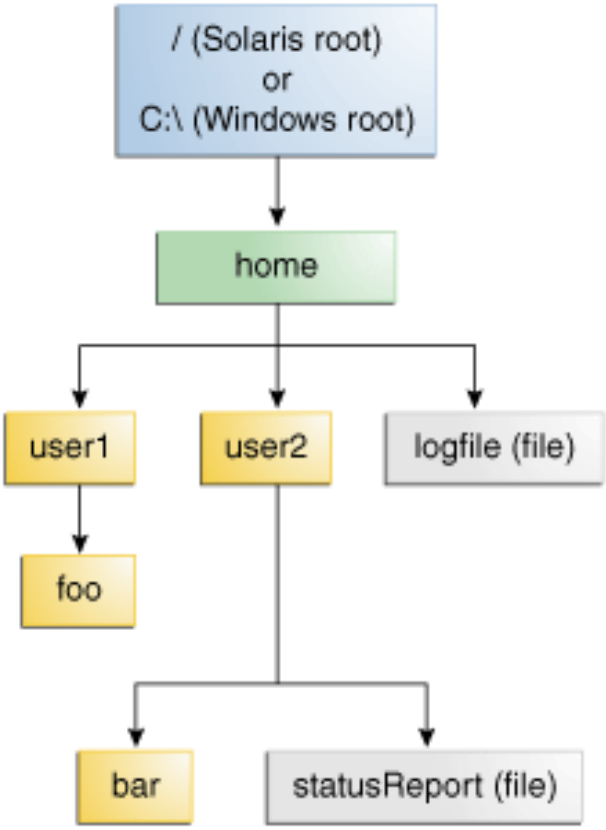
What Is a Path? (And Other File System Facts)

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.

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, /.

File I/O(Feat. NIO.2)



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.

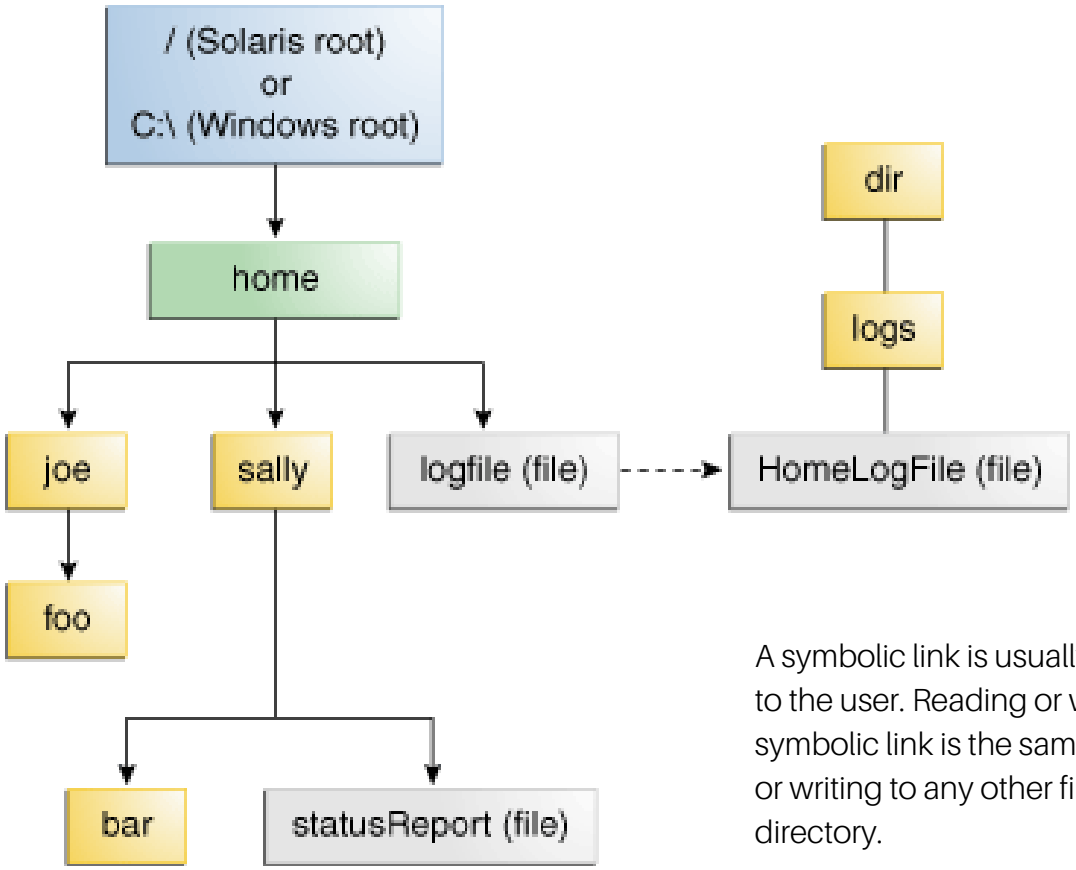
File I/O(Feat. NIO.2)

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.



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.

File I/O(Feat. NIO.2)

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.

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.

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 syntacticoperations, because they operate on the path itself and don't access the file system.

1. 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");
```

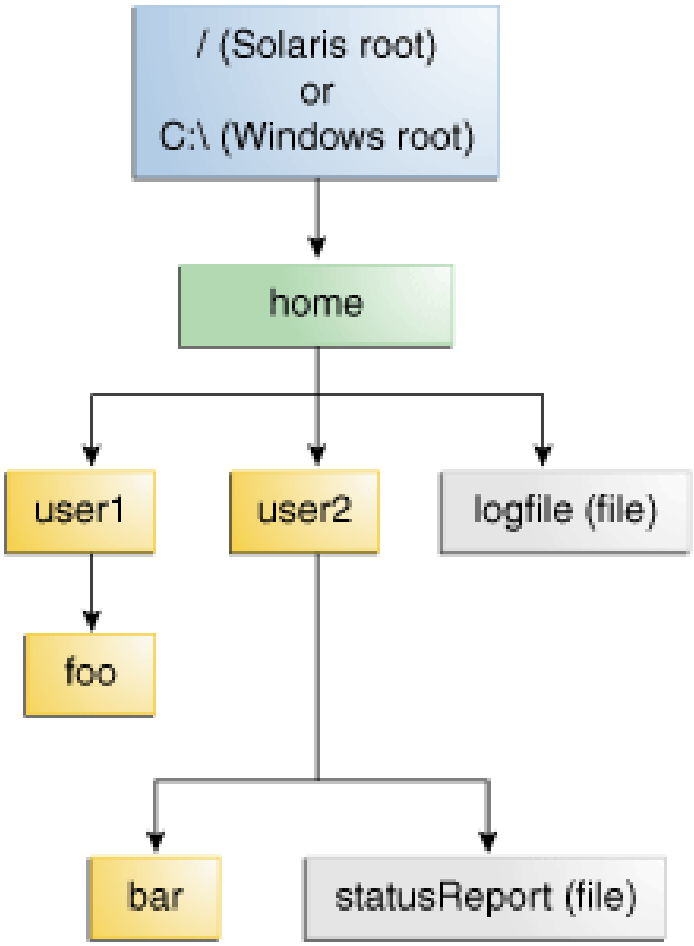
Next Eg creates /u/joe/logs/foo.log assuming your home directory is /u/joe, or C:\joe\logs\foo.log if you are on Windows.

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

2. 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());
```

File I/O(Feat. NIO.2)

Here is the output for both Windows & 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 & 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

3. 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.

File I/O(Feat. NIO.2)

4. 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:

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.

```
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 toRealPath method returns the real path of an existing file. This method performs 7 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.

Example

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

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.

File I/O(Feat. NIO.2)

5. 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:

```
// 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");
```

6. 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.

Consider 2 relative paths defined as *joe* & *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);
```

Example

The 2 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.

7. 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 I/O(Feat. NIO.2)

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.

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);
}
```

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();
}
```

File I/O(Feat. NIO.2)

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());
}
```

your code should always include exception handling

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 *CopyOptionargument* 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);
```

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.

File I/O(Feat. NIO.2)

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?

2 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 7 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.

File I/O(Feat. NIO.2)

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, & 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.

A glob pattern is specified as a string and is matched against other strings, such as directory or file names.

More glob syntax via API specification for getPathMatcher

Module java.base
Package java.nio.file

Class FileSystem

java.lang.Object
    java.nio.file.FileSystem

All Implemented Interfaces:
Closeable, AutoCloseable

```
public abstract class FileSystem
extends Object
implements Closeable
```

Provides an interface to a file system and is the factory for objects to access files and other objects in the file system.

The default file system, obtained by invoking the FileSystems.getDefault method, provides access to the file system that is accessible to the Java virtual machine. The FileSystems class defines methods to create file systems that provide access to other types of (custom) file systems.

A file system is the factory for several types of objects:

- The getPath method converts a system dependent path string, returning a Path object that may be used to locate & access a file.
- The getPathMatcher method is used to create a PathMatcher that performs match operations on paths.
- The getFileStores method returns an iterator over the underlying file-stores.
- The getUserPrincipalLookupService method returns the UserPrincipalLookupService to lookup users or groups by name.
- The newWatchService method creates a WatchService that may be used to watch objects for changes and events.

File systems vary greatly. In some cases the file system is a single hierarchy of files with one top-level root directory. In other cases it may have several distinct file hierarchies, each with its own top-level root directory. The getRootDirectories method may be used to iterate over the root directories in the file system. A file system is typically composed of one or more underlying file-stores that provide the storage for the files. Theses file stores can also vary in the features they support, and the file attributes or meta-data that they associate with files.

File I/O(Feat. NIO.2)

A file system is open upon creation and can be closed by invoking its close method. Once closed, any further attempt to access objects in the file system cause ClosedFileSystemException to be thrown. File systems created by the default provider cannot be closed.

A FileSystem can provide read-only or read-write access to the file system. Whether or not a file system provides read-only access is established when the FileSystem is created and can be tested by invoking its isReadOnly method. Attempts to write to file stores by means of an object associated with a read-only file system throws ReadOnlyFileSystemException.

File systems are safe for use by multiple concurrent threads. The close method may be invoked at any time to close a file system but whether a file system is asynchronously closeable is provider specific and therefore unspecified. In other words, if a thread is accessing an object in a file system, and another thread invokes the close method then it may require to block until the first operation is complete. Closing a file system causes all open channels, watch services, and other closeable objects associated with the file system to be closed.

getPathMatcher

```
public abstract PathMatcher getPathMatcher(String syntaxAndPattern)
```

Returns a `PathMatcher` that performs match operations on the `String` representation of `Path` objects by interpreting a given pattern. The `syntaxAndPattern` parameter identifies the syntax and the pattern and takes the form:

syntax:pattern

where `':'` stands for itself.

A `FileSystem` implementation supports the "`glob`" and "`regex`" syntaxes, and may support others. The value of the syntax component is compared without regard to case.

When the syntax is "`glob`" then the `String` representation of the path is matched using a limited pattern language that resembles regular expressions but with a simpler syntax. For example:

Example	Description
`*.java`	Matches a path that represents a file name ending in `.java`
`*.*`	Matches file names containing a dot
`*.{java,class}`	Matches file names ending with `.java` or `.class`
`foo.?`	Matches file names starting with `foo.` and a single character extension
`/home/*/*`	Matches `/home/gus/data` on UNIX platforms
`/home/**`	Matches `/home/gus` and `/home/gus/data` on UNIX platforms
`C:\\*`	Matches `C:\foo` and `C:\bar` on the Windows platform (note that the backslash is escaped; as a string literal in the Java Language the pattern would be `"C:\\\\*"`)

The following rules are used to interpret glob patterns:

- The * character matches zero or more characters of a name component without crossing directory boundaries.
- The ** characters matches zero or more characters crossing directory boundaries.
- The ? character matches exactly one character of a name component.
- The backslash character (\) is used to escape characters that would otherwise be interpreted as special characters. The expression \\ matches a single backslash and "\{" matches a left brace for example.

File I/O(Feat. NIO.2)

- The [ ] characters are a bracket expression that match a single character of a name component out of a set of characters. For example, [abc] matches "a", "b", or "c". The hyphen (-) may be used to specify a range so [a-z] specifies a range that matches from "a" to "z" (inclusive). These forms can be mixed so [abce-g] matches "a", "b", "c", "e", "f" or "g". If the character after the [is a ! then it is used for negation so [!a-c] matches any character except "a", "b", or "c".
- Within a bracket expression the *, ? and \ characters match themselves. The (-) character matches itself if it is the first character within the brackets, or the first character after the ! if negating.
- The { } characters are a group of subpatterns, where the group matches if any subpattern in the group matches. The "," character is used to separate the subpatterns. Groups cannot be nested.
- Leading period/dot characters in file name are treated as regular characters in match operations. For example, the "*" glob pattern matches file name ".login". The Files.isHidden(java.nio.file.Path) method may be used to test whether a file is considered hidden.
- All other characters match themselves in an implementation dependent manner. This includes characters representing any name-separators.
- The matching of root components is highly implementation-dependent and is not specified.

When the syntax is "regex" then the pattern component is a regular expression as defined by the Pattern class.

For both the glob and regex syntaxes, the matching details, such as whether the matching is case sensitive, are implementation-dependent and therefore not specified.

Parameters:

`syntaxAndPattern` - The syntax and pattern

Returns:

A path matcher that may be used to match paths against the pattern

Throws:

`IllegalArgumentException` - If the parameter does not take the form: `syntax:pattern`

`PatternSyntaxException` - If the pattern is invalid

`UnsupportedOperationException` - If the pattern syntax is not known to the implementation

See Also:

`Files.newDirectoryStream(Path,String)`

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.

File I/O(Feat. NIO.2)

Checking a File / 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).

Testing a file's existence:

1. The file is verified to exist.
2. The file is verified to not exist.
3. 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 can't 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
}
```

File I/O(Feat. NIO.2)

Deleting a File / 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 2 deletion methods.

`delete`

The delete(Path) method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist a NoSuchFileExceptionis 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);
}
```

`deleteIfExists`

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.

File I/O(Feat. NIO.2)

Copying a File / 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 & LinkOption enums are supported:

1. 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 DirectoryNotEmptyException exception.
2. 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.
3. 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).

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.

File I/O(Feat. NIO.2)

Moving a File / 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_EXISTINGoption 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:

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

1. 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.
2. 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_MOVEyou 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.

File I/O(Feat. NIO.2)

Managing Metadata (File & 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.

File I/O(Feat. NIO.2)

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.

Supported views:

- 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.

File I/O(Feat. NIO.2)

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

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

The remainder of this section covers the following topics:

- Basic File Attributes
- Setting Time Stamps
- DOS File Attributes
- POSIX File Permissions
- Setting a File or Group Owner
- User-Defined File Attributes
- File Store Attributes

Basic File Attributes

As mentioned previously, to read the basic attributes of a file, you can use one of the Files.readAttributes methods, which reads all the basic attributes in one bulk operation. This is far more efficient than accessing the file system separately to read each individual attribute. The varargs argument currently supports the LinkOptionenum, 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.

File I/O(Feat. NIO.2)

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 PosixFileAttributesclass.

```
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 umaskalso applies, so the new file might be more secure than the permissions that were requested.

```
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);
```

File I/O(Feat. NIO.2)

User-Defined File Attributes

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

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

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

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

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

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

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

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

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

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

File Store Attributes

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

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

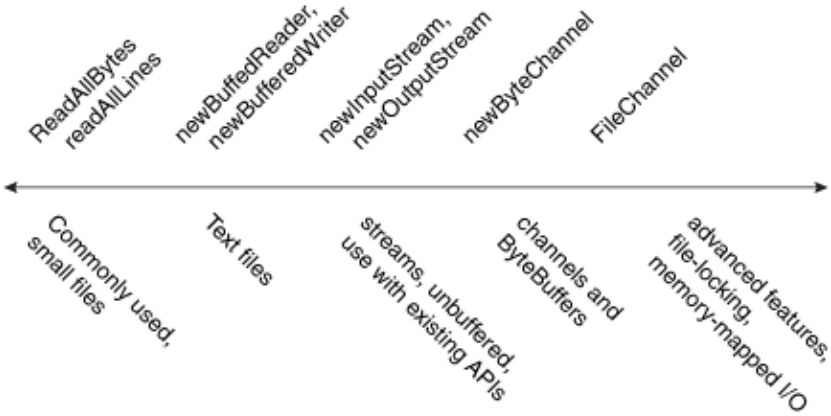
```
Path file = ...;
FileStore store = Files.getFileStore(file);

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

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

File I/O(Feat. NIO.2)

Reading, Writing & 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.



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.

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.

File I/O(Feat. NIO.2)

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);
}
```

File I/O(Feat. NIO.2)

Methods for Unbuffered Streams & 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 & 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 example 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.*;
import java.nio.file.*;
import java.io.*;

public class LogFileTest {

  public static void main(String[] args) {

    // Convert the string to a
    // byte array.
    String s = "Hello World! ";
    byte data[] = s.getBytes();
    Path p = Paths.get("./logfile.txt");

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

File I/O(Feat. NIO.2)

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 are specified, then the channel is opened for reading.

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

```
public static void readFile(Path path) throws IOException {

    // Files.newByteChannel() defaults to StandardOpenOption.READ
    try (SeekableByteChannel sbc = Files.newByteChannel(path)) {
        final int BUFFER_CAPACITY = 10;
        ByteBuffer buf = ByteBuffer.allocate(BUFFER_CAPACITY);

        // Read the bytes with the proper encoding for this platform. If
        // you skip this step, you might see foreign or illegible
        // characters.
        String encoding = System.getProperty("file.encoding");
        while (sbc.read(buf) > 0) {
            buf.flip();
            System.out.print(Charset.forName(encoding).decode(buf));
            buf.clear();
        }
    }
}
```

File I/O(Feat. NIO.2)

The following example, 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.StandardOpenOption.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.*;
import java.util.*;

public class LogFilePermissionsTest {

  public static void main(String[] args) {

    // Create the set of options for appending to the file.
    Set<OpenOption> 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 = "Hello World! ";
    byte data[] = s.getBytes();
    ByteBuffer bb = ByteBuffer.wrap(data);

    Path file = Paths.get("./permissions.log");

    try (SeekableByteChannel sbc =
      Files.newByteChannel(file, options, attr)) {
      sbc.write(bb);
    } catch (IOException x) {
      System.out.println("Exception thrown: " + x);
    }
  }
}
```

Methods for Creating Regular & 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.

File I/O(Feat. NIO.2)

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.

File I/O(Feat. NIO.2)

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.newByteChannelmethods 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);
}
```

File I/O(Feat. NIO.2)

Creating and Reading Directories

Some of the methods previously discussed, such as delete, work on files, links anddirectories. 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
- Creating a Directory
- Creating a Temporary Directory
- Listing a Directory's Contents
- Filtering a Directory Listing By Using Globbing
- Writing Your Own Directory Filter

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.

File I/O(Feat. NIO.2)

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-file 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 finallyblock. 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);
}
```

File I/O(Feat. NIO.2)

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.

File I/O(Feat. NIO.2)

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 Pathmethods work seamlessly with hard links.

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

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

Creating a Symbolic Link

If your file system supports it, you can create a symbolic link by using the createSymbolicLink(Path, Path, FileAttribute<?>) method. The second Pathargument 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.

File I/O(Feat. NIO.2)

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.

File I/O(Feat. NIO.2)

Walking the File Tree

You create an application that will recursively visit all the files in a file tree or delete every .class file in a tree, or find every file that hasn't been accessed in the last year by using the FileVisitor interface.

1 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 4 methods that correspond to these situations:

1. preVisitDirectory – Invoked before a directory's entries are visited.
2. postVisitDirectory – Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
3. 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 DosFileAttributeViewto determine if the file has the "hidden" bit set.
4. 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 4 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.

```
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;
    }
}
```

File I/O(Feat. NIO.2)

2 Kickstarting the Process

Once you have implemented your FileVisitor, how do you initiate the file walk? There are 2 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);
```

3 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.

EG: 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.

This case can occur only when the program is following symbolic links.

```
@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;
}
```

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.

File I/O(Feat. NIO.2)

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. Pattern matching uses special characters to create a pattern and then file names can be compared against that pattern. In most shell scripts, the asterisk, *, matches any number of characters.

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:

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"`

File I/O(Feat. NIO.2)

Here is the source code for the Find example:

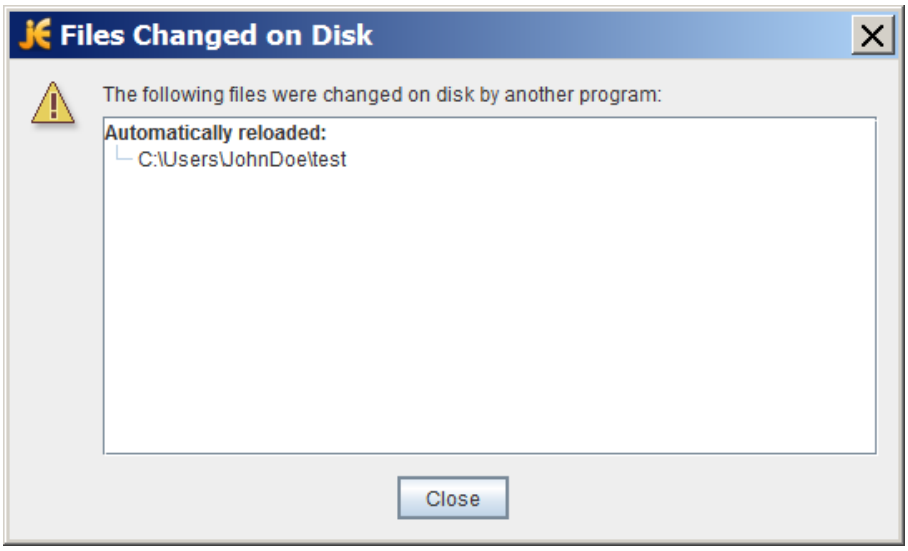
```java
/**
 * Sample code that finds files that match the specified glob pattern.
 * For more information on what constitutes a glob pattern, see
 * https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob
 *
 * The file or directories that match the pattern are printed to
 * standard 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.

File I/O(Feat. NIO.2)

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.

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.

File I/O(Feat. NIO.2)

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 WatchEventsfrom 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.

(next page)

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.

File I/O(Feat. NIO.2)

Other Useful Methods

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 determined 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 I/O(Feat. NIO.2)

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 retrieve 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.

File I/O(Feat. NIO.2)

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.

File I/O (Feat. NIO.2)

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.filepackage, your easiest solution is to use the File.toPath method as suggested in the previous section. Don't want to use that approach/insufficient 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`	`Files.createDirectory`	Creating a Directory
`File.listRoots`	`FileSystem.getRootDirectories`	Listing a File System's Root Directories
`File.getTotalSpace`, `File.getFreeSpace`, `File.getUsableSpace`	`FileStore.getTotalSpace`, `FileStore.getUnallocatedSpace`, `FileStore.getUsableSpace`, `FileStore.getTotalSpace`	File Store Attributes

Summary

The java.io package contains many classes that your programs can use to read and write data. Most of the classes implement sequential access streams. The sequential access streams can be divided into two groups: those that read and write bytes and those that read and write Unicode characters. Each sequential access stream has a speciality, such as reading from or writing to a file, filtering data as its read or written, or serializing an object.

The java.nio.file package provides extensive support for file and file system I/O. This is a very comprehensive API, but the key entry points are as follows:

- The Path class has methods for manipulating a path.
- The Files class has methods for file operations, such as moving, copy, deleting, and also methods for retrieving and setting file attributes.
- The FileSystem class has a variety of methods for obtaining information about the file system.

More information on NIO.2 can be found on the OpenJDK: NIO project website. This site includes resources for features provided by NIO.2 that are beyond the scope of this tutorial, such as multicasting, asynchronous I/O, and creating your own file system implementation.