**BASIC I/O**

* **IO Streams**
  + An I/O Streams 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.
  + A stream is a *sequence of data*.
  + A program uses an input stream to *read data from a source*, one item at a time.

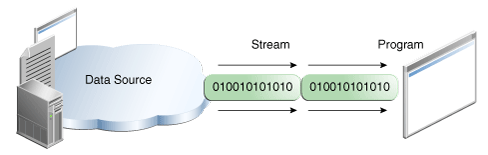


Fig: Reading information into a program.

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

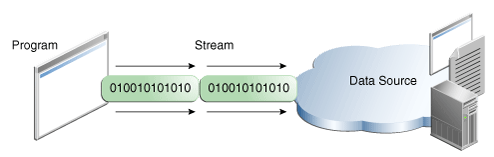


Fig: Writing information from a program.

* + For sample input, we'll use the *example file* input.txt, which contains the following verse:

In Xanadu did Kubla Khan

A stately pleasure-dome decree:

Where Alph, the sacred river, ran

Through caverns measureless to man

Down to a sunless sea.

* **Byte Streams**
  + Programs use byte streams to perform input and output of *8-bit bytes*.
  + All byte stream classes are descended from *InputStream* and *OutputStream*.
  + **Using Byte Streams**
    - We'll explore *FileInputStream* and *FileOutputStream* by examining an example program named CopyBytes.
    - Which uses byte streams to copy xanadu.txt, one byte at a time.

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class CopyBytes {

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

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("input.txt");

out = new FileOutputStream("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

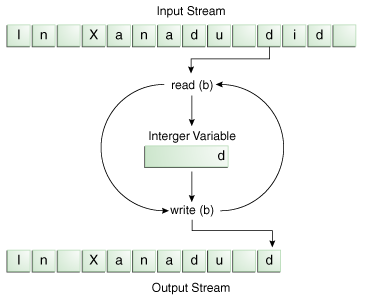
}

}

}

}

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



Fiig: Simple byte stream input and output.

* + - Notice that read() returns an *int value*.
    - If the input is a stream of bytes, why doesn't read() return *a byte value*?
    - Using a int as a return type allows read() to use -1 to indicate that it has *reached the end of the stream*.
  + **Always Close Streams**
    - *Closing a stream* when it's no longer needed is very important.
    - So important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs.
    - This practice helps avoid serious *resource leaks*.
  + **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.
    - There are also streams for more complicated data types. Byte streams should only be used for *the most primitive I/O.*
* **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.
  + **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();

}

}

}

}

* + - Notice that both CopyBytes and CopyCharacters use an *int variable* to read to and write from.
    - However, in CopyCharacters, the *int variable* holds a *character value in its last 16 bits*; in CopyBytes, the *int variable* holds *a byte value in its last 8 bits*.
  + **Character Streams that Use Byte Streams**
    - Character streams are often "*wrappers*" for byte streams.
    - The character stream uses the byte stream to perform the *physical I/O*.
    - While the character stream handles translation between characters and bytes.
    - FileReader, for example, uses *FileInputStream*, while FileWriter uses *FileOutputStream*.
    - There are two general-purpose byte-to-character "bridge" streams: *InputStreamReader* and *OutputStreamWriter*.
    - Use them to create character streams when there are no prepackaged character stream classes that meet your needs.
  + **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.
    - The CopyLines example invokes *BufferedReader.readLine* and *PrintWriter.println* to do input and output one line at a time.

import java.io.FileReader;

import java.io.FileWriter;

import java.io.BufferedReader;

import java.io.PrintWriter;

import java.io.IOException;

public class CopyLines {

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

BufferedReader inputStream = null;

PrintWriter outputStream = null;

try {

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

String l;

while ((l = inputStream.readLine()) != null) {

outputStream.println(l);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

* + - Invoking *readLine* returns *a line of text* with the line.
    - CopyLines *outputs* each line using *println*, which *appends* the line terminator for the current operating system.
    - This might not be the same line terminator that was used in the input file.
* **Buffered Streams**
  + Most of the examples we have seen so far use *unbuffered I/O*.
  + This means each read or writes 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’s 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 CopyCharacter example to use buffered I/O.

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

* + There are *four buffered stream classes* used to wrap unbuffered streams.
    - *BufferedInputStream* and *BufferedOutputStream* create buffered byte streams
    - While *BufferedReader* and BufferedWriter create buffered character stream.
  + **Flushing Buffered Streams**
    - It often makes sense *to write out a buffer* at critical points, without waiting for it to fill.
    - This is known as *flushing* the buffer.
    - Some buffered output classes support *autoflush*, specified by an optional constructor argument.
    - When autoflush is enabled, *certain key events* cause the buffer to be flushed.
    - For example, an autoflush PrintWriter object flushes the buffer on every invocation of *println* or format.
    - To flush a stream manually, invoke its *flush method*.
    - The flush method is *valid* on any output stream, but has *no effect* unless the stream is buffered.
* **Scanning And Formatting**
* **Scanning**
  + Objects of type *scanner* are useful for breaking down *formatted input* into *tokens* and translating individual tokens according to their *data type*.
  + **Breaking Input into Tokens**
    - By default, a *scanner* uses whitespace to separate *tokens*.
    - White space characters include *blanks*, *tabs* and *line terminators*.

import java.io.\*;

import java.util.Scanner;

public class ScanXan {

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

Scanner s = null;

try {

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

while (s.hasNext()) {

System.out.println(s.next());

}

} finally {

if (s != null) {

s.close();

}

}

}

}

* + - Notice that ScanXan invoke Scanner’s *close method* when it’s done with the scanner object.
    - Even though a *scanner is not a stream*, you need to close it to indicate that you are done with its underlying stream.
    - The output of ScanXan looks like this

In

Xanadu

did

Kubla

Khan

A

stately

pleasure-dome

...

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

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

* + **Translating Individual Tokens**
    - The ScanXan example treats all input tokens as *simple String values*.
    - Scanner also supports tokens for all of the java language’s *primitive types* (except for char) as well as *BigInteger* and *BigDecimal*.
    - Also numeric values can use *thousands separators*.
    - Thus, in *US locale*, Scanner correctly reads the string “32.122” as representing an integer value.
    - We have to mention the locale, because *thousand separators* and *decimal symbols* are locale specific.
    - 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](about:blank)

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 does not provide a way to override the default locale.
* **Formatting**
  + Stream objects that implement formatting are instance of either PrintWriter a character stream class or PrintStream a byte stream class.
  + Like all byte and character stream objects, instance of PrintStream and PrintWriter implements a standard set of write methods for simple byte and character output.
  + In addition, both PrintStream and PrintWriter implements the same set of methods for converting internal data into formatted output.
  + Two levels of formatting are provided.
    - print and println format individual values in a standard way.
    - Format formats almost any number of values based on a format string, with many options for precise formatting.
  + **The print and println methods**
    - Invoking print or println outputs a single value after converting the value using the appropriate toString method.

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 the code in an overload of print.
    - The second time by conversion code automatically generated by the Java compiler, which also utilize toString.
    - You can format any value this way.
    - But you don’t have much control over the result.
  + **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.

public class Root2 {

public static void main(String[] args) {

int i = 2;

double r = Math.sqrt(i);

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

}

}

Here is the output:

The square root of 2 is 1.414214.

* + - All format specifiers begin with a % and end with a 1 or 2 character conversion that specifies the kind of formatted output being generated.
    - The three conversions used here are
      * d formats an integer value as a decimal value
      * f formats a floating point value as a decimal value
      * n outputs a platform specific line terminator
    - Here are some other conversions
      * x formats an integer value as a hexadecimal value
      * s formats any value as a string
      * tB formats an integer as a locale specific month name
* **I/O from 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
    - Through the console
  + **Standard Streams**
    - Standard Streams are a feature of many operating systems.
    - By default, they read input from the keyboard and write output to the display.
    - They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.
    - The Java platform supports three Standard Streams:
      * Standard Input, accessed through System.in.
      * Standard Output, accessed through System.out.
      * And Standard Error, accessed through System.err.
    - These objects are defined automatically and do not need to be opened.
    - Standard Output and Standard Error are both for output.
    - Having error output separately allows the user to divert regular output to a file and still be able to read error messages.
    - You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams.
    - System.out and System.err are defined as PrintStream objects.
    - Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.
    - By contrast, System.in is a byte stream with no character stream features.
    - To use Standard Input as a character stream, wrap System.in in InputStreamReader.

InputStreamReader cin = new InputStreamReader(System.in);

* + **The Console**
    - A more advanced alternative to the Standard Streams is the Console.
    - This is a single, predefined object of type Console that has most of the features provided by the Standard Streams, and others besides.
    - The Console is particularly useful for secure password entry.
    - The Console object also provides input and output streams that are true character streams, through its reader and writer methods.
    - Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console().
    - If the Console object is available, this method returns it.
    - If System.console returns NULL, then Console operations are not permitted.
    - Either because the OS doesn't support them or because the program was launched in a noninteractive environment.
    - The Console object supports secure password entry through its readPassword method.
    - This method helps secure password entry in two ways.
      * First, it suppresses echoing, so the password is not visible on the user's screen.
      * Second, readPassword returns a character array, not a String.
    - So the password can be overwritten, removing it from memory as soon as it is no longer needed.
    - The Password example is a prototype program for changing a user's password.
    - It demonstrates several Console methods.

import java.io.Console;

import java.util.Arrays;

import java.io.IOException;

public class Password {

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

Console c = System.console();

if (c == null) {

System.err.println("No console.");

System.exit(1);

}

String login = c.readLine("Enter your login: ");

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

if (verify(login, oldPassword)) {

boolean noMatch;

do {

char [] newPassword1 = c.readPassword("Enter your new password: ");

char [] newPassword2 = c.readPassword("Enter new password again: ");

noMatch = ! Arrays.equals(newPassword1, newPassword2);

if (noMatch) {

c.format("Passwords don't match. Try again.%n");

} else {

change(login, newPassword1);

c.format("Password for %s changed.%n", login);

}

Arrays.fill(newPassword1, ' ');

Arrays.fill(newPassword2, ' ');

} while (noMatch);

}

Arrays.fill(oldPassword, ' ');

}

// Dummy change method.

static boolean verify(String login, char[] password) {

// This method always returns

// true in this example.

// Modify this method to verify

// password according to your rules.

return true;

}

// Dummy change method.

static void change(String login, char[] password) {

// Modify this method to change

// password according to your rules.

}

}

* **Data Streams**
  + Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.
  + All data streams implement either the DataInput interface or the DataOutput interface.
  + This section focuses on the most widely-used implementations of these interfaces, DataInputStream and DataOutputStream.
  + The DataStreams example demonstrates data streams by writing out a set of data records, and then reading them in again.
  + Each record consists of three values related to an item on an invoice, as shown in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Order in record** | **Data type** | **Data description** | **Output Method** | **Input Method** | **Sample Value** |
| 1 | double | Item price | DataOutputStream.writeDouble | DataInputStream.readDouble | 19.99 |
| 2 | int | Unit count | DataOutputStream.writeInt | DataInputStream.readInt | 12 |
| 3 | String | Item description | DataOutputStream.writeUTF | DataInputStream.readUTF | "Java T-Shirt" |

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

static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };

static final int[] units = { 12, 8, 13, 29, 50 };

static final String[] descs = {

"Java T-shirt",

"Java Mug",

"Duke Juggling Dolls",

"Java Pin",

"Java Key Chain"

};

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

out = new DataOutputStream(new BufferedOutputStream(

new FileOutputStream(dataFile)));

* + DataStreams writes out the records and closes the output stream.

for (int i = 0; i < prices.length; i ++) {

out.writeDouble(prices[i]);

out.writeInt(units[i]);

out.writeUTF(descs[i]);

}

* + The writeUTF method writes out String values in a modified form of UTF-8.
  + This is a variable-width character encoding that only needs a single byte for common Western characters.
  + Now DataStreams reads the data back in again.
  + First it must provide an input stream, and variables to hold the input data.
  + Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

in = new DataInputStream(new

BufferedInputStream(new FileInputStream(dataFile)));

double price;

int unit;

String desc;

double total = 0.0;

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

try {

while (true) {

price = in.readDouble();

unit = in.readInt();

desc = in.readUTF();

System.out.format("You ordered %d" + " units of %s at $%.2f%n",

unit, desc, price);

total += unit \* price;

}

} catch (EOFException e) {

}

* + Notice that DataStreams detects an end-of-file condition by catching EOFException, instead of testing for an invalid return value.
  + All implementations of DataInput methods use EOFException instead of return values.
  + Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read.
  + It is up to the programmer to make sure that output types and input types are matched in this way:
  + The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.
  + DataStreams uses one very bad programming technique:
  + It uses floating point numbers to represent monetary values.
  + In general, floating point is bad for precise values.
  + It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.
  + The correct type to use for currency values is java.math.BigDecimal.
  + Unfortunately, BigDecimal is an object type, so it won't work with data streams.
  + However, BigDecimal will work with object streams.
* **Object Streams**
  + Just as data streams support I/O of primitive data types
  + Object streams support I/O of objects.
  + Most, but not all, standard classes support serialization of their objects.
  + Those that do implement the marker interface Serializable.
  + The object stream classes are ObjectInputStream and ObjectOutputStream.
  + These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.
  + That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams.
  + So an object stream can contain a mixture of primitive and object values.
  + The ObjectStreams example illustrates this.
  + ObjectStreams creates the same application as DataStreams, with a couple of changes.
  + First, prices are now 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.

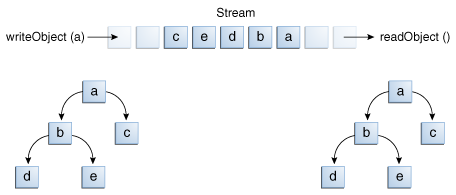


Fig : I/O of multiple referred-to objects

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

*Object ob = new Object();*

*out.writeObject(ob);*

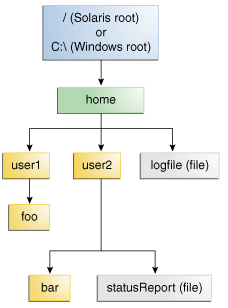
*out.writeObject(ob);*

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

*Object ob1 = in.readObject();*

*Object ob2 = in.readObject();*

* + - This results in two variables, ob1 and ob2, that are references to a single object.
    - However, if a single object is written to two different streams, it is effectively duplicated.
    - A single program reading both streams back will see two distinct objects.
* **File I/O**
* **What Is a Path**
  + 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, /.



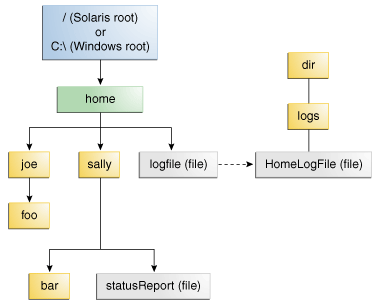
* + - A file is identified by its path through the file system, beginning from the root node.
    - For example, the statusReport file in the previous figure is described by the following notation in the Solaris OS:

*/home/sally/statusReport*

* + - In Microsoft Windows, statusReport is described by the following notation:

*C:\home\sally\statusReport*

* + - The character used to separate the directory names (also called the delimiter) is specific to the file system.
    - The Solaris OS uses the forward slash (/)
    - And Microsoft Windows uses the backslash slash (\).
  + **Relative or Absolute**
    - A path is either relative or absolute.
    - An absolute path always contains the root element and the complete directory list required to locate the file.
    - For example, /home/sally/statusReport is an absolute path.
    - All of the information needed to locate the file is contained in the path string.
    - A relative path needs to be combined with another path in order to access a file.
    - For example, joe/foo is a relative path.
    - Without more information, a program cannot reliably locate the joe/foo directory in the file system.
  + **Symbolic Links**
    - File system objects are most typically directories or files.
    - Everyone is familiar with these objects.
    - But some file systems also support the notion of symbolic links.
    - A symbolic link is also referred to as a symlink or a soft link.
    - A symbolic link is a special file that serves as a reference to another file.
    - For the most part, symbolic links are transparent to applications,
    - And operations on symbolic links are automatically redirected to the target of the link.
    - The file or directory being pointed to is called the target of the link.
    - Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.
    - In the following figure, logFile appears to be a regular file to the user, but it is actually a symbolic link to dir/logs/HomeLogFile.
    - HomeLogFile is the target of the link.



* + - 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.
* **The Path Class**
  + The Path class, introduced in the Java SE 7 release
  + It is one of the primary entrypoints of the java.nio.file package.
  + 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.

* **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 is sometimes called syntactic operations,
  + Because they operate on the path itself and don't access the file system.
  + **Creating a Path**
    - A Path instance contains the information used to specify the location of a file or directory.
    - At the time it is defined, a Path is provided with a series of one or more names.
    - A root element or a file name might be included, but neither are required.
    - A Path might consist of just a single directory or file name.
    - You can easily create a Path object by using one of the following get methods from the Paths (note the plural) helper class:

*Path p1 = Paths.get("/tmp/foo");*

*Path p2 = Paths.get(args[0]);*

*Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));*

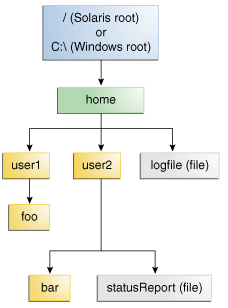
* + - The Paths.get method is shorthand for the following code:

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

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

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

* + **Retrieving Information about a Path**
    - You can think of the Path as storing these name elements as a sequence.
    - The highest element in the directory structure would be located at index 0.
    - The lowest element in the directory structure would be located at index [n-1],
    - Where n is the number of name elements in the Path.
    - Methods are available for retrieving individual elements or a subsequence of the Path using these indexes.
    - The examples in this lesson use the following directory structure.



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

// None of these methods requires that the file corresponding

// to the Path exists.

// Microsoft Windows syntax

Path path = Paths.get("C:\\home\\joe\\foo");

// Solaris syntax

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

System.out.format("toString: %s%n", path.toString());

System.out.format("getFileName: %s%n", path.getFileName());

System.out.format("getName(0): %s%n", path.getName(0));

System.out.format("getNameCount: %d%n", path.getNameCount());

System.out.format("subpath(0,2): %s%n", path.subpath(0,2));

System.out.format("getParent: %s%n", path.getParent());

System.out.format("getRoot: %s%n", path.getRoot());

* + - Here is the output for both Windows and the Solaris OS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Method Invoked** | **Returns in the Solaris OS** | **Returns in Microsoft Windows** | **Comment** |
| toString | /home/joe/foo | C:\home\joe\foo | Returns the string representation of the Path. If the path was created usingFilesystems.getDefault().getPath(String) or Paths.get (the latter is a convenience method forgetPath), the method performs minor syntactic cleanup. For example, in a UNIX operating system, it will correct the input string //home/joe/foo to /home/joe/foo. |
| getFileName | foo | Foo | Returns the file name or the last element of the sequence of name elements. |
| getName(0) | home | Home | Returns the path element corresponding to the specified index. The 0th element is the path element closest to the root. |
| getNameCount | 3 | 3 | Returns the number of elements in the path. |
| subpath(0,2) | home/joe | home\joe | Returns the subsequence of the Path (not including a root element) as specified by the beginning and ending indexes. |
| getParent | /home/joe | \home\joe | Returns the path of the parent directory. |
| getRoot | / | C:\ | Returns the root of the path. |

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

// Solaris syntax

Path path = Paths.get("sally/bar");

or

// Microsoft Windows syntax

Path path = Paths.get("sally\\bar");

* + - Here is the output for Windows and the Solaris OS:

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

* + **Removing Redundancies From a Path**
    - Many file systems use "." notation to denote the current directory
    - And ".." to denote the parent directory.
    - You might have a situation where a Path contains redundant directory information.
    - Perhaps a server is configured to save its log files in the "/dir/logs/." directory,
    - And you want to delete the trailing "/." notation from the path.
    - The following examples both include redundancies:
      * /home/./joe/foo
      * /home/sally/../joe/foo
    - The normalize method removes any redundant elements, which includes any "." or "directory/.." occurrences.
    - Both of the preceding examples normalize to /home/joe/foo.
    - It is important to note that normalize doesn't check at the file system when it cleans up a path.
    - It is a purely syntactic operation.
    - In the second example, if sally were a symbolic link, removing sally/.. might result in a Path that no longer locates the intended file.
    - To clean up a path while ensuring that the result locates the correct file, you can use the toRealPath method.
  + **Converting a Path**
    - You can use three methods to convert the Path.
    - If you need to convert the path to a string that can be opened from a browser, you can use toUri. For example:

Path p1 = Paths.get("/home/logfile");

// Result is **file:///home/logfile**

System.out.format("%s%n", p1.toUri());

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

public class FileTest {

public static void main(String[] args) {

if (args.length < 1) {

System.out.println("usage: FileTest file");

System.exit(-1);

}

// Converts the input string to a Path object.

Path inputPath = Paths.get(args[0]);

**// Converts the input Path**

**// to an absolute path.**

**// Generally, this means prepending**

**// the current working**

**// directory. If this example**

**// were called like this:**

**// java FileTest foo**

**// the getRoot and getParent methods**

**// would return null**

**// on the original "inputPath"**

**// instance. Invoking getRoot and**

**// getParent on the "fullPath"**

**// instance returns expected values.**

**Path fullPath = inputPath.toAbsolutePath();**

}

}

* + - The toAbsolutePath method converts the user input and returns a Path that returns useful values when queried.
    - The file does not need to exist for this method to work.
    - The toRealPath method returns the real path of an existing file. This method performs several operations in one:
      * If true is passed to this method and the file system supports symbolic links, this method resolves any symbolic links in the path.
      * If the Path is relative, it returns an absolute path.
      * If the Path contains any redundant elements, it returns a path with those elements removed.
      * This method throws an exception if the file does not exist or cannot be accessed.
    - You can catch the exception when you want to handle any of these cases. For example:

try {

Path fp = path.toRealPath(true);

} catch (NoSuchFileException x) {

System.err.format("%s: no such" + " file or directory%n", path);

// Logic for case when file doesn't exist.

} catch (IOException x) {

System.err.format("%s%n", x);

// Logic for other sort of file error.

}

* + **Joining Two Paths**
    - You can combine paths by using the resolve method.
    - You pass in a partial path , which is a path that does not include a root element,
    - And that partial path is appended to the original path.
    - For example, consider the following code snippet:

// Solaris

Path p1 = Paths.get("/home/joe/foo");

// Result is **/home/joe/foo/bar**

System.out.format("%s%n", p1.resolve("bar"));

or

// Microsoft Windows

Path p1 = Paths.get("C:\\home\\joe\\foo");

// Result is **C:\home\joe\foo\bar**

System.out.format("%s%n", p1.resolve("bar"));

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

// Result is **/home/joe**

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

* + **Creating a Path Between Two Paths**
    - A common requirement when you are writing file I/O code is the capability to construct a path from one location in the file system to another location.
    - You can meet this using the relativize method.
    - This method constructs a path originating from the original path and ending at the location specified by the passed-in path.
    - The new path is relative to the original path.
    - For example, consider two relative paths defined as joe and sally:

Path p1 = Paths.get("joe");

Path p2 = Paths.get("sally");

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

// Result is **../sally**

Path p1\_to\_p2 = p1.relativize(p2);

// Result is **../joe**

Path p2\_to\_p1 = p2.relativize(p1);

* + - Consider a slightly more complicated example:

Path p1 = Paths.get("home");

Path p3 = Paths.get("home/sally/bar");

// Result is **sally/bar**

Path p1\_to\_p3 = p1.relativize(p3);

// Result is **../..**

Path p3\_to\_p1 = p3.relativize(p1);

* + - In this example, the two paths share the same node, home.
    - To navigate from home to bar, you first navigate one level down to sally and then one more level down to bar.
    - Navigating from bar to home requires moving up two levels.
    - A relative path cannot be constructed if only one of the paths includes a root element.
    - If both paths include a root element, the capability to construct a relative path is system dependent.
  + **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.
    - When you want to verify that two Path objects locate the same file, you can use the isSameFile method.
* **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, handles this step for you.
  + **Catching Exceptions**
    - With file I/O, unexpected conditions are a fact of life
    - 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.
    - 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.
    - 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());

}

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

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

* + - When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (CopyOption[]) of values.
    - In the move example, the method can be invoked as follows:

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

Path source = ...;

Path target = ...;

Files.move(source,

target,

REPLACE\_EXISTING,

ATOMIC\_MOVE);

* + **Automic Operations**
    - Several Files methods, such as move, can perform certain operations atomically in some file systems.
    - An atomic file operation is an operation that cannot be interrupted or "partially" performed.
    - Either the entire operation is performed or the operation fails.
    - This is important when you have multiple processes operating on the same area of the file system,
    - And you need to guarantee that each process accesses a complete file.
  + **Method Chaining**
    - Many of the file I/O methods support the concept of method chaining.
    - You first invoke a method that returns an object.
    - You then immediately invoke a method on that object,
    - Which returns yet another object, and so on.
    - Many of the I/O examples use the following technique:

String value = Charset.defaultCharset().decode(buf).toString();

UserPrincipal group =

file.getFileSystem().getUserPrincipalLookupService().

lookupPrincipalByName("me");

* + **What Is a Glob**
    - Two methods in the Files class accept a glob argument, but what is a glob?
    - You can use glob syntax to specify pattern-matching behaviour.
    - A glob pattern is specified as a string and is matched against other strings, such as directory or file names.
    - Glob syntax follows several simple rules:
      * An asterisk, \*, matches any number of characters (including none).
      * Two asterisks, \*\*, works like \* but crosses directory boundaries. This syntax is generally used for matching complete paths.
      * A question mark, ?, matches exactly one character.
      * Braces specify a collection of subpatterns. For example:
        + {sun,moon,stars} matches "sun", "moon", or "stars."
        + {temp\*,tmp\*} matches all strings beginning with "temp" or "tmp."
      * Square brackets convey a set of single characters or, when the hyphen character (-) is used, a range of characters. For example:
      * [aeiou] matches any lowercase vowel.
      * [0-9] matches any digit.
      * [A-Z] matches any uppercase letter.
      * [a-z,A-Z] matches any uppercase or lowercase letter.
      * Within the square brackets, \*, ?, and \ match themselves.
      * All other characters match themselves.
      * To match \*, ?, or the other special characters, you can escape them by using the backslash character, \.
      * For example: \\ matches a single backslash, and \? matches the question mark.
    - Here are some examples of glob syntax:
      * \*.html – Matches all strings that end in .html
      * ??? – Matches all strings with exactly three letters or digits
      * \*[0-9]\* – Matches all strings containing a numeric value
      * \*.{htm,html,pdf} – Matches any string ending with .htm, .html or .pdf
      * a?\*.java – Matches any string beginning with a, followed by at least one letter or digit, and ending with .java
      * {foo\*,\*[0-9]\*} – Matches any string beginning with foo or any string containing a numeric value
  + **Link Awareness**
    - The Files class is a *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 behaviour* when a symbolic link is encountered.
* **Checking a File or Directory**
  + You have a *Path instance* representing a file or directory.
  + But does that *file exist* on the file system? Is it *Readable*? *Writable*? *Executable*?
  + **Verifying the Existence of a File or a 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 *notExists(Path, LinkOption…)* methods.
    - Note that !Files.exists(path) is *not equivalent* to Files.notExists(path)
    - When you are testing a file’s existence three results are possible.
      * The file is verified to *exist*.
      * The file is verified to *not exist*.
      * The file’s status is unknown. This result can occur when the program does not have access to the file.
    - If both exists and notExists return false, the *existence* of the file cannot be verified.
  + **Checking File Accessibility**
    - To verify that the program can access the file as needed,
    - You can use the isReadable(path), isWriteable(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);

* + **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.

Path p1 = ...;

Path p2 = ...;

if (Files.isSameFile(p1, p2)) {

// Logic when the paths locate the same file

}

* **Deleting a File or Directory**
  + You can delete files, directories or links.
  + With symbolic links, the link is deleted and not the target of the link.
  + With directories, the directory must be empty or the deletion fails.
  + The Files class provides two deletion methods.
    - delete(Path)
    - deleteIfExists(Path)
  + The delete(Path) method deletes the file or throws an exception if the deletion fails.
  + 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.

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);

}

* **Copying a File or Directory**
  + You can copy a file or directory by using the copy(Path, Path, CopyOption…) method.
  + The copy fails, if the target file exists, unless the REPLACE\_EXISTING option is specified.
  + Directories can be copied. However files inside the directory are not copied
  + So the new directory is empty even when the original directory contains files.
  + When copying a symbolic link, the target of the link is copied.
  + If you want to copy the link itself and not the contents of the link, specify either the NOFOLLOW\_LINKS or REPLACE\_EXISTING option.
  + This method takes a varargs argument.
  + The following StandardCopyOption and LinkOption enums are supported.
    - REPLACE\_EXISTING
      * Performs the copy even when the target file already exists.
      * If the target is a symbolic link, the link itself copied(and not the target of the link).
    - COPY\_ATTRIBUTES
      * Copies the file attributes associated with the file to the target file.
    - NOFOLLOW\_LINKS
      * If the file to be copied id 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.
  + Files.walkFileTree method to support a recursive copy.
* **Moving a File or Directory**
  + You can move a file or directory by using the move(Path, Path, CopyOption…) method.
  + The move fails, if the target file exists, unless the REPLACE\_EXISTING option is specified.
  + Empty directories can be moved.
  + If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory.
  + On UNIX systems, moving a directory within the same partition generally consist of renaming the directory.
  + In that situation move works even when the directory contains files.
  + This method takes a varargs argument.
  + The following StandardCopyOption enums are supported.
    - REPLACE\_EXISTING
      * Performs the move even when the target file already exists.
      * If the target is a symbolic link, the symbolic lick is replaced but what it points to is not affected.
    - AUTOMIC\_MOVE
      * Performs the move as an atomic file operation.
      * If the file system does not support an atomic move, an exception is thrown.
  + The following shows how to use the move method.

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

...

Files.move(source, target, REPLACE\_EXISTING);

* **Managing Metadata (File And File Store Attributes)**
  + The definition of metadata is data about other data.
  + With a file system, the data is contained in its files and directories.
  + And the metadata tracks information 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 permission.
  + 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)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#size(java.nio.file.Path)) | Returns the size of the specified file in bytes. |
| [isDirectory(Path, LinkOption)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#isDirectory(java.nio.file.Path,%20java.nio.file.LinkOption...)) | Returns true if the specified Path locates a file that is a directory. |
| [isRegularFile(Path, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#isRegularFile(java.nio.file.Path,%20java.nio.file.LinkOption...)) | Returns true if the specified Path locates a file that is a regular file. |
| [isSymbolicLink(Path)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#isSymbolicLink(java.nio.file.Path)) | Returns true if the specified Path locates a file that is a symbolic link. |
| [isHidden(Path)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#isHidden(java.nio.file.Path)) | Returns true if the specified Path locates a file that is considered hidden by the file system. |
| [getLastModifiedTime(Path, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#getLastModifiedTime(java.nio.file.Path,%20java.nio.file.LinkOption...)) [setLastModifiedTime(Path, FileTime)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#setLastModifiedTime(java.nio.file.Path,%20java.nio.file.attribute.FileTime)) | Returns or sets the specified file's last modified time. |
| [getOwner(Path, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#getOwner(java.nio.file.Path,%20java.nio.file.LinkOption...)) [setOwner(Path, UserPrincipal)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#setOwner(java.nio.file.Path,%20java.nio.file.attribute.UserPrincipal)) | Returns or sets the owner of the file. |
| [getPosixFilePermissions(Path, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#getPosixFilePermissions(java.nio.file.Path,%20java.nio.file.LinkOption...)) [setPosixFilePermissions(Path, Set<PosixFilePermission>)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#setPosixFilePermissions(java.nio.file.Path,%20java.util.Set)) | Returns or sets a file's POSIX file permissions. |
| [getAttribute(Path, String, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#getAttribute(java.nio.file.Path,%20java.lang.String,%20java.nio.file.LinkOption...)) [setAttribute(Path, String, Object, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#setAttribute(java.nio.file.Path,%20java.lang.String,%20java.lang.Object,%20java.nio.file.LinkOption...)) | Returns or sets the value of a file attribute. |

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

|  |  |
| --- | --- |
| **Method** | **Comment** |
| [readAttributes(Path, String, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#readAttributes(java.nio.file.Path,%20java.lang.String,%20java.nio.file.LinkOption...)) | Reads a file's attributes as a bulk operation. The String parameter identifies the attributes to be read. |
| [readAttributes(Path, Class<A>, LinkOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#readAttributes(java.nio.file.Path,%20java.lang.Class,%20java.nio.file.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. |

* + Different file systems have different notions about which attributes should be tracked.
  + For this reason, related file attributes are grouped together into views.
  + A view maps to a particular file system implementation, such as POSIX or DOS, or to a common functionality, such as file ownership.
  + The supported views are as follows
    - BasicFileAttributeView
      * Provides a view of basic attributes that are required to be supported by all file system implementations.
    - DosFileAttributeView
      * Extends the basic attribute view with the standard four bits supported on file systems that support the DOS attributes.
    - PosixFileAttributeView
      * Extends the basic attribute view with attributes supported on file systems that support the POSIX family of standards, such as UNIX.
      * These attributes include file owner, group owner, and the nine related access permissions.
    - FileOwnerAttributeView
      * Supported by any file system implementation that supports the concept of a file owner.
    - AclFileAttributeView
      * Supports reading or updating a file's Access Control Lists (ACL).
      * The NFSv4 ACL model is supported. Any ACL model, such as the Windows ACL model, that has a well-defined mapping to the NFSv4 model might also be supported.
    - UserDefinedFileAttributeView
      * Enables support of metadata that is user defined.
      * This view can be mapped to any extension mechanisms that a system supports.
      * In the Solaris OS, for example, you can use this view to store the MIME type of a file.
  + A specific file system implementation might support only the basic file attribute view
  + Or it may support several of these file attribute views.
  + A file system implementation might support other attribute views not included in this API.
  + In most instances, you should not have to deal directly with any of the FileAttributeView interfaces.
  + If you do need to work directly with the FileAttributeView, you can access it via the getFileAttributeView(Path, Class<V>, LinkOption...) method.)
  + The readAttributes methods use generics and can be used to read the attributes for any of the file attributes views.
  + **Basic File Attributes**
    - As mentioned previously, to read the basic attributes of a file, you can use one of the Files.readAttributes methods
    - Which reads all the basic attributes in one bulk operation.
    - This is far more efficient than accessing the file system separately to read each individual attribute.
    - The varargs argument currently supports the LinkOption enum, NOFOLLOW\_LINKS.
    - Use this option when you do not want symbolic links to be followed.
    - The set of basic attributes includes three time stamps.
    - creationTime, lastModifiedTime, and lastAccessTime.
    - Any of these time stamps might not be supported in a particular implementation
    - In which case the corresponding accessor method returns an implementation-specific value.
    - When supported, the time stamp is returned as an FileTime object.
    - The following code snippet reads and prints the basic file attributes for a given file and uses the methods in the BasicFileAttributes class.

Path file = ...;

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

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

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

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

System.out.println("isDirectory: " + attr.isDirectory());

System.out.println("isOther: " + attr.isOther());

System.out.println("isRegularFile: " + attr.isRegularFile());

System.out.println("isSymbolicLink: " + attr.isSymbolicLink());

System.out.println("size: " + attr.size());

* + - In addition to the accessor methods shown in this example, there is a *fileKey method* that returns either an object that uniquely identifies the file or null if no file key is available.
  + **Setting Time Stamps**
    - The following code snippet sets the last modified time in milliseconds:

Path file = ...;

BasicFileAttributes attr =

Files.readAttributes(file, BasicFileAttributes.class);

long currentTime = System.currentTimeMillis();

FileTime ft = FileTime.fromMillis(currentTime);

Files.setLastModifiedTime(file, ft);

}

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

Path file = ...;

try {

DosFileAttributes attr =

Files.readAttributes(file, DosFileAttributes.class);

System.out.println("isReadOnly is " + attr.isReadOnly());

System.out.println("isHidden is " + attr.isHidden());

System.out.println("isArchive is " + attr.isArchive());

System.out.println("isSystem is " + attr.isSystem());

} catch (UnsupportedOperationException x) {

System.err.println("DOS file" +

" attributes not supported:" + x);

}

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

Path file = ...;

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

* + **POSIX File Permissions**
    - POSIX is an acronym for Portable Operating System Interface for UNIX and is a set of IEEE and ISO standards designed to ensure interoperability among different flavours of UNIX.
    - If a program conforms to these POSIX standards, it should be easily ported to other POSIX-compliant operating systems.
    - Besides file owner and group owner, POSIX supports nine file permissions:
    - Read, write, and execute permissions for the file owner, members of the same group, and "everyone else."
    - The following code snippet reads the POSIX file attributes for a given file and prints them to standard output.
    - The code uses the methods in the PosixFileAttributes class.

Path file = ...;

PosixFileAttributes attr =

Files.readAttributes(file, PosixFileAttributes.class);

System.out.format("%s %s %s%n",

attr.owner().getName(),

attr.group().getName(),

PosixFilePermissions.toString(attr.permissions()));

* + - The PosixFilePermissions helper class provides several useful methods, as follows:
      * The toString method, used in the previous code snippet, converts the file permissions to a string (for example, rw-r--r--).
      * The fromString method accepts a string representing the file permissions and constructs a Set of file permissions.
      * The asFileAttribute method accepts a Set of file permissions and constructs a file attribute that can be passed to the Path.createFile or Path.createDirectory method.
    - The following code snippet reads the attributes from one file and creates a new file, assigning the attributes from the original file to the new file:

Path sourceFile = ...;

Path newFile = ...;

PosixFileAttributes attrs =

Files.readAttributes(sourceFile, PosixFileAttributes.class);

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(attrs.permissions());

Files.createFile(file, attr);

* + - The asFileAttribute method wraps the permissions as a FileAttribute.
    - The code then attempts to create a new file with those permissions.
    - Note that the umask also applies, so the new file might be more secure than the permissions that were requested.
    - To set a file's permissions to values represented as a hard-coded string, you can use the following code:

Path file = ...;

Set<PosixFilePermission> perms =

PosixFilePermissions.fromString("rw-------");

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(perms);

Files.setPosixFilePermissions(file, perms);

* + **Setting a File or Group Owner**
    - To translate a name into an object you can store as a file owner or a group owner.
    - You can use the UserPrincipalLookupService service.
    - This service looks up a name or group name as a string and returns a UserPrincipal object representing that string.
    - You can obtain the user principal look-up service for the default file system by using the FileSystem.getUserPrincipalLookupService method.
    - The following code snippet shows how to set the file owner by using the setOwner method:

Path file = ...;

UserPrincipal owner = file.GetFileSystem().getUserPrincipalLookupService()

.lookupPrincipalByName("sally");

Files.setOwner(file, owner);

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

Path file = ...;

GroupPrincipal group =

file.getFileSystem().getUserPrincipalLookupService()

.lookupPrincipalByGroupName("green");

Files.getFileAttributeView(file, PosixFileAttributeView.class)

.setGroup(group);

* + **User-Defined File Attributes**
    - If the file attributes supported by your file system implementation aren't sufficient for your needs.
    - You can use the UserDefinedAttributeView to create and track your own file attributes.
    - Some implementations map this concept to features like NTFS Alternative Data Streams and extended attributes on file systems such as ext3 and ZFS.
    - Most implementations impose restrictions on the size of the value, for example, ext3 limits the size to 4 kilobytes.
    - A file's MIME type can be stored as a user-defined attribute by using this code snippet:

Path file = ...;

UserDefinedFileAttributeView view = Files

.getFileAttributeView(file, UserDefinedFileAttributeView.class);

view.write("user.mimetype",

Charset.defaultCharset().encode("text/html");

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

Path file = ...;

UserDefinedFileAttributeView view = Files

.getFileAttributeView(file,UserDefinedFileAttributeView.class);

String name = "user.mimetype";

ByteBuffer buf = ByteBuffer.allocate(view.size(name));

view.read(name, buf);

buf.flip();

String value = Charset.defaultCharset().decode(buf).toString();

* + - 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.

$ 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);

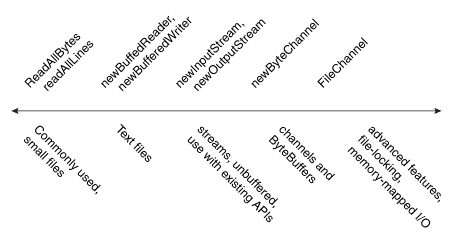
long total = store.getTotalSpace() / 1024;

long used = (store.getTotalSpace() -

store.getUnallocatedSpace()) / 1024;

long avail = store.getUsableSpace() / 1024;

* **Reading, Writing, and Creating Files**
  + This section discusses the details of reading, writing, creating, and opening files.
  + There are wide arrays of file I/O methods to choose from.
  + To help make sense of the API, the following diagram arranges the file I/O methods by complexity.



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

* + **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 behaviour 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 honoured on some file systems, such as NTFS, where large files with data "gaps" can be stored in a more efficient manner where those empty gaps do not consume disk space.
      * SYNC – Keeps the file (both content and metadata) synchronized with the underlying storage device.
      * DSYNC – Keeps the file content synchronized with the underlying storage device.
  + **Commonly Used Methods for Small Files**
  + **Reading All Bytes or Lines from a File**
    - If you have a small-ish file and you would like to read its entire contents in one pass,
    - You can use the readAllBytes(Path) or readAllLines(Path, Charset) method.
    - These methods take care of most of the work for you,
    - Such as opening and closing the stream, but are not intended for handling large files.
    - The following code shows how to use the readAllBytes method:

Path file = ...;

byte[] fileArray;

fileArray = Files.readAllBytes(file);

* + **Writing All Bytes or Lines to a File**
    - You can use one of the write methods to write bytes, or lines, to a file.

[write(Path, byte[], OpenOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#write(java.nio.file.Path,%20byte%5B%5D,%20java.nio.file.OpenOption...))

[write(Path, Iterable< extends CharSequence>, Charset, OpenOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#write(java.nio.file.Path,%20java.lang.Iterable,%20java.nio.charset.Charset,%20java.nio.file.OpenOption...))

* + - The following code snippet shows how to use a write method.

Path file = ...;

byte[] buf = ...;

Files.write(file, buf);

* + **Buffered I/O Methods for Text Files**
    - The java.nio.file package supports channel I/O, which moves data in buffers,
    - Bypassing some of the layers that can bottleneck stream I/O.
  + **Reading a File by Using Buffered Stream I/O**
    - The newBufferedReader(Path, Charset) method opens a file for reading, returning a BufferedReader that can be used to read text from a file in an efficient manner.
    - The following code snippet shows how to use the newBufferedReader method to read from a file.
    - The file is encoded in "US-ASCII."

Charset charset = Charset.forName("US-ASCII");

try (BufferedReader reader = Files.newBufferedReader(file, charset)) {

String line = null;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

* + **Writing a File by Using Buffered Stream I/O**
    - You can use the newBufferedWriter(Path, Charset, OpenOption...) method to write to a file using a BufferedWriter.
    - The following code snippet shows how to create a file encoded in "US-ASCII" using this method:

Charset charset = Charset.forName("US-ASCII");

String s = ...;

try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {

writer.write(s, 0, s.length());

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

* + **Methods for Unbuffered Streams and Interoperable with java.io APIs**
  + **Reading a File by Using Stream I/O**
    - To open a file for reading, you can use the *newInputStream(Path, OpenOption...)* method.
    - This method returns an unbuffered input stream for reading bytes from the file.

Path file = ...;

try (InputStream in = Files.newInputStream(file);

BufferedReader reader =

new BufferedReader(new InputStreamReader(in))) {

String line = null;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException x) {

System.err.println(x);

}

* + **Creating and Writing a File by Using Stream I/O**
    - You can create a file, append to a file, or write to a file by using the newOutputStream(Path, OpenOption...) method.
    - This method opens or creates a file for writing bytes and returns an unbuffered output stream.
    - The method takes an optional OpenOption parameter.
    - If no open options are specified, and the file does not exist, a new file is created.
    - If the file exists, it is truncated.
    - This option is equivalent to invoking the method with the CREATE and TRUNCATE\_EXISTING options.
    - The following code snippet opens a log file.
    - If the file does not exist, it is created.
    - If the file exists, it is opened for appending.

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

Path logfile = ...;

// Convert the string to a

// byte array.

String s = ...;

byte data[] = s.getBytes();

try (OutputStream out = new BufferedOutputStream(

logfile.newOutputStream(CREATE, APPEND))) {

...

out.write(data, 0, data.length);

} catch (IOException x) {

System.err.println(x);

}

* + **Methods for Channels and ByteBuffers**
  + **Reading and Writing Files by Using Channel I/O**
    - While stream I/O reads a character at a time,
    - Channel I/O reads a buffer at a time.
    - The ByteChannel interface provides basic read and write functionality.
    - A SeekableByteChannel is a ByteChannel that has the capability to maintain a position in the channel and to change that position.
    - A SeekableByteChannel also supports truncating the file associated with the channel and querying the file for its size.
    - The capability to move to different points in the file and then read from or write to that location makes random access of a file possible.
    - There are two methods for reading and writing channel I/O.

[newByteChannel(Path, OpenOption...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#newByteChannel(java.nio.file.Path,%20java.nio.file.OpenOption...))

[newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#newByteChannel(java.nio.file.Path,%20java.util.Set,%20java.nio.file.attribute.FileAttribute...))

* + - The newByteChannel methods return an instance of a SeekableByteChannel.
    - With a default file system, you can cast this seekable byte channel to a FileChannel providing access to more advanced features such mapping a region of the file directly into memory for faster access,
    - locking a region of the file so other processes cannot access it,
    - or reading and writing bytes from an absolute position without affecting the channel's current position.
    - Both newByteChannel methods enable you to specify a list of OpenOption options.
    - The same open options used by the newOutputStream methods are supported,
    - in addition to one more option: READ is required because the SeekableByteChannel supports both reading and writing.
    - Specifying READ opens the channel for reading.
    - Specifying WRITE or APPEND opens the channel for writing.
    - If none of these options is specified, the channel is opened for reading.
    - The following code snippet reads a file and prints it to standard output:

// Defaults to READ

try (SeekableByteChannel sbc = Files.newByteChannel(file)) {

ByteBuffer buf = ByteBuffer.allocate(10);

// Read the bytes with the proper encoding for this platform. If

// you skip this step, you might see something that looks like

// Chinese characters when you expect Latin-style characters.

String encoding = System.getProperty("file.encoding");

while (sbc.read(buf) > 0) {

buf.rewind();

System.out.print(Charset.forName(encoding).decode(buf));

buf.flip();

}

} catch (IOException x) {

System.out.println("caught exception: " + x);

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

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

// Create the set of options for appending to the file.

Set<OpenOptions> options = new HashSet<OpenOption>();

options.add(APPEND);

options.add(CREATE);

// Create the custom permissions attribute.

Set<PosixFilePermission> perms =

PosixFilePermissions.fromString("rw-r------");

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(perms);

// Convert the string to a ByteBuffer.

String s = ...;

byte data[] = s.getBytes();

ByteBuffer bb = ByteBuffer.wrap(data);

try (SeekableByteChannel sbc = Files.newByteChannel(file, options, attr)) {

sbc.write(bb);

} catch (IOException x) {

System.out.println("exception thrown: " + x);

}

* + **Methods for Creating Regular and Temporary Files**
  + **Creating Files**
    - You can create an empty file with an initial set of attributes by using the createFile(Path, FileAttribute<?>) method.
    - For example, if, at the time of creation, you want a file to have a particular set of file permissions, use the createFile method to do so.
    - If you do not specify any attributes, the file is created with default attributes.
    - If the file already exists, createFile throws an exception.
    - In a single atomic operation, the createFile method checks for the existence of the file and creates that file with the specified attributes,
    - Which makes the process more secure against malicious code.
    - The following code snippet creates a file with default attributes:

Path file = ...;

try {

// Create the empty file with default permissions, etc.

Files.createFile(file);

} catch (FileAlreadyExistsException x) {

System.err.format("file named %s" +

" already exists%n", file);

} catch (IOException x) {

// Some other sort of failure, such as permissions.

System.err.format("createFile error: %s%n", x);

}

* + **Creating Temporary Files**
    - You can create a temporary file using one of the following createTempFile methods:

[createTempFile(Path, String, String, FileAttribute<?>)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#createTempFile%28java.nio.file.Path,%20java.lang.String,%20java.lang.String,%20java.nio.file.attribute.FileAttribute...%29)

[createTempFile(String, String, FileAttribute<?>)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#createTempFile%28java.lang.String,%20java.lang.String,%20java.nio.file.attribute.FileAttribute...%29)

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

try {

Path tempFile = Files.createTempFile(null, ".myapp");

System.out.format("The temporary file" +

" has been created: %s%n", tempFile)

;

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

* + - The result of running this file would be something like the following:
      * The temporary file has been created: /tmp/509668702974537184.myapp
    - The specific format of the temporary file name is platform specific.
* **Random Access Files**
  + Random access files permit nonsequential, or random, access to a file's contents.
  + To access a file randomly, you open the file, seek a particular location, and read from or write to that file.
  + This functionality is possible with the SeekableByteChannel interface.
  + The SeekableByteChannel interface extends channel I/O with the notion of a current position.
  + Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location.
  + The API consists of a few, easy to use, methods:
    - position – Returns the channel's current position
    - position(long) – Sets the channel's position
    - read(ByteBuffer) – Reads bytes into the buffer from the channel
    - write(ByteBuffer) – Writes bytes from the buffer to the channel
    - truncate(long) – Truncates the file (or other entity) connected to the channel
  + Reading and Writing Files With Channel I/O shows that the Path.newByteChannel methods return an instance of a SeekableByteChannel.
  + On the default file system, you can use that channel as is, or you can cast it to a FileChannel giving you access to more advanced features,
  + Such as mapping a region of the file directly into memory for faster access, locking a region of the file,
  + Or reading and writing bytes from an absolute location without affecting the channel's current position.
  + The following code snippet opens a file for both reading and writing by using one of the newByteChannel methods.
  + The SeekableByteChannel that is returned is cast to a FileChannel.
  + Then, 12 bytes are read from the beginning of the file, and the string "I was here!" is written at that location.
  + The current position in the file is moved to the end, and the 12 bytes from the beginning are appended.
  + Finally, the string, "I was here!" is appended, and the channel on the file is closed.

String s = "I was here!\n";

byte data[] = s.getBytes();

ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {

// Read the first 12

// bytes of the file.

int nread;

do {

nread = fc.read(copy);

} while (nread != -1 && copy.hasRemaining());

// Write "I was here!" at the beginning of the file.

fc.position(0);

while (out.hasRemaining())

fc.write(out);

out.rewind();

// Move to the end of the file. Copy the first 12 bytes to

// the end of the file. Then write "I was here!" again.

long length = fc.size();

fc.position(length-1);

copy.flip();

while (copy.hasRemaining())

fc.write(copy);

while (out.hasRemaining())

fc.write(out);

} catch (IOException x) {

System.out.println("I/O Exception: " + x);

}

* **Creating and Reading Directories**
  + Some of the methods previously discussed, such as delete, work on files, links and directories.
  + **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.

Path dir = ...;

Files.createDirectory(path);

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

Set<PosixFilePermission> perms =

PosixFilePermissions.fromString("rwxr-x---");

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(perms);

Files.createDirectory(file, attr);

* + - To create a directory several levels deep when one or more of the parent directories might not yet exist,
    - You can use the convenience method, createDirectories(Path, FileAttribute<?>).
    - As with the createDirectory(Path, FileAttribute<?>) method, you can specify an optional set of initial file attributes.
    - The following code snippet uses default attributes:

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

* + - The directories are created, as needed, from the top down.
    - In the foo/bar/test example, if the foo directory does not exist, it is created.
    - Next, the bar directory is created, if needed, and, finally, the test directory is created.
    - It is possible for this method to fail after creating some, but not all, of the parent directories.
  + **Creating a Temporary Directory**
    - You can create a temporary directory using one of createTempDirectory methods:

[createTempDirectory(Path, String, FileAttribute<?>...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#createTempDirectory%28java.nio.file.Path,%20java.lang.String,%20java.nio.file.attribute.FileAttribute...%29)

[createTempDirectory(String, FileAttribute<?>...)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#createTempDirectory%28java.lang.String,%20java.nio.file.attribute.FileAttribute...%29)

* + - 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.
    - The returned DirectoryStream is a stream.
    - If you are not using a try-with-resources statement, don't forget to close the stream in the finally block.
    - The try-with-resources statement takes care of this for you.
    - The following code snippet shows how to print the contents of a directory:

Path dir = ...;

try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {

for (Path file: stream) {

System.out.println(file.getFileName());

}

} catch (IOException | DirectoryIteratorException x) {

// IOException can never be thrown by the iteration.

// In this snippet, it can only be thrown by newDirectoryStream.

System.err.println(x);

}

* + - The Path objects returned by the iterator are the names of the entries resolved against the directory.
    - So, if you are listing the contents of the /tmp directory, the entries are returned with the form /tmp/a, /tmp/b, and so on.
    - This method returns the entire contents of a directory: files, links, subdirectories, and hidden files.
    - If you want to be more selective about the contents that are retrieved, you can use one of the other newDirectoryStream methods, as described later in this page.
    - Note that if there is an exception during directory iteration then DirectoryIteratorException is thrown with the IOException as the cause.
    - Iterator methods cannot throw exception exceptions.
  + **Filtering a Directory Listing By Using Globbing**
    - If you want to fetch only files and subdirectories where each name matches a particular pattern,
    - You can do so by using the newDirectoryStream(Path, String) method, which provides a built-in glob filter.
    - For example, the following code snippet lists files relating to Java: .class, .java, and .jar files.

Path dir = ...;

try (DirectoryStream<Path> stream =

Files.newDirectoryStream(dir, "\*.{java,class,jar}")) {

for (Path entry: stream) {

System.out.println(entry.getFileName());

}

} catch (IOException x) {

// IOException can never be thrown by the iteration.

// In this snippet, it can // only be thrown by newDirectoryStream.

System.err.println(x);

}

* + **Writing Your Own Directory Filter**
    - Perhaps you want to filter the contents of a directory based on some condition other than pattern matching.
    - You can create your own filter by implementing the DirectoryStream.Filter<T> interface.
    - This interface consists of one method, accept, which determines whether a file fulfills the search requirement.
    - For example, the following code snippet implements a filter that retrieves only directories:

DirectoryStream.Filter<Path> filter =

newDirectoryStream.Filter<Path>() {

public boolean accept(Path file) throws IOException {

try {

return (Files.isDirectory(path));

} catch (IOException x) {

// Failed to determine if it's a directory.

System.err.println(x);

return false;

}

}

};

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

Path dir = ...;

try (DirectoryStream<Path>

stream = Files.newDirectoryStream(dir, filter)) {

for (Path entry: stream) {

System.out.println(entry.getFileName());

}

} catch (IOException x) {

System.err.println(x);

}

* + - This method is used to filter a single directory only.
    - However, if you want to find all the subdirectories in a file tree, you would use the mechanism for Walking the File Tree.
* **Links, Symbolic or Otherwise**
  + As mentioned previously, the java.nio.file package, and the Path class in particular, is "link aware."
  + Every Path method either detects what to do when a symbolic link is encountered,
  + Or it provides an option enabling you to configure the behaviour when a symbolic link is encountered.
  + The discussion so far has been about symbolic or soft links, but some file systems also support hard links.
  + Hard links are more restrictive than symbolic links, as follows:
    - The target of the link must exist.
    - Hard links are generally not allowed on directories.
    - Hard links are not allowed to cross partitions or volumes. Therefore, they cannot exist across file systems.
    - A hard link looks, and behaves, like a regular file, so they can be hard to find.
    - A hard link is, for all intents and purposes, the same entity as the original file.
    - They have the same file permissions, time stamps, and so on. All attributes are identical.
  + Because of these restrictions, hard links are not used as often as symbolic links, but the Path methods work seamlessly with hard links.
  + **Creating a Symbolic Link**
    - If your file system supports it, you can create a symbolic link by using the createSymbolicLink(Path, Path, FileAttribute<?>) method.
    - The second Path argument represents the target file or directory and might or might not exist.
    - The following code snippet creates a symbolic link with default permissions:

Path newLink = ...;

Path target = ...;

try {

Files.createSymbolicLink(newLink, target);

} catch (IOException x) {

System.err.println(x);

} catch (UnsupportedOperationException x) {

// Some file systems do not support symbolic links.

System.err.println(x);

}

* + - The FileAttributes vararg enables you to specify initial file attributes that are set atomically when the link is created.
    - However, this argument is intended for future use and is not currently implemented.
  + **Creating a Hard Link**
    - You can create a hard (or regular) link to an existing file by using the createLink(Path, Path) method.
    - The second Path argument locates the existing file, and it must exist or a NoSuchFileException is thrown.
    - The following code snippet shows how to create a link:

Path newLink = ...;

Path existingFile = ...;

try {

Files.createLink(newLink, existingFile);

} catch (IOException x) {

System.err.println(x);

} catch (UnsupportedOperationException x) {

// Some file systems do not

// support adding an existing

// file to a directory.

System.err.println(x);

}

* + **Detecting a Symbolic Link**
    - To determine whether a Path instance is a symbolic link, you can use the isSymbolicLink(Path) method.

Path file = ...;

boolean isSymbolicLink =

Files.isSymbolicLink(file);

* + **Finding the Target of a Link**
    - You can obtain the target of a symbolic link by using the readSymbolicLink(Path) method, as follows:

Path link = ...;

try {

System.out.format("Target of link" +

" '%s' is '%s'%n", link,

Files.readSymbolicLink(link));

} catch (IOException x) {

System.err.println(x);

}

* + - If the Path is not a symbolic link, this method throws a NotLinkException.
* **Walking the File Tree**
  + Do you need to create an application that will recursively visit all the files in a file tree?
  + Perhaps you need to delete every .class file in a tree, or find every file that hasn't been accessed in the last year.
  + You can do so with the FileVisitor interface.
  + **The FileVisitor Interface**
    - To walk a file tree, you first need to implement a FileVisitor.
    - A FileVisitor specifies the required behaviour at key points in the traversal process:
    - when a file is visited, before a directory is accessed, after a directory is accessed, or when a failure occurs.
    - The interface has four methods that correspond to these situations:
      * preVisitDirectory – Invoked before a directory's entries are visited.
      * postVisitDirectory – Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
      * visitFile – Invoked on the file being visited. The file's BasicFileAttributes is passed to the method, or you can use the file attributes package to read a specific set of attributes. For example, you can choose to read the file's DosFileAttributeView to determine if the file has the "hidden" bit set.
      * visitFileFailed – Invoked when the file cannot be accessed. The specific exception is passed to the method. You can choose whether to throw the exception, print it to the console or a log file, and so on.
    - If you don't need to implement all four of the FileVisitor methods, instead of implementing the FileVisitor interface, you can extend the SimpleFileVisitor class.
    - This class, which implements the FileVisitor interface, visits all files in a tree and throws an IOError when an error is encountered.
    - You can extend this class and override only the methods that you require.

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

public static class PrintFiles

extends SimpleFileVisitor<Path> {

// Print information about

// each type of file.

@Override

**public FileVisitResult visitFile(Path file,**

**BasicFileAttributes attr)** {

if (attr.isSymbolicLink()) {

System.out.format("Symbolic link: %s ", file);

} else if (attr.isRegularFile()) {

System.out.format("Regular file: %s ", file);

} else {

System.out.format("Other: %s ", file);

}

System.out.println("(" + attr.size() + "bytes)");

return CONTINUE;

}

// Print each directory visited.

@Override

**public FileVisitResult postVisitDirectory(Path dir,**

**IOException exc)** {

System.out.format("Directory: %s%n", dir);

return CONTINUE;

}

// If there is some error accessing

// the file, let the user know.

// If you don't override this method

// and an error occurs, an IOException

// is thrown.

@Override

**public FileVisitResult visitFileFailed(Path file,**

**IOException exc)** {

System.err.println(exc);

return CONTINUE;

}

}

* + **Kickstarting the Process**
    - Once you have implemented your FileVisitor, how do you initiate the file walk?
    - There are two walkFileTree methods in the Files class.

[walkFileTree(Path, FileVisitor)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#walkFileTree%28java.nio.file.Path,%20java.nio.file.FileVisitor%29)

[walkFileTree(Path, Set<FileVisitOption>, int, FileVisitor)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#walkFileTree%28java.nio.file.Path,%20java.util.Set,%20int,%20java.nio.file.FileVisitor%29)

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

Path startingDir = ...;

PrintFiles pf = new PrintFiles();

Files.walkFileTree(startingDir, pf);

* + - The second walkFileTree method enables you to additionally specify a limit on the number of levels visited and a set of FileVisitOption enums.
    - If you want to ensure that this method walks the entire file tree, you can specify Integer.MAX\_VALUE for the maximum depth argument.
    - You can specify the FileVisitOption enum, FOLLOW\_LINKS, which indicates that symbolic links should be followed.
    - This code snippet shows how the four-argument method can be invoked:

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

Path startingDir = ...;

EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW\_LINKS);

Finder finder = new Finder(pattern);

Files.walkFileTree(startingDir, opts, Integer.MAX\_VALUE, finder);

* + **Considerations When Creating a FileVisitor**
    - A file tree is walked depth first, but you cannot make any assumptions about the iteration order that subdirectories are visited.
    - If your program will be changing the file system, you need to carefully consider how you implement your FileVisitor.
    - For example, if you are writing a recursive delete, you first delete the files in a directory before deleting the directory itself.
    - In this case, you delete the directory in postVisitDirectory.
    - If you are writing a recursive copy, you create the new directory in preVisitDirectory before attempting to copy the files to it (in visitFiles).
    - If you want to preserve the attributes of the source directory (similar to the UNIX cp -p command), you need to do that after the files have been copied, in postVisitDirectory.
    - 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.
    - You need to decide whether you want symbolic links to be followed.
    - If you are deleting files, for example, following symbolic links might not be advisable.
    - If you are copying a file tree, you might want to allow it. By default, walkFileTree does not follow symbolic links.
    - The visitFile method is invoked for files.
    - If you have specified the FOLLOW\_LINKS option and your file tree has a circular link to a parent directory,
    - The looping directory is reported in the visitFileFailed method with the FileSystemLoopException.
    - The following code snippet shows how to catch a circular link and is from the Copy example:

@Override

public FileVisitResult

visitFileFailed(Path file,

IOException exc) {

if (exc instanceof FileSystemLoopException) {

System.err.println("cycle detected: " + file);

} else {

System.err.format("Unable to copy:" + " %s: %s%n", file, exc);

}

return CONTINUE;

}

* + - This case can occur only when the program is following symbolic links.
  + **Controlling the Flow**
    - Perhaps you want to walk the file tree looking for a particular directory and, when found, you want the process to terminate.
    - Perhaps you want to skip specific directories.
    - The FileVisitor methods return a FileVisitResult value.
    - You can abort the file walking process or control whether a directory is visited by the values you return in the FileVisitor methods:
      * CONTINUE – Indicates that the file walking should continue. If the preVisitDirectory method returns CONTINUE, the directory is visited.
      * TERMINATE – Immediately aborts the file walking. No further file walking methods are invoked after this value is returned.
      * SKIP\_SUBTREE – When preVisitDirectory returns this value, the specified directory and its subdirectories are skipped. This branch is "pruned out" of the tree.
      * SKIP\_SIBLINGS – When preVisitDirectory returns this value, the specified directory is not visited, postVisitDirectory is not invoked, and no further unvisited siblings are visited. If returned from the postVisitDirectory method, no further siblings are visited. Essentially, nothing further happens in the specified directory.
    - In this code snippet, any directory named SCCS is skipped:

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

public FileVisitResult

preVisitDirectory(Path dir,

BasicFileAttributes attrs) {

(if (dir.getFileName().toString().equals("SCCS")) {

return SKIP\_SUBTREE;

}

return CONTINUE;

}

* + - In this code snippet, as soon as a particular file is located, the file name is printed to standard output, and the file walking terminates:

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

// The file we are looking for.

Path lookingFor = ...;

public FileVisitResult

visitFile(Path file,

BasicFileAttributes attr) {

if (file.getFileName().equals(lookingFor)) {

System.out.println("Located file: " + file);

return TERMINATE;

}

return CONTINUE;

}

* **Finding Files**
  + If you have ever used a shell script, you have most likely used pattern matching to locate files.
  + In fact, you have probably used it extensively.
  + If you haven't used it, pattern matching uses special characters to create a pattern and then file names can be compared against that pattern.
  + For example, in most shell scripts, the asterisk, \*, matches any number of characters.
  + For example, the following command lists all the files in the current directory that end in .html

% ls \*.html

* + The java.nio.file package provides programmatic support for this useful feature.
  + Each file system implementation provides a PathMatcher.
  + You can retrieve a file system's PathMatcher by using the getPathMatcher(String) method in the FileSystem class.
  + The following code snippet fetches the path matcher for the default file system:

String pattern = ...;

PathMatcher matcher =

FileSystems.getDefault().getPathMatcher("glob:" + pattern);

* + The string argument passed to getPathMatcher specifies the syntax flavour and the pattern to be matched.
  + This example specifies glob syntax.
  + Glob syntax is easy to use and flexible but, if you prefer, you can also use regular expressions, or regex, syntax.
  + For further information about regex, see the Regular Expressions lesson.
  + Some file system implementations might support other syntaxes.
  + If you want to use some other form of string-based pattern matching, you can create your own PathMatcher class.
  + The examples in this page use glob syntax.
  + Once you have created your PathMatcher instance, you are ready to match files against it.
  + The PathMatcher interface has a single method, matches, that takes a Path argument and returns a boolean:
  + It either matches the pattern, or it does not.
  + The following code snippet looks for files that end in .java or .class and prints those files to standard output:

PathMatcher matcher =

FileSystems.getDefault().getPathMatcher("glob:\*.{java,class}");

Path filename = ...;

if (matcher.matches(filename)) {

System.out.println(filename);

}

* + **Recursive Pattern Matching**
    - Searching for files that match a particular pattern goes hand-in-hand with walking a file tree.
    - How many times do you know a file is somewhere on the file system, but where?
    - Or perhaps you need to find all files in a file tree that have a particular file extension.
    - The Find example does precisely that.
    - Find is similar to the UNIX find utility, but has pared down functionally.
    - You can extend this example to include other functionality.
    - For example, the find utility supports the -prune flag to exclude an entire subtree from the search.
    - You could implement that functionality by returning SKIP\_SUBTREE in the preVisitDirectory method.
    - To implement the -L option, which follows symbolic links,
    - you could use the four-argument walkFileTree method and pass in the FOLLOW\_LINKS enum (but make sure that you test for circular links in the visitFile method).
    - To run the Find application, use the following format:

% java Find <path> -name "<glob\_pattern>"

* + - The pattern is placed inside quotation marks so any wildcards are not interpreted by the shell. For example:

% java Find . -name "\*.html"

* + - Here is the source code for the Find example:

/\*\*

\* Sample code that finds files that match the specified glob pattern.

\* For more information on what constitutes a glob pattern, see

\* http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob

\*

\* The file or directories that match the pattern are printed to

\* standard out. The number of matches is also printed.

\*

\* When executing this application, you must put the glob pattern

\* in quotes, so the shell will not expand any wild cards:

\* java Find . -name "\*.java"

\*/

import java.io.\*;

import java.nio.file.\*;

import java.nio.file.attribute.\*;

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

import static java.nio.file.FileVisitOption.\*;

import java.util.\*;

public class Find {

public static class Finder

extends SimpleFileVisitor<Path> {

private final PathMatcher matcher;

private int numMatches = 0;

Finder(String pattern) {

matcher =

FileSystems.getDefault()

.getPathMatcher("glob:" + pattern);

}

// Compares the glob pattern against

// the file or directory name.

void find(Path file) {

Path name = file.getFileName();

if (name != null && matcher.matches(name)) {

numMatches++;

System.out.println(file);

}

}

// Prints the total number of

// matches to standard out.

void done() {

System.out.println("Matched: "

+ numMatches);

}

// Invoke the pattern matching

// method on each file.

@Override

public FileVisitResult

visitFile(Path file,

BasicFileAttributes attrs) {

find(file);

return CONTINUE;

}

// Invoke the pattern matching

// method on each directory.

@Override

public FileVisitResult

preVisitDirectory(Path dir,

BasicFileAttributes attrs) {

find(dir);

return CONTINUE;

}

@Override

public FileVisitResult

visitFileFailed(Path file,

IOException exc) {

System.err.println(exc);

return CONTINUE;

}

}

static void usage() {

System.err.println("java Find <path>" +

" -name \"<glob\_pattern>\"");

System.exit(-1);

}

public static void main(String[] args)

throws IOException {

if (args.length < 3

|| !args[1].equals("-name"))

usage();

Path startingDir = Paths.get(args[0]);

String pattern = args[2];

Finder finder = new Finder(pattern);

Files.walkFileTree(startingDir, finder);

finder.done();

}

}

* **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.
  + 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.
  + **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:
    - 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.
    - Process the pending events for the key. You fetch the List of WatchEventsfrom the pollEvents method.
    - 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.
    - 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.
    - After the events for the key have been processed, you need to put the key back into a ready state by invoking reset. If this method returns false, the key is no longer valid and the loop can exit. This step is very important. If you fail to invoke reset, this key will not receive any further events.
    - A watch key has a state. At any given time, its state might be one of the following:
      * Ready indicates that the key is ready to accept events. When first created, a key is in the ready state.
      * Signaled indicates that one or more events are queued. Once the key has been signaled, it is no longer in the ready state until the reset method is invoked.
      * Invalid indicates that the key is no longer active. This state happens when one of the following events occurs:
        + The process explicitly cancels the key by using the cancel method.
        + The directory becomes inaccessible.
        + The watch service is closed.
    - Here is an example of an event processing loop.
    - It is taken from the Email example, which watches a directory, waiting for new files to appear.
    - When a new file becomes available, it is examined to determine if it is a text/plain file by using the probeContentType(Path) method.
    - The intention is that text/plain files will be emailed to an alias, but that implementation detail is left to the reader.
    - The methods specific to the watch service API are shown in bold:

for (;;) {

// wait for key to be signaled

WatchKey key;

try {

**key = watcher.take()**;

} catch (InterruptedException x) {

return;

}

for (WatchEvent<?> event: **key.pollEvents()**) {

WatchEvent.Kind<?> **kind = event.kind()**;

// This key is registered only

// for ENTRY\_CREATE events,

// but an OVERFLOW event can

// occur regardless if events

// are lost or discarded.

**if (kind == OVERFLOW)** {

continue;

}

// The filename is the

// context of the event.

**WatchEvent<Path> ev = (WatchEvent<Path>)event;**

**Path filename = ev.context();**

// Verify that the new

// file is a text file.

try {

// Resolve the filename against the directory.

// If the filename is "test" and the directory is "foo",

// the resolved name is "test/foo".

Path child = dir.resolve(filename);

if (!Files.probeContentType(child).equals("text/plain")) {

System.err.format("New file '%s'" +

" is not a plain text file.%n", filename);

continue;

}

} catch (IOException x) {

System.err.println(x);

continue;

}

// Email the file to the

// specified email alias.

System.out.format("Emailing file %s%n", filename);

//Details left to reader....

}

// Reset the key -- this step is critical if you want to

// receive further watch events. If the key is no longer valid,

// the directory is inaccessible so exit the loop.

boolean **valid = key.reset()**;

if (!valid) {

break;

}

}

* + **Retrieving the File Name**
    - The file name is retrieved from the event context.
    - The Email example retrieves the file name with this code:

WatchEvent<Path> ev = (WatchEvent<Path>)event;

Path filename = ev.context();

* + - When you compile the Email example, it generates the following error:

Note: Email.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

* + - This error is a result of the line of code that casts the WatchEvent<T> to a WatchEvent<Path>.
    - The WatchDir example avoids this error by creating a utility cast method that suppresses the unchecked warning, as follows:

@SuppressWarnings("unchecked")

static <T> WatchEvent<T> cast(WatchEvent<?> event) {

return (WatchEvent<Path>)event;

}

* + **When to Use and Not Use This API**
    - The Watch Service API is designed for applications that need to be notified about file change events.
    - It is well suited for any application, like an editor or IDE, that potentially has many open files and needs to ensure that the files are synchronized with the file system.
    - It is also well suited for an application server that watches a directory, perhaps waiting for .jsp or .jar files to drop, in order to deploy them.
    - This API is not designed for indexing a hard drive.
    - Most file system implementations have native support for file change notification.
    - The Watch Service API takes advantage of this support where available.
    - However, when a file system does not support this mechanism, the Watch Service will poll the file system, waiting for events.
* **Other Useful Methods**
  + **Determining MIME Type**
    - To determine the MIME type of a file, you might find the probeContentType(Path) method useful. For example:

try {

String type = Files.probeContentType(filename);

if (type == null) {

System.err.format("'%s' has an" + " unknown filetype.%n", filename);

} else if (!type.equals("text/plain") {

System.err.format("'%s' is not" + " a plain text file.%n", filename);

continue;

}

} catch (IOException x) {

System.err.println(x);

}

* + - Note that probeContentType returns null if the content type cannot be determined.
    - The implementation of this method is highly platform specific and is not infallible.
    - The content type is determind by the platform's default file type detector.
    - For example, if the detector determines a file's content type to be application/x-java based on the .class extension, it might be fooled.
    - You can provide a custom FileTypeDetector if the default is not sufficient for your needs.
    - The Email example uses the probeContentType method.
  + **Default File System**
    - To retrieve the default file system, use the getDefault method.
    - Typically, this FileSystems method (note the plural) is chained to one of the FileSystem methods (note the singular), as follows:

PathMatcher matcher =

FileSystems.getDefault().getPathMatcher("glob:\*.\*");

* + **Path String Separator**
    - The path separator for POSIX file systems is the forward slash, /, and for Microsoft Windows is the backslash, \.
    - Other file systems might use other delimiters.
    - To retrieve the Path separator for the default file system, you can use one of the following approaches:
    - The getSeparator method is also used to retrieve the path separator for any available file system.

String separator = File.separator;

String separator = FileSystems.getDefault().getSeparator();

* + **File System's File Stores**
    - A file system has one or more file stores to hold its files and directories.
    - The file store represents the underlying storage device.
    - In UNIX operating systems, each mounted file system is represented by a file store.
    - In Microsoft Windows, each volume is represented by a file store: C:, D:, and so on.
    - To retrieve a list of all the file stores for the file system, you can use the getFileStores method.
    - This method returns an Iterable,
    - Which allows you to use the enhanced for statement to iterate over all the root directories.

for (FileStore store: FileSystems.getDefault().getFileStores()) {

...

}

* + - If you want to retrive the file store where a particular file is located, use the getFileStore method in the Files class, as follows:

Path file = ...;

FileStore store= Files.getFileStore(file);

* **Legacy File I/O Code**
  + **Interoperability With Legacy Code**
    - Prior to the Java SE 7 release, the java.io.File class was the mechanism used for file I/O, but it had several drawbacks.
      * Many methods didn't throw exceptions when they failed, so it was impossible to obtain a useful error message.
      * For example, if a file deletion failed, the program would receive a "delete fail" but wouldn't know if it was because the file didn't exist, the user didn't have permissions, or there was some other problem.
      * The rename method didn't work consistently across platforms.
      * There was no real support for symbolic links.
      * More support for metadata was desired, such as file permissions, file owner, and other security attributes.
      * Accessing file metadata was inefficient.
      * Many of the File methods didn't scale. Requesting a large directory listing over a server could result in a hang.
      * Large directories could also cause memory resource problems, resulting in a denial of service.
      * It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.
    - Perhaps you have legacy code that uses java.io.File and would like to take advantage of the java.nio.file.Path functionality with minimal impact to your code.
    - The java.io.File class provides the toPath method, which converts an old style File instance to a java.nio.file.Path instance, as follows:

Path input = file.toPath();

* + - You can then take advantage of the rich feature set available to the Path class.
    - For example, assume you had some code that deleted a file:

file.delete();

* + - You could modify this code to use the Files.delete method, as follows:

Path fp = file.toPath();

Files.delete(fp);

* + - Conversely, the Path.toFile method constructs a java.io.File object for a Path object.
  + **Mapping java.io.File Functionality to java.nio.file**

|  |  |  |
| --- | --- | --- |
| * **java.io.File Functionality** | **java.nio.file Functionality** | **Tutorial Coverage** |
| java.io.File | java.nio.file.Path | [The Path Class](http://docs.oracle.com/javase/tutorial/essential/io/pathClass.html) |
| java.io.RandomAccessFile | The SeekableByteChannel functionality. | [Random Access Files](http://docs.oracle.com/javase/tutorial/essential/io/rafs.html) |
| File.canRead, canWrite, canExecute | Files.isReadable, Files.isWritable, and Files.isExecutable. On UNIX file systems, the [Managing Metadata (File and File Store Attributes)](http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) package is used to check the nine file permissions. | [Checking a File or Directory](http://docs.oracle.com/javase/tutorial/essential/io/check.html) [Managing Metadata](http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) |
| File.isDirectory(), File.isFile(), and File.length() | Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...), and Files.size(Path) | [Managing Metadata](http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) |
| File.lastModified() and File.setLastModified(long) | Files.getLastModifiedTime(Path, LinkOption...) and Files.setLastMOdifiedTime(Path, FileTime) | [Managing Metadata](http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) |
| 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](http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) |
| new File(parent, "newfile") | parent.resolve("newfile") | [Path Operations](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html) |
| File.renameTo | Files.move | [Moving a File or Directory](http://docs.oracle.com/javase/tutorial/essential/io/move.html) |
| File.delete | Files.delete | [Deleting a File or Directory](http://docs.oracle.com/javase/tutorial/essential/io/delete.html) |
| File.createNewFile | Files.createFile | [Creating Files](http://docs.oracle.com/javase/tutorial/essential/io/file.html#createFile) |
| File.deleteOnExit | Replaced by the DELETE\_ON\_CLOSE option specified in the createFile method. | [Creating Files](http://docs.oracle.com/javase/tutorial/essential/io/file.html#createFile) |
| File.createTempFile | Files.createTempFile(Path, String, FileAttributes<?>), Files.createTempFile(Path, String, String, FileAttributes<?>) | [Creating Files](http://docs.oracle.com/javase/tutorial/essential/io/file.html#createFile) [Creating and Writing a File by Using Stream I/O](http://docs.oracle.com/javase/tutorial/essential/io/file.html#createStream) [Reading and Writing Files by Using Channel I/O](http://docs.oracle.com/javase/tutorial/essential/io/file.html#channelio) |
| File.exists | Files.exists and Files.notExists | [Verifying the Existence of a File or Directory](http://docs.oracle.com/javase/tutorial/essential/io/check.html) |
| File.compareTo and equals | Path.compareTo and equals | [Comparing Two Paths](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#compare) |
| File.getAbsolutePath and getAbsoluteFile | Path.toAbsolutePath | [Converting a Path](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#convert) |
| File.getCanonicalPath and getCanonicalFile | Path.toRealPath or normalize | [Converting a Path (toRealPath)](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#convert) [Removing Redundancies From a Path (normalize)](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#normal) |
| File.toURI | Path.toURI | [Converting a Path](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#convert) |
| File.isHidden | Files.isHidden | [Retrieving Information About the Path](http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#info) |
| File.list and listFiles | Path.newDirectoryStream | [Listing a Directory's Contents](http://docs.oracle.com/javase/tutorial/essential/io/dirs.html#listdir) |
| File.mkdir and mkdirs | Path.createDirectory | [Creating a Directory](http://docs.oracle.com/javase/tutorial/essential/io/dirs.html#create) |
| File.listRoots | FileSystem.getRootDirectories | [Listing a File System's Root Directories](http://docs.oracle.com/javase/tutorial/essential/io/dirs.html#listall) |
| File.getTotalSpace, File.getFreeSpace, File.getUsableSpace | FileStore.getTotalSpace, FileStore.getUnallocatedSpace, FileStore.getUsableSpace, FileStore.getTotalSpace | [File Store Attributes](http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#store) |