Capitolul 14 - IO
Referencing Files and Directories

TABLE 14.1 File system symbols

Symbol	Description
.	A reference to the current directory
..	A reference to the parent of the current directory

> A *symbolic link* is a special file within a file system that serves as a reference or pointer to another file or directory.

Creating a *File* or *Path*

> Remember, a `File` or `Path` can represent a file or a directory.

Creating a *File*

```
File zooFile1 = new File("/home/tiger/data/stripes.txt");
File zooFile2 = new File("/home/tiger", "data/stripes.txt");

File parent = new File("/home/tiger");
File zooFile3 = new File(parent, "data/stripes.txt");

System.out.println(zooFile1.exists());
```

All three create a `File` object that points to the same location on disk.

Creating a *Path*

Since `Path` is an interface, we can't create an instance directly.

```
public static Path of(String first, String… more)
```

```
Path zooPath1 = Path.of("/home/tiger/data/stripes.txt");
Path zooPath2 = Path.of("/home", "tiger", "data", "stripes.txt");
```

```
System.out.println(Files.exists(zooPath1));
```

Other Ways to Create a Path

An older way to create a `Path` is to use the `Paths` helper class.

```
Path zooPath1 = Paths.get("/home", "tiger", "data",
"stripes.txt");
```

Behind the scenes, `Path.of()` actually uses the `FileSystems` helper class to create a `Path`, which means you can call it directly:

```
Path zooPath2 =
FileSystems.getDefault().getPath("/home/tiger/data/stripes.txt");
```

Finally, a `Path` can also be created using a uniform resource identifier (URI).

```
Path zooPath3 = Path.of(URI.create("https://www.selikoff.net"));
```

Switching Between *File* and *Path*

Since `File` and `Path` both reference locations on disk, it is helpful to be able to convert between them.

```
File file = new File("rabbit");
Path newPath = file.toPath();
File backToFile = newPath.toFile();
```

Operating on *File* and *Path*

Using Shared Functionality

TABLE 14.2 Common File and Path operations

Description	I/O File instance method	NIO.2 Path instance method
Gets name of file/directory	getName()	getFileName()
Retrieves parent directory or null if there is none	getParent()	getParent()
Checks if file/directory is absolute path	isAbsolute()	isAbsolute()
Retrieves absolute path of file/directory	getAbsolutePath()	toAbsolutePath()

TABLE 14.3 Common File and Files operations

Description	I/O File instance method	NIO.2 Files static method
Deletes file/directory	delete()	deleteIfExists(Path p) throws IOException
Checks if file/directory exists	exists()	exists(Path p, LinkOption... o)
Checks if resource is directory	isDirectory()	isDirectory(Path p, LinkOption... o)
Checks if resource is file	isFile()	isRegularFile(Path p, LinkOption... o)
Returns the time the file was last modified	lastModified()	getLastModifiedTime(Path p,LinkOption... o) throwsIOException
Retrieves number of bytes in file	length()	size(Path p) throws IOException
Lists contents of directory	listFiles()	list(Path p) throws IOException
Creates directory	mkdir()	createDirectory(Path p,FileAttribute... a) throws IOException
Creates directory including any nonexistent parent directories	mkdirs()	createDirectories(Path p,FileAttribute... a) throws IOException
Renames file/directory denoted	renameTo(File dest)	move(Path src, Path dest,CopyOption... o) throws IOException

> The following is a sample program using only legacy I/O APIs.

```
11: public void io(File file) {
12:     if (file.exists()) {
13:         System.out.println("Absolute Path: " +
file.getAbsolutePath());
14:         System.out.println("Is Directory: " +
```

```
file.isDirectory());
15:       System.out.println("Parent Path: " + file.getParent());
16:       if (file.isFile()) {
17:          System.out.println("Size: " + file.length());
18:          System.out.println("Last Modified: " +
file.lastModified());
19:       } else {
20:          for (File subfile : file.listFiles()) {
21:             System.out.println("   " + subfile.getName());
22:       } } } }
```

If the path provided points to a valid file, the program outputs something similar to the following due to the if statement on line 16:

```
Absolute Path: C:\data\zoo.txt
Is Directory: false
Parent Path: C:\data
Size: 12382
Last Modified: 1650610000000
```

Finally, if the path provided points to a valid directory, such as C:\data, the program outputs something similar to the following, thanks to the else block:

```
Absolute Path: C:\data
Is Directory: true
Parent Path: C:\
   employees.txt
   zoo.txt
   zoo-backup.txt
```

> In the previous example, we used two backslashes (`\\`) in the path `String`, such as `C:\\data\\zoo.txt`. When the compiler sees a `\\` inside a `String` expression, it interprets it as a single `\` value.

> Let's write that same program using only NIO.2

```
26: public void nio(Path path) throws IOException {
27:    if (Files.exists(path)) {
28:       System.out.println("Absolute Path: " +
path.toAbsolutePath());
29:       System.out.println("Is Directory: " +
Files.isDirectory(path));
```

```
30:       System.out.println("Parent Path: " + path.getParent());
31:       if (Files.isRegularFile(path)) {
32:          System.out.println("Size: " + Files.size(path));
33:          System.out.println("Last Modified: "
34:             + Files.getLastModifiedTime(path));
35:       } else {
36:          try (Stream<Path> stream = Files.list(path)) {
37:             stream.forEach(p →
38:                System.out.println("   " + p.getFileName()));
39:       } } } }
```

Most of this example is equivalent and replaces the I/O method calls in the previous tables with the NIO.2 versions. However, there are key differences. First, line 25 declares a checked exception. More APIs in NIO.2 throw `IOException` than the I/O APIs did. In this case, `Files.size()`, `Files.getLastModifiedTime()`, and `Files.list()` throw an `IOException`.

The NIO.2 stream-based methods open a connection to the file system *that must be properly closed*; otherwise, a resource leak could ensue. A resource leak within the file system means the path may be locked from modification long after the process that used it is completed.

Handling Methods That Declare *IOException*

Many of the methods presented in this chapter declare `IOException`. Common causes of a method throwing this exception include the following:

- Loss of communication to the underlying file system.
- File or directory exists but cannot be accessed or modified.
- File exists but cannot be overwritten.
- File or directory is required but does not exist.

Providing NIO.2 Optional Parameters

TABLE 14.4 Common NIO.2 method arguments

Enum type	Interface inherited	Enum value	Details
LinkOption	CopyOption OpenOption	NOFOLLOW_LINKS	Do not follow symbolic links.
StandardCopyOption	CopyOption	ATOMIC_MOVE	Move file as atomic file system operation.
		COPY_ATTRIBUTES	Copy existing attributes to new file.
		REPLACE_EXISTING	Overwrite file if it already exists.
StandardOpenOption	OpenOption	APPEND	If file is already open for write, append to the end.
		CREATE	Create new file if it does not exist.
		CREATE_NEW	Create new file only if it does not exist; fail otherwise.
		READ	Open for read access.
		TRUNCATE_EXISTING	If file is already open for write, erase file and append to beginning.
		WRITE	Open for write access.
FileVisitOption	N/A	FOLLOW_LINKS	Follow symbolic links.

```
Path path = Path.of("schedule.xml");
boolean exists = Files.exists(path, LinkOption.NOFOLLOW_LINKS);
```

The `Files.exists()` simply checks whether a file exists. But if the parameter is a symbolic link, the method checks whether the target of the symbolic link exists, instead. Providing `LinkOption.NOFOLLOW_LINKS` means the default behavior will be overridden, and the method will check whether the symbolic link itself exists.

Interacting with NIO.2 Paths

> `Path` instances are immutable.

```
Path p = Path.of("whale");
p.resolve("krill");
System.out.println(p);  // whale
```

Viewing the Path

The `Path` interface contains three methods to retrieve basic information about the path representation.

1. `toString()` -> returns a String representation of the entire path;
2. `getNameCount()` & `getName()` -> are often used together to retrieve the number of elements in the path and a reference to each element, respectively.

```
Path path = Path.of("/land/hippo/harry.happy");
System.out.println("The Path is: " + path);
for(int i=0; i<path.getNameCount(); i++)
   System.out.println("   Element " + i + " is: " +
path.getName(i));
```

```
The Path is: /land/hippo/harry.happy
   Element 0 is: land
   Element 1 is: hippo
   Element 2 is: harry.happy
```

Even though this is an absolute path, the root element is not included in the list of names. As we said, these methods do not consider the root part of the path.

```
var p = Path.of("/");
System.out.print(p.getNameCount()); // 0
System.out.print(p.getName(0));     // IllegalArgumentException
```

Creating Part of the Path

The `Path` interface includes the `subpath()` method to select portions of a path. It takes two parameters: an inclusive `beginIndex` and an exclusive `endIndex`.

```
var p = Path.of("/mammal/omnivore/raccoon.image");
System.out.println("Path is: " + p);
for (int i = 0; i < p.getNameCount(); i++) {
   System.out.println("   Element " + i + " is: " +
p.getName(i));
}

System.out.println();
System.out.println("subpath(0,3): " + p.subpath(0, 3));
```

```
System.out.println("subpath(1,2): " + p.subpath(1, 2));
System.out.println("subpath(1,3): " + p.subpath(1, 3));
```

```
Path is: /mammal/omnivore/raccoon.image
   Element 0 is: mammal
   Element 1 is: omnivore
   Element 2 is: raccoon.image

subpath(0,3): mammal/omnivore/raccoon.image
subpath(1,2): omnivore
subpath(1,3): omnivore/raccoon.image
```

```
var q = p.subpath(0, 4); // IllegalArgumentException
var x = p.subpath(1, 1); // IllegalArgumentException
```

The first example throws an exception at runtime, since the maximum index value allowed is `3`. The second example throws an exception since the start and end indexes are the same, leading to an empty path value.

Accessing Path Elements

1. `getFileName()` -> returns the path element of the current file/directory
2. `getParent()` -> returns the full path of the containing directory, returns `null` if operated on the root path or at the top of a relative path.
3. `getRoot()` -> returns the root element of the file within the file system, or `null` if the path is a relative path.

```
public void printPathInformation(Path path) {
   System.out.println("Filename is: " + path.getFileName());
   System.out.println("   Root is: " + path.getRoot());
   Path currentParent = path;
   while((currentParent = currentParent.getParent()) != null)
      System.out.println("   Current parent is: " +
currentParent);
   System.out.println();
}

printPathInformation(Path.of("zoo"));
printPathInformation(Path.of("/zoo/armadillo/shells.txt"));
printPathInformation(Path.of("./armadillo/../shells.txt"));
```

```
Filename is: zoo
   Root is: null

Filename is: shells.txt
   Root is: /
   Current parent is: /zoo/armadillo
   Current parent is: /zoo
   Current parent is: /

Filename is: shells.txt
   Root is: null
   Current parent is: ./armadillo/..
   Current parent is: ./armadillo
   Current parent is: .
```

Resolving Paths

The `resolve()` method provides overloaded versions that let you pass either a `Path` or `String` parameter. The object on which the `resolve()` method is invoked becomes the basis of the new `Path` object, with the input argument being appended onto the `Path`.

```
Path path1 = Path.of("/cats/../panther");
Path path2 = Path.of("food");
System.out.println(path1.resolve(path2));
```

```
/cats/../panther/food
```

Like the other methods we've seen, `resolve()` does not clean up path symbols. In this example, the input argument to the `resolve()` method was a relative path, but what if it had been an absolute path?

```
Path path3 = Path.of("/turkey/food");
```

Since the input parameter is an absolute path, the output would be the following:

```
/tiger/cage
```

Relativizing a Path

> The `Path` interface includes a `relativize()` method for constructing the relative path from one `Path` to another, often using path symbols.

```
var path1 = Path.of("fish.txt");
var path2 = Path.of("friendly/birds.txt");
System.out.println(path1.relativize(path2));
System.out.println(path2.relativize(path1));
```

```
../friendly/birds.txt
../../fish.txt
```

The idea is this: if you are pointed at a path in the file system, what steps would you need to take to reach the other path? For example, to get to `fish.txt` from `friendly/birds.txt`, you need to go up two levels (the file itself counts as one level) and then select `fish.txt`.

If both path values are relative, the `relativize()` method computes the paths as if they are in the same current working directory. Alternatively, if both path values are absolute, the method computes the relative path from one absolute location to another, regardless of the current working directory. The following example demonstrates this property when run on a Windows computer:

```
var path3 = Path.of("E:\\habitat");
var path4 = Path.of("E:\\sanctuary\\raven\\poe.txt");
System.out.println(path3.relativize(path4));
System.out.println(path4.relativize(path3));
```

This code snippet produces the following output:

```
..\sanctuary\raven\poe.txt
..\..\..\habitat
```

> The `relativize()` method requires both paths to be absolute or relative and throws an exception if the types are mixed.
> On Windows-based systems, it also requires that if absolute paths are used, both paths must have the same root directory or drive letter.

Normalizing a Path

Remember, the path symbol `..` refers to the parent directory, while the path symbol `.` refers to the current directory. We can apply `normalize()`

to some of our previous paths.

```
var p1 = Path.of("./armadillo/../shells.txt");
System.out.println(p1.normalize()); // shells.txt

var p2 = Path.of("/cats/../panther/food");
System.out.println(p2.normalize()); // /panther/food

var p3 = Path.of("../../fish.txt");
System.out.println(p3.normalize()); //  ../../fish.txt
```

The first two examples apply the path symbols to remove the redundancies, but what about the last one? That is as simplified as it can be. The `normalize()` method does not remove all of the path symbols, only the ones that can be reduced.
The `normalize()` method also allows us to compare equivalent paths.

```
var p1 = Path.of("/pony/../weather.txt");
var p2 = Path.of("/weather.txt");
System.out.println(p1.equals(p2));                          //
false
System.out.println(p1.normalize().equals(p2.normalize())); //
true
```

Retrieving the Real File System Path

While working with theoretical paths is useful, sometimes you want to verify that the path exists within the file system using `toRealPath()`. This method is similar to `normalize()` in that it eliminates any redundant path symbols. It is also similar to `toAbsolutePath()`, in that it will join the path with the current working directory if the path is relative.

Unlike those two methods, though, `toRealPath()` will throw an exception if the path does not exist. In addition, it will follow symbolic links, with an optional `LinkOption` varargs parameter to ignore them.

Reviewing NIO.2 Path APIs

Description	Path instance method
File path as string	`String toString()`
Single segment	`Path getName()`
Number of segment	`int getNameCount()`

Description	Path instance method
Segments in range	`Path subpath(int beginIndex, int endIndex)`
Final segment	`Path getFileName()`
Immediate parent	`Path getParent()`
Top-level segment	`Path getRoot()`
Concatenate paths	`Path resolve(String p)` `Path resolve(Path p)`
Construct path to one provided	`Path relativize(Path p)`
Remove redundant parts of paths	`Path normalize()`
Follow symbolic links to find path on file system	`Path toRealPath()`

Creating, Moving, and Deleting Files and Directories

Making Directories

```
public static Path createDirectory(Path dir, FileAttribute<?>…
attrs) throws IOException
public static Path createDirectories(Path dir, FileAttribute<?>…
attrs) throws IOException
```

The `createDirectory()` method will create a directory and throw an exception if it already exists or if the paths leading up to the directory do not exist. The `createDirectories()` method creates the target directory along with any nonexistent parent directories leading up to the path. If all of the directories already exist, `createDirectories()` will simply complete without doing anything. This is useful in situations where you want to ensure a directory exists and create it if it does not.

```
Files.createDirectory(Path.of("/bison/field"));
Files.createDirectories(Path.of("/bison/field/pasture/green"));
```

The first example creates a new directory, `field`, in the directory `/bison`, assuming `/bison` exists; otherwise, an exception is thrown. Contrast this with the second example, which creates the directory `green` along with any of the following parent directories if they do not already exist, including `bison`, `field`, and `pasture`.

Copying Files

```
public static Path copy(Path source, Path target, CopyOption…
options) throws IOException
```

```
Files.copy(Path.of("/panda/bamboo.txt"),    Path.of("/panda-
save/bamboo.txt"));
Files.copy(Path.of("/turtle"), Path.of("/turtleCopy"));
```

When directories are copied, the copy is shallow. A *shallow copy* means that the files and subdirectories within the directory are not copied. A *deep copy* means that the entire tree is copied, including all of its content and subdirectories. A deep copy typically requires *recursion*, where a method calls itself.

```
public void copyPath(Path source, Path target) {
   try {
      Files.copy(source, target);
      if(Files.isDirectory(source))
         try (Stream<Path> s = Files.list(source)) {
            s.forEach(p → copyPath(p,
               target.resolve(p.getFileName())));
         }
   } catch(IOException e) {
      //Handle exception
   }
}
```

Copying and Replacing Files

By default, if the target already exists, the `copy()` method will throw an exception. You can change this behavior by providing the `StandardCopyOption` enum value `REPLACE_EXISTING` to the method. The following method call will overwrite the `movie.txt` file if it already exists:

```
Files.copy(Path.of("book.txt"), Path.of("movie.txt"),
StandardCopyOption.REPLACE_EXISTING);
```

Copying Files with I/O Streams

```
public static long copy(InputStream in, Path target,
   CopyOption… options) throws IOException
```

```
public static long copy(Path source, OutputStream out)
   throws IOException
```

The first method reads the contents of an I/O stream and writes the output to a file. The second method reads the contents of a file and writes the output to an I/O stream. These methods are quite convenient if you need to quickly read/write data from/to disk.

```
try (var is = new FileInputStream("source-data.txt")) {
   // Write I/O stream data to a file
   Files.copy(is, Path.of("/mammals/wolf.txt"));
}

Files.copy(Path.of("/fish/clown.xsl"), System.out);
```

Copying Files into a Directory

For example, let's say we have a file, `food.txt`, and a directory, `/enclosure`. Both the file and directory exist. What do you think is the result of executing the following process?

```
var file = Path.of("food.txt");
var directory = Path.of("/enclosure");
Files.copy(file, directory);
```

It throws an exception. The command tries to create a new file named `/enclosure`. Since the path `/enclosure` already exists, an exception is thrown at runtime.
On the other hand, if the directory did not exist, the process would create a new file with the contents of `food.txt`, but the file would be called `/enclosure`. Remember, we said files may not need to have extensions, and in this example, it matters.
The correct way to copy the file into the directory is to do the following:

```
var file = Path.of("food.txt");
var directory = Path.of("/enclosure/food.txt");
Files.copy(file, directory);
```

Moving or Renaming Paths with *move()*

```
public static Path move(Path source, Path target, CopyOption…
```

```
options) throws IOException
```

```
Files.move(Path.of("C:\\zoo"), Path.of("C:\\zoo-new"));

Files.move(Path.of("C:\\user\\addresses.txt"), Path.of("C:\\zoo-
new\\addresses2.txt"));
```

The first example renames the `zoo` directory to a `zoo-new` directory, keeping all of the original contents from the source directory. The second example moves the `addresses.txt` file from the directory `user` to the directory `zoo-new` and renames it `addresses2.txt`.

Similarities between *move()* and *copy()*

Like `copy()`, `move()` requires `REPLACE_EXISTING` to overwrite the target if it exists; otherwise, it will throw an exception. Also like `copy()`, `move()` will not put a file in a directory if the source is a file and the target is a directory. Instead, it will create a new file with the name of the directory.

Performing an Atomic Move

```
Files.move(Path.of("mouse.txt"), Path.of("gerbil.txt"),
StandardCopyOption.ATOMIC_MOVE);
```

Put another way, any process monitoring the file system never sees an incomplete or partially written file. If the file system does not support this feature, an `AtomicMoveNotSupportedException` will be thrown.

Note that while `ATOMIC_MOVE` is available as a member of the `StandardCopyOption` type, it will likely throw an exception if passed to a `copy()` method.

Deleting a File with *delete()* and *deleteIfExists()*

```
public static void delete(Path path) throws IOException

public static boolean deleteIfExists(Path path) throws
IOException
```

To delete a directory, it must be empty. Both of these methods throw an exception if operated on a nonempty directory. In addition, if the path is a

symbolic link, the symbolic link will be deleted, not the path that the symbolic link points to.

The `delete()` method throws an exception if the path does not exist, while the `deleteIfExists()` method returns `true` if the delete was successful or `false` otherwise. Similar to `createDirectories()`, `deleteIfExists()` is useful in situations where you want to ensure that a path does not exist and delete it if it does.

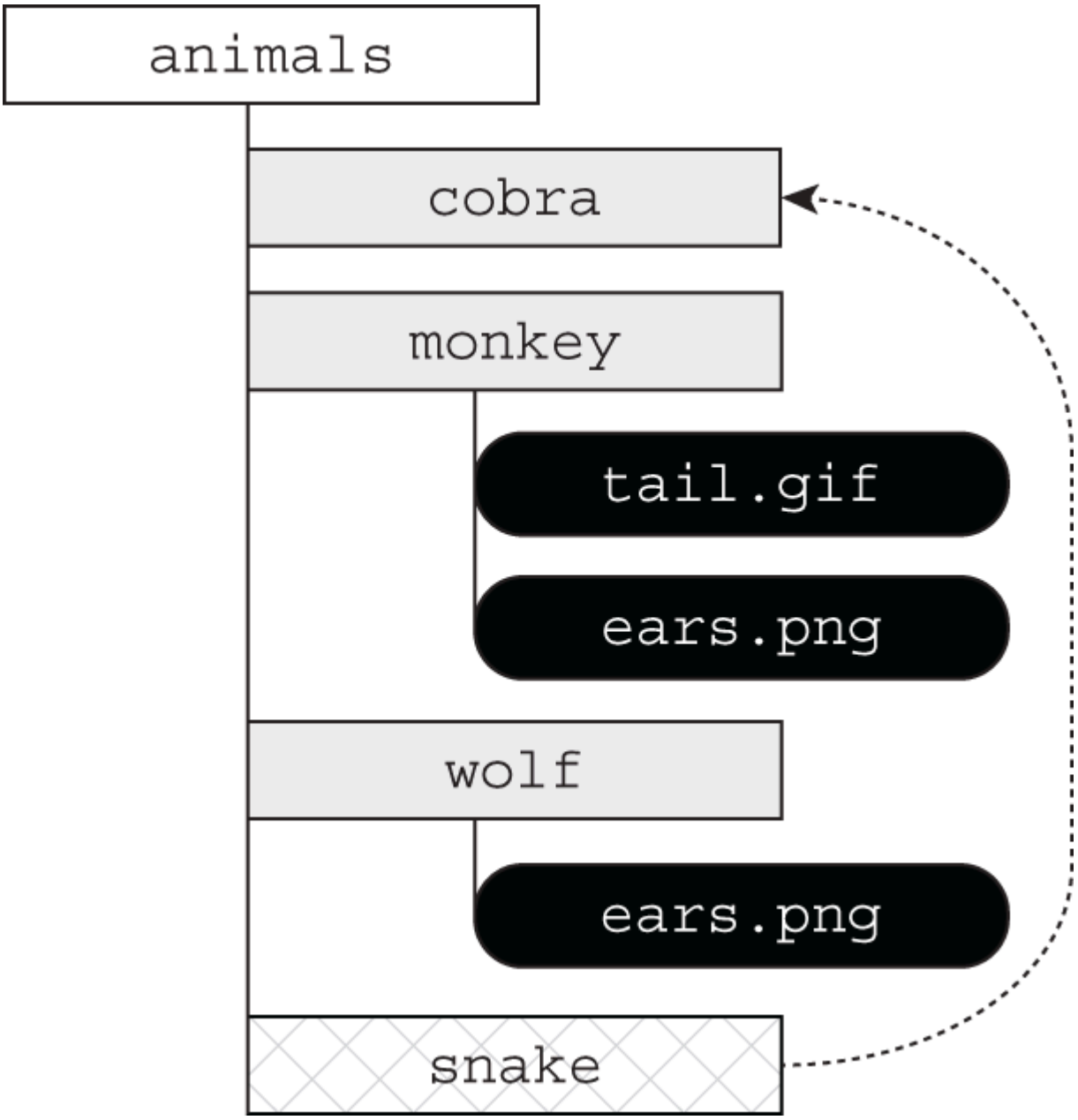
```
Files.delete(Path.of("/vulture/feathers.txt"));
Files.deleteIfExists(Path.of("/pigeon"));
```

The first example deletes the `feathers.txt` file in the `vulture` directory, and it throws a `NoSuchFileException` if the file or directory does not exist. The second example deletes the `pigeon` directory, assuming it is empty. If the `pigeon` directory does not exist, the second line will not throw an exception.

Comparing Files with *isSameFile()* and *mismatch()*

`isSameFile()` takes two `Path` objects as input, resolves all path symbols, and follows symbolic links. Despite the name, the method can also be used to determine whether two `Path` objects refer to the same directory.

While most uses of `isSameFile()` will trigger an exception if the paths do not exist, there is a special case in which it does not. If the two path objects are equal in terms of `equals()`, the method will just return `true` without checking whether the file exists.




```
System.out.println(Files.isSameFile(
   Path.of("/animals/cobra"),
   Path.of("/animals/snake"))); // true

System.out.println(Files.isSameFile(
   Path.of("/animals/monkey/ears.png"),
   Path.of("/animals/wolf/ears.png"))); // false
```

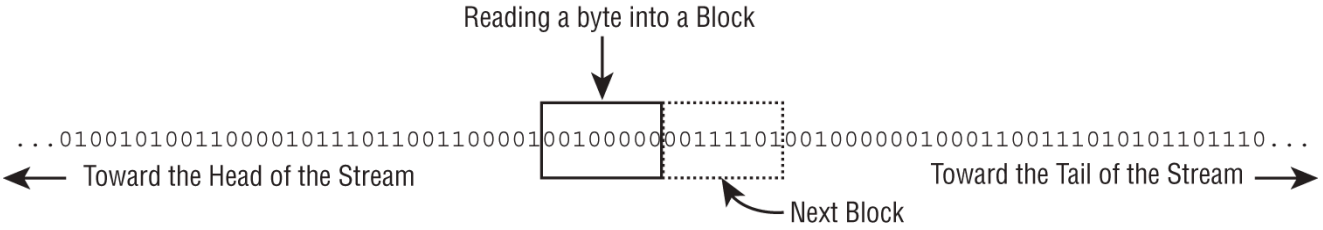
Sometimes you want to compare the contents of the file rather than whether it is physically the same file. For example, we could have two files with text `hello`. The `mismatch()` method was introduced in Java 12 to help us out here. It takes two `Path` objects as input. The method returns `-1` if the files are the same; otherwise, it returns the index of the first position in the file that differs.

```
System.out.println(Files.mismatch(
   Path.of("/animals/monkey.txt"),
   Path.of("/animals/wolf.txt")))
```

Suppose `monkey.txt` contains the name `Harold` and `wolf.txt` contains the name `Howler`. The previous code prints `1` in that case because the second position is different, and we use zero-based indexing in Java.
The `mismatch()` method is symmetric and returns the same result regardless of the order of the parameters.

Introducing I/O Streams

Understanding I/O Stream Fundamentals



Learning I/O Stream Nomenclature

Byte Streams vs. Character Streams

The `java.io` API defines two sets of I/O stream classes for reading and writing I/O streams: byte I/O streams and character I/O streams. We use both types of I/O streams throughout this chapter.

Differences Between Byte and Character I/O Streams

- Byte I/O streams read/write binary data (`0`s and `1`s) and have class names that end in `InputStream` or `OutputStream`.
- Character I/O streams read/write text data and have class names that end in `Reader` or `Writer`.

The API frequently includes similar classes for both byte and character I/O streams, such as `FileInputStream` and `FileReader`. The difference between the two classes is based on how the bytes are read or written.

The byte I/O streams are primarily used to work with binary data, such as an image or executable file, while character I/O streams are used to work with text files.
The *character encoding* determines how characters are encoded and stored in bytes in an I/O stream and later read back or decoded as characters.

In Java, the character encoding can be specified using the `Charset` class by passing a name value to the static `Charset.forName()` method, such as in the following examples:

```
Charset usAsciiCharset = Charset.forName("US-ASCII");
Charset utf8Charset = Charset.forName("UTF-8");
Charset utf16Charset = Charset.forName("UTF-16");
```

Input vs. Output Streams

Most `InputStream` classes have a corresponding `OutputStream` class, and vice versa.
It follows, then, that most `Reader` classes have a corresponding `Writer` class.
There are some exceptions to this rule. For the exam, you should know that `PrintWriter` has no accompanying `PrintReader` class. Likewise, the `PrintStream` is an `OutputStream` that has no corresponding `InputStream` class. It also does not have `Output` in its name.

Low-Level vs. High-Level Streams

> A *low-level stream* connects directly with the source of the data, such as a file, an array, or a `String`. Low-level I/O streams process the raw data or resource and are accessed in a direct and unfiltered manner. For example, a `FileInputStream` is a class that reads file data one byte at a time.

> A *high-level stream* is built on top of another I/O stream using wrapping. *Wrapping* is the process by which an instance is passed to the constructor of another class, and operations on the resulting instance are filtered and applied to the original instance.

```
try (var br = new BufferedReader(new FileReader("zoo-data.txt")))
{
   System.out.println(br.readLine());
}
```

In this example, `FileReader` is the low-level I/O stream, whereas `BufferedReader` is the high-level I/O stream that takes a `FileReader` as input. Many operations on the high-level I/O stream pass through as operations to the underlying low-level I/O stream, such as `read()` or `close()`. Other operations override or add new functionality to the low-level I/O stream methods. The high-level I/O stream may add new methods,

such as `readLine()`, as well as performance enhancements for reading and filtering the low-level data.

High-level I/O streams can also take other high-level I/O streams as input. For example, although the following code might seem a little odd at first, the style of wrapping an I/O stream is quite common in practice:

```
try (var ois = new ObjectInputStream(
       new BufferedInputStream(
          new FileInputStream("zoo-data.ser")))) {
   System.out.print(ois.readObject());
}
```

In this example, the low-level FileInputStream interacts directly with the file, which is wrapped by a high-level BufferedInputStream to improve performance. Finally, the entire object is wrapped by another high-level ObjectInputStream, which allows us to interpret the data as a Java object.

Stream Base Classes

The `java.io` library defines four abstract classes that are the parents of all I/O stream classes defined within the API: `InputStream`, `OutputStream`, `Reader`, and `Writer`.
The constructors of high-level I/O streams often take a reference to the abstract class. For example, BufferedWriter takes a Writer object as input, which allows it to take any subclass of Writer.

One common area where the exam likes to play tricks on you is mixing and matching I/O stream classes that are not compatible with each other. For example, take a look at each of the following examples and see whether you can determine why they do not compile:

```
new BufferedInputStream(new FileReader("z.txt"));  // DOES NOT
COMPILE
new BufferedWriter(new FileOutputStream("z.txt")); // DOES NOT
COMPILE
new ObjectInputStream(new FileOutputStream("z.txt"));
// DOES NOT COMPILE
new BufferedInputStream(new InputStream());        // DOES NOT
COMPILE
```

Decoding I/O Class Names

Class name	Description
`InputStream`	Abstract class for all input byte streams
`OutputStream`	Abstract class for all output byte streams
`Reader`	Abstract class for all input character streams
`Writer`	Abstract class for all output character streams

Class name	Low/High level	Description
`FileInputStream`	Low	Reads file data as bytes
`FileOutputStream`	Low	Writes file data as bytes
`FileReader`	Low	Reads file data as characters
`FileWriter`	Low	Writes file data as characters
`BufferedInputStream`	High	Reads byte data from existing `InputStream` in buffered manner, which improves efficiency and performance
`BufferedOutputStream`	High	Writes byte data to existing `OutputStream` in buffered manner, which improves efficiency and performance
`BufferedReader`	High	Reads character data from existing `Reader` in buffered manner, which improves efficiency and performance
`BufferedWriter`	High	Writes character data to existing `Writer` in buffered manner, which improves efficiency and performance
`ObjectInputStream`	High	Deserializes primitive Java data types and graphs of Java objects from existing `InputStream`
`ObjectOutputStream`	High	Serializes primitive Java data types and graphs of Java objects to existing `OutputStream`
`PrintStream`	High	Writes formatted representations of Java objects to binary stream

Class name	Low/High level	Description
`PrintWriter`	High	Writes formatted representations of Java objects to character stream

Reading and Writing Files

Using I/O Streams

I/O streams are all about reading/writing data, so it shouldn't be a surprise that the most important methods are `read()` and `write()`. Both `InputStream` and `Reader` declare a `read()` method to read byte data from an I/O stream. Likewise, `OutputStream` and `Writer` both define a `write()` method to write a byte to the stream.

```
void copyStream(InputStream in, OutputStream out) throws
IOException {
   int b;
   while ((b = in.read()) != -1) {
      out.write(b);
   }
}

void copyStream(Reader in, Writer out) throws IOException {
   int b;
   while ((b = in.read()) != -1) {
      out.write(b);
   }
}
```

The `offset` and `length` values are applied to the array itself. For example, an `offset` of `3` and `length` of `5` indicates that the stream should read up to five bytes/characters of data and put them into the array starting with position `3`. Let's look at an example:

```
10: void copyStream(InputStream in, OutputStream out) throws
IOException {
11:    int batchSize = 1024;
12:    var buffer = new byte[batchSize];
13:    int lengthRead;
14:    while ((lengthRead = in.read(buffer, 0, batchSize))> 0) {
15:       out.write(buffer, 0, lengthRead);
```

```
16:         out.flush();
17:     }
```

We also added a `flush()` method on line 16 to reduce the amount of data lost if the application terminates unexpectedly. When data is written to an output stream, the underlying operating system does not guarantee that the data will make it to the file system immediately. The `flush()` method requests that all accumulated data be written immediately to disk.

```
26: void copyTextFile(File src, File dest) throws IOException {
27:     try (var reader = new BufferedReader(new FileReader(src));
28:         var writer = new BufferedWriter(new FileWriter(dest)))
{
29:         String line = null;
30:         while ((line = reader.readLine()) ≠ null) {
31:             writer.write(line);
32:             writer.newLine();
33:         } } }
```

The key is to choose the most useful high-level classes. In this case, we are dealing with a `File`, so we want to use a `FileReader` and `FileWriter`. Both classes have constructors that can take either a `String` representing the location or a `File` directly.

If the source file does not exist, a `FileNotFoundException`, which inherits `IOException`, will be thrown. If the destination file already exists, this implementation will overwrite it. We can pass an optional `boolean` second parameter to `FileWriter` for an `append` flag if we want to change this behavior.

We can do a little better than `BufferedOutputStream` and `BufferedWriter` by using a `PrintStream` and `PrintWriter`. These classes contain four key methods. The `print()` and `println()` methods print data with and without a new line, respectively. There are also the `format()` and `printf()` methods, which we describe in the section on user interactions.

```
void copyTextFile(File src, File dest) throws IOException {
    try (var reader = new BufferedReader(new FileReader(src));
        var writer = new PrintWriter(new FileWriter(dest))) {
        String line = null;
        while ((line = reader.readLine()) ≠ null)
            writer.println(line);
```

```
        }
    }
```

The line separator is `\n` or `\r\n`, depending on your operating system. The `println()` method takes care of this for you. If you need to get the character directly, either of the following will return it for you as a `String`:

```
System.getProperty("line.separator");
System.lineSeparator()
```

Enhancing with *Files*

```
private void copyPathAsString(Path input, Path output) throws
IOException {
   String string = Files.readString(input);
   Files.writeString(output, string);
}

private void copyPathAsBytes(Path input, Path output) throws
IOException {
    byte[] bytes = Files.readAllBytes(input);
    Files.write(output, bytes);
}

private void copyPathAsLines(Path input, Path output) throws
IOException {
   List<String> lines = Files.readAllLines(input);
   Files.write(output, lines);
}
```

That's pretty concise! You can read a `Path` as a `String`, a `byte` array, or a `List`. Be aware that the entire file is read at once for all three of these, thereby storing all of the contents of the file in memory at the same time. If the file is significantly large, you may trigger an `OutOfMemoryError` when trying to load all of it into memory. Luckily, there is an alternative. This time, we print out the file as we read it.

```
private void readLazily(Path path) throws IOException {
   try (Stream<String> s = Files.lines(path)) {
      s.forEach(System.out::println);
```

```
    }
}
```

Now the contents of the file are read and processed lazily, which means that only a small portion of the file is stored in memory at any given time. Taking things one step further, we can leverage other stream methods for a more powerful example.

```
try (var s = Files.lines(path)) {
   s.filter(f -> f.startsWith("WARN:"))
      .map(f -> f.substring(5))
      .forEach(System.out::println);
}
```

This sample code searches a log for lines that start with `WARN:`, outputting the text that follows.

```
Files.readAllLines(Path.of("birds.txt")).forEach(System.out::prin
tln);
Files.lines(Path.of("birds.txt")).forEach(System.out::println);
```

The first line reads the entire file into memory and performs a print operation on the result, while the second line lazily processes each line and prints it as it is read. The advantage of the second code snippet is that it does not require the entire file to be stored in memory at any time.

```
Files.readAllLines(Path.of("birds.txt"))
    .filter(s -> s.length()> 2)
    .forEach(System.out::println);
```

The `readAllLines()` method returns a `List`, not a `Stream`, so the `filter()` method is not available.

Combining with *newBufferedReader()* and *newBufferedWriter()*

```
private void copyPath(Path input, Path output) throws IOException
{
   try (var reader = Files.newBufferedReader(input);
        var writer = Files.newBufferedWriter(output)) {
      String line = null;
      while ((line = reader.readLine()) != null) {
```

```
            writer.write(line);
            writer.newLine();
        } } }
```

You can wrap I/O stream constructors to produce the same effect, although it's a lot easier to use the factory method. The first method, `newBufferedReader()`, reads the file specified at the `Path` location using a `BufferedReader` object.

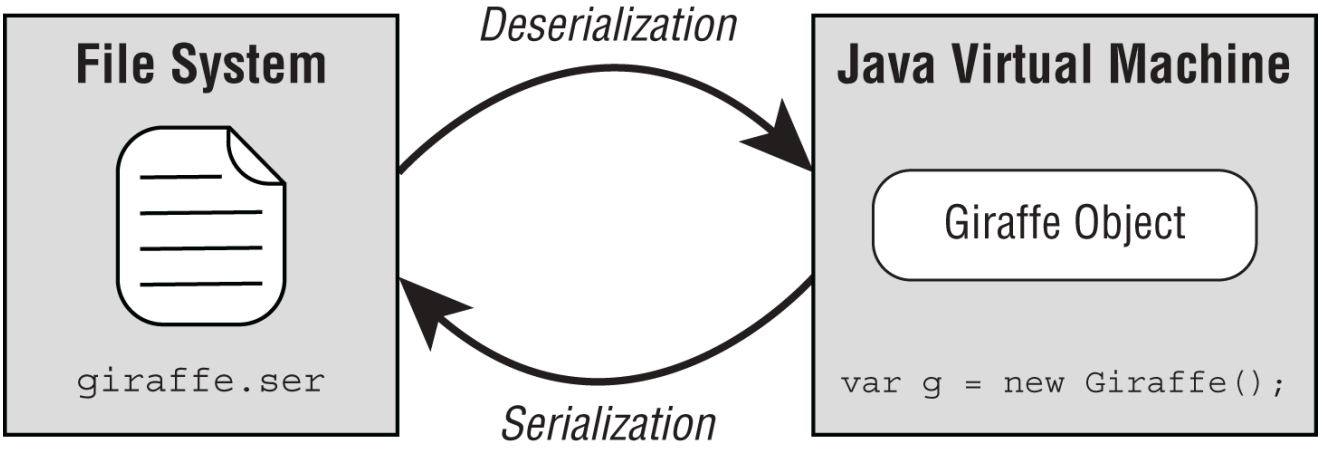
Class	Method name	Description
All input streams	`int read()`	Reads single byte or returns `-1` if no bytes available.
`InputStream`	`int read(byte[] b)`	Reads values into buffer and returns number of bytes or characters read.
`Reader`	`int read(char[] c)`	
`InputStream`	`int read(byte[] b, int offset, int length)`	
All output streams	`void write(int b)`	Writes single byte.
`OutputStream`	`int write(byte[] b)`	Writes `length` values from array into stream, starting with `offset` index.
`Writer`	`int write(char[] c)`	
`OutputStream`	`int write(byte[] b, int offset, int length)`	
`Writer`	`int write(byte[] b, int offset, int length)`	
`InputStream`	`byte[] readAllBytes()`	Reads data in bytes.
`BufferedReader`	`String readLine()`	Reads line of data.
`Writer`	`void write(String line)`	Writes line of data.
`BufferedWriter`	`void newLine()`	Writes new line.
All output streams	`void flush()`	Flushes buffered data through stream.
All streams	`void close()`	Closes stream and releases resources.

TABLE 14.9 Common `Files` NIO.2 read and write `static` methods

Method Name	Description
`byte[] readAllBytes(Path path)`	Reads all data as bytes
`String readString(Path path)`	Reads all data into `String`
`List<String> readAllLines(Path path)`	Read all data into `List`
`Stream<String> lines(Path path)`	Lazily reads data
`void write(Path path, byte[] bytes)`	Writes array of bytes
`void writeString(Path path, String string)`	Writes `String`
`void write(Path path, List<String> list)`	Writes list of lines (technically, an `Iterable` of `CharSequence`, but you don't need to know that for the exam)

Serializing Data

> *Serialization* is the process of converting an in-memory object to a byte stream. Likewise, *deserialization* is the process of converting from a byte stream into an object. Serialization often involves writing an object to a stored or transmittable format, while deserialization is the reciprocal process.



Applying the *Serializable* Interface

To serialize an object using the I/O API, the object must implement the `java.io.Serializable` interface. The `Serializable` interface is a marker interface, which means it does not have any methods. Any class can implement the `Serializable` interface since there are no required methods to implement.

The purpose of using the `Serializable` interface is to inform any process attempting to serialize the object that you have taken the proper steps to make the object serializable.

```
import java.io.Serializable;

public class Gorilla implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private int age;
   private Boolean friendly;
   private transient String favoriteFood;
   // Constructors/Getters/Setters/toString() omitted
}
```

Marking Data *transient*

The `transient` modifier can be used for sensitive data of the class, like a `password`. There are other objects it does not make sense to serialize, like the state of an in-memory `Thread`. If the object is part of a serializable object, we just mark it `transient` to ignore these select instance members.
What happens to data marked `transient` on deserialization? It reverts to its default Java values, such as `0.0` for `double`, or `null` for an object.

Ensuring That a Class Is Serializable

How to Make a Class Serializable

- The class must be marked `Serializable`.
- Every instance member of the class must be serializable, marked `transient`, or have a `null` value at the time of serialization.

Storing Data with *ObjectOutputStream* and *ObjectInputStream*

> The `ObjectInputStream` class is used to deserialize an object, while the `ObjectOutputStream` is used to serialize an object. They are high-level streams that operate on existing I/O streams.

```
void saveToFile(List<Gorilla> gorillas, File dataFile) throws
IOException {
   try (var out = new ObjectOutputStream(
            new BufferedOutputStream(
               new FileOutputStream(dataFile)))) {
      for (Gorilla gorilla : gorillas)
         out.writeObject(gorilla);
```

```
    }
}

List<Gorilla> readFromFile(File dataFile) throws IOException,
ClassNotFoundException {
    var gorillas = new ArrayList<Gorilla>();
    try (var in = new ObjectInputStream(
          new BufferedInputStream(new FileInputStream(dataFile))))
{
       while (true) {
          var object = in.readObject();
          if (object instanceof Gorilla g)
             gorillas.add(g);
       }
    } catch (EOFException e) {
       // File end reached
    }
    return gorillas;
}
```

If your program happens to know the number of objects in the I/O stream, you can call `readObject()` a fixed number of times, rather than using an infinite loop.

Understanding the Deserialization Creation Process

> When you deserialize an object, *the constructor of the serialized class, along with any instance initializers, is not called when the object is created*.

Any `static` or `transient` fields are ignored. Values that are not provided will be given their default Java value, such as `null` for `String`, or `0` for `int` values.

Interacting with Users

Printing Data to the User

Java includes two `PrintStream` instances for providing information to the user: `System.out` and `System.err`. While `System.out` should be old hat to you, `System.err` might be new to you. The syntax for calling and using `System.err` is the same as for `System.out`, but it is used to report errors to the user in a separate I/O stream from the regular output information.

Reading Input as an I/O Stream

The System.in returns an InputStream and is used to retrieve text input from the user. It is commonly wrapped with a BufferedReader via an InputStreamReader to use the readLine() method.

```
var reader = new BufferedReader(new
InputStreamReader(System.in));
String userInput = reader.readLine();
System.out.println("You entered: " + userInput);
```

When executed, this application first fetches text from the user until the user presses the Enter key. It then outputs the text the user entered to the screen.

Acquiring Input with *Console*

The `Console` class is a singleton because it is accessible only from a factory method and only one instance of it is created by the JVM.

```
Console console = System.console();
if (console ≠ null) {
   String userInput = console.readLine();
   console.writer().println("You entered: " + userInput);
   console.flush();
} else {
   System.err.println("Console not available");
}
```

Obtaining Underlying I/O Streams

The `Console` class includes access to two streams for reading and writing data.

```
public Reader reader()
public PrintWriter writer()
```

Formatting Console Data

```
// PrintStream
public PrintStream format(String format, Object… args)
public PrintStream format(Locale loc, String format, Object…
args)

// PrintWriter
```

```
public PrintWriter format(String format, Object… args)
public PrintWriter format(Locale loc, String format, Object…
args)
```

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
   console.writer().println("Welcome to Our Zoo!");
   console.format("It has %d animals and employs %d people", 391,
25);
   console.writer().println();
   console.printf("The zoo spans %5.1f acres", 128.91);
   console.flush();
}
```

```
-- Output --
Welcome to Our Zoo!
It has 391 animals and employs 25 people
The zoo spans 128.9 acres.
```

Reading Console Data

```
public String readLine()
public String readLine(String fmt, Object… args)

public char[] readPassword()
public char[] readPassword(String fmt, Object… args)
```

Like using `System.in` with a `BufferedReader`, the `Console readLine()` method reads input until the user presses the Enter key. The overloaded version of `readLine()` displays a formatted message prompt prior to requesting input.

The `readPassword()` methods are similar to the `readLine()` method, with two important differences.

- The text the user types is not echoed back and displayed on the screen as they are typing. Note that the password is not encrypted.
- The data is returned as a `char[]` instead of a `String`.

Working with Advanced APIs

Manipulating Input Streams

```
// InputStream and Reader
public boolean markSupported()
public void mark(int readLimit)
public void reset() throws IOException
public long skip(long n) throws IOException
```

The `mark()` and `reset()` methods return an I/O stream to an earlier position. Before calling either of these methods, you should call the `markSupported()` method, which returns `true` only if `mark()` is supported. The `skip()` method is pretty simple; it basically reads data from the I/O stream and discards the contents.

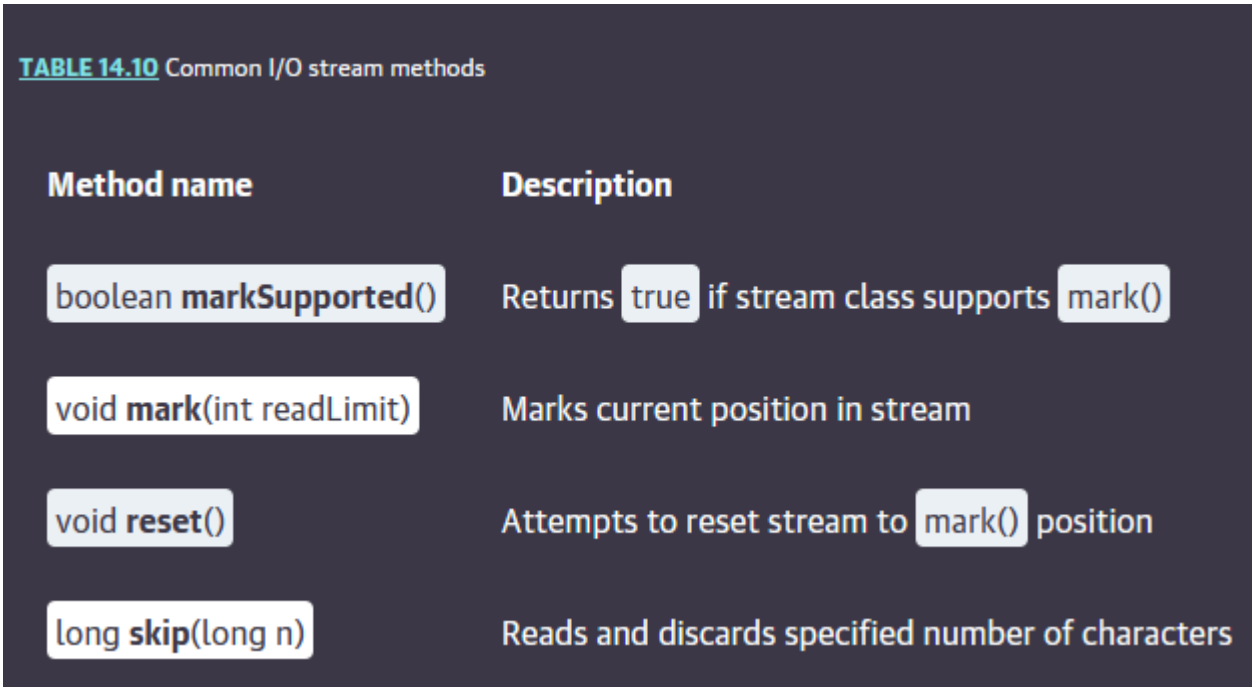


TABLE 14.10 Common I/O stream methods

Method name	Description
boolean **markSupported**()	Returns `true` if stream class supports `mark()`
void **mark**(int readLimit)	Marks current position in stream
void **reset**()	Attempts to reset stream to `mark()` position
long **skip**(long n)	Reads and discards specified number of characters

Discovering File Attributes

Checking for Symbolic Links

Earlier, we saw that the `Files` class has methods called `isDirectory()` and `isRegularFile()`, which are similar to the `isDirectory()` and `isFile()` methods on `File`. While the `File` object can't tell you if a reference is a symbolic link, the `isSymbolicLink()` method on `Files` can. It is possible for `isDirectory()` or `isRegularFile()` to return `true` for a symbolic link, as long as the link resolves to a directory or regular file, respectively.

Checking File Accessibility

In many file systems, it is possible to set a `boolean` attribute to a file that marks it hidden, readable, or executable. The `Files` class includes methods that expose this information: `isHidden()`, `isReadable()`, `isWriteable()`, and `isExecutable()`.

A hidden file can't normally be viewed when listing the contents of a directory. The readable, writable, and executable flags are important in file systems where the filename can be viewed, but the user may not have permission to open the file's contents, modify the file, or run the file as a program, respectively.

Understanding Attribute and View Types

TABLE 14.11 The attributes and view types

Attributes interface	View interface	Description
BasicFileAttributes	BasicFileAttributeView	Basic set of attributes supported by all file systems
DosFileAttributes	DosFileAttributeView	Basic set of attributes along with those supported by DOS/Windows-based systems
PosixFileAttributes	PosixFileAttributeView	Basic set of attributes along with those supported by POSIX systems, such as Unix, Linux, Mac, etc.

Modifying Attributes

```
public static <V extends FileAttributeView> V
getFileAttributeView(
   Path path,
   Class<V> type,
   LinkOption… options)

// BasicFileAttributeView instance method
public void setTimes(FileTime lastModifiedTime,
   FileTime lastAccessTime, FileTime createTime)
```

Traversing a Directory Tree

Traversing a directory, also referred to as walking a directory tree, is the process by which you start with a parent directory and iterate over all of its descendants until some condition is met or there are no more elements over which to iterate.

Walking a Directory

```
public static Stream<Path> walk(Path start, FileVisitOption…
options) throws IOException

public static Stream<Path> walk(Path start, int maxDepth,
FileVisitOption… options) throws IOException
```

Searching a Directory

```
public static Stream<Path> find(Path start,
   int maxDepth,
   BiPredicate<Path, BasicFileAttributes> matcher,
   FileVisitOption… options) throws IOException
```

The `find()` method behaves in a similar manner as the `walk()` method, except that it takes a `BiPredicate` to filter the data. It also requires a depth limit to be set. Like `walk()`, `find()` also supports the `FOLLOW_LINK` option.

```
var path = Path.of("/bigcats");
long minSize = 1_000;
try (var s = Files.find(path, 10,
      (p, a) -> a.isRegularFile()
         && p.toString().endsWith(".java")
         && a.size() > minSize)) {
   s.forEach(System.out::println);
}
```

This example searches a directory tree and prints all `.java` files with a size of at least 1,000 bytes, using a depth limit of `10`.

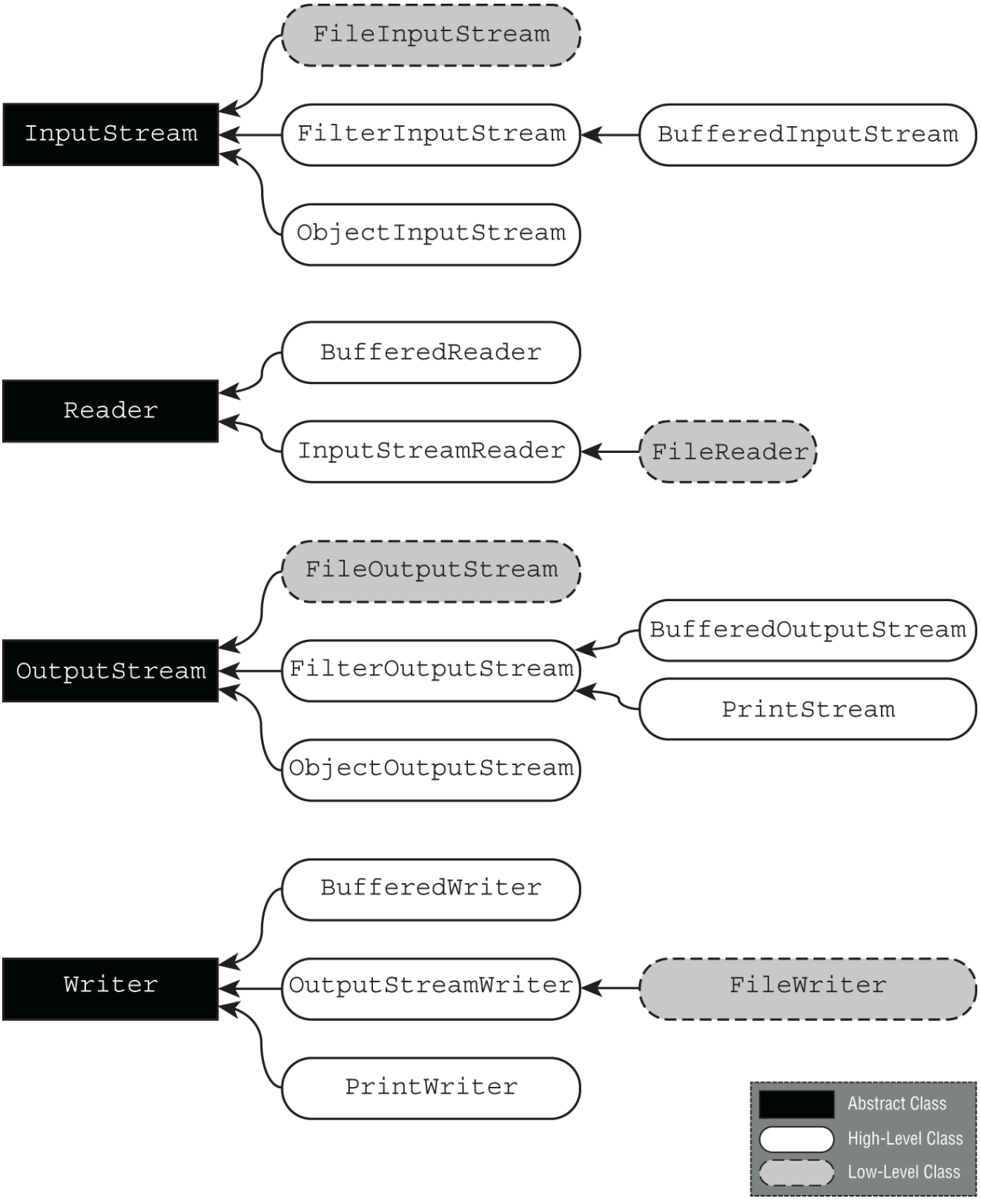
An older way of getting the contents of a directory is with `DirectoryStream`. The following gets all the files in a directory ending in `.txt` and `.java`.

```
try (DirectoryStream<Path> dirStream = Files
      .newDirectoryStream(path, "*.{txt,java}")) {
   for (Path entry: dirStream)
      System.out.println(entry);
}
```

Review of Key APIs

TABLE 14.13 Key APIs

Class	Purpose
`File`	I/O representation of location in file system
`Files`	Helper methods for working with `Path`
`Path`	NIO.2 representation of location in file system
`Paths`	Contains factory methods to get `Path`
`InputStream`	Superclass for reading files based on bytes
`OuputStream`	Superclass for writing files based on bytes
`Reader`	Superclass for reading files based on characters
`Writer`	Superclass for writing files based on characters


InputStream
FileInputStream
FilterInputStream
BufferedInputStream
ObjectInputStream
Reader
BufferedReader
InputStreamReader
FileReader
OutputStream
FileOutputStream
FilterOutputStream
BufferedOutputStream
PrintStream
ObjectOutputStream
Writer
BufferedWriter
OutputStreamWriter
FileWriter
PrintWriter
Abstract Class
High-Level Class
Low-Level Class