Chapter 14
IIO

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Using Java I/O API**
  - Read and write console and file data using I/O streams.
  - Serialize and de-serialize Java objects.
  - Construct, traverse, create, read, and write Path objects and their properties using the java.nio.file API.

What can Java applications do outside the scope of managing objects and attributes in memory? How can they save data so that information is not lost every time the program is terminated? They use files, of course! You can design code that writes the current state of an application to a file every time the application is closed and then reloads the data when the application is executed the next time. In this manner, information is preserved between program executions.

This chapter focuses on using I/O (input/output) and NIO.2 (non-blocking I/O) APIs to interact with files and I/O streams. The preferred approach for working with files and directories with newer software applications is to use NIO.2 rather than I/O where possible. However, you'll see that the two relate, and both are in wide use.

We start by describing how files and directories are organized within a file system and show how to access them with the `File` class and `Path` interface. Then we show how to work with files and directories. We conclude this chapter with advanced topics like serializing data, reading user input at runtime using the `Console` class, and interacting with file attributes.

NIO stands for non-blocking input/output API and is sometimes referred to as *new I/O*. The exam covers NIO version 2. There was a version 1 that covered channels, but it is not on the exam.

Referencing Files and Directories

We begin this chapter by reviewing what files and directories are within a file system. We also present the `File` class and `Path` interface along with how to create them.

Conceptualizing the File System

We start with the basics. Data is stored on persistent storage devices, such as hard disk drives and memory cards. A *file* within the storage device holds data. Files are organized into hierarchies using directories. A *directory* is a location that can contain files as well as other directories.

When working with directories in Java, we often treat them like files. In fact, we use many of the same classes and interfaces to operate on files and directories. For example, a file and directory both can be renamed with the same Java method. In this chapter, we often say *file* to mean *file or directory*.

To interact with files, we need to connect to the file system. The *file system* is in charge of reading and writing data within a computer. Different operating systems use different file systems to manage their data. For example, Windows-based systems use a different file system than Unix-based ones. For the exam, you just need to know how to issue commands using the Java APIs. The JVM will automatically connect to the local file system, allowing you to perform the same operations across multiple platforms.

Next, the *root directory* is the topmost directory in the file system, from which all files and directories inherit. In Windows, it is denoted with a drive

letter such as `C:\`, while on Linux, it is denoted with a single forward slash, `/`.

A *path* is a representation of a file or directory within a file system. Each file system defines its own path separator character that is used between directory entries. The value to the left of a separator is the parent of the value to the right of the separator. For example, the path value `/user/home/zoo.txt` means that the file `zoo.txt` is inside the `home` directory, with the `home` directory inside the `user` directory.

Operating System File Separators

Different operating systems vary in their format of pathnames. For example, Unix-based systems use the forward slash, `/`, for paths, whereas Windows-based systems use the backslash, `\`, character. That said, many programming languages and file systems support both types of slashes when writing path statements. Java offers a system property to retrieve the local separator character for the current environment:

```
System.out.print(System.getProperty("file.separator"));
```

We show how a directory and file system is organized in a hierarchical manner in Figure 14.1.

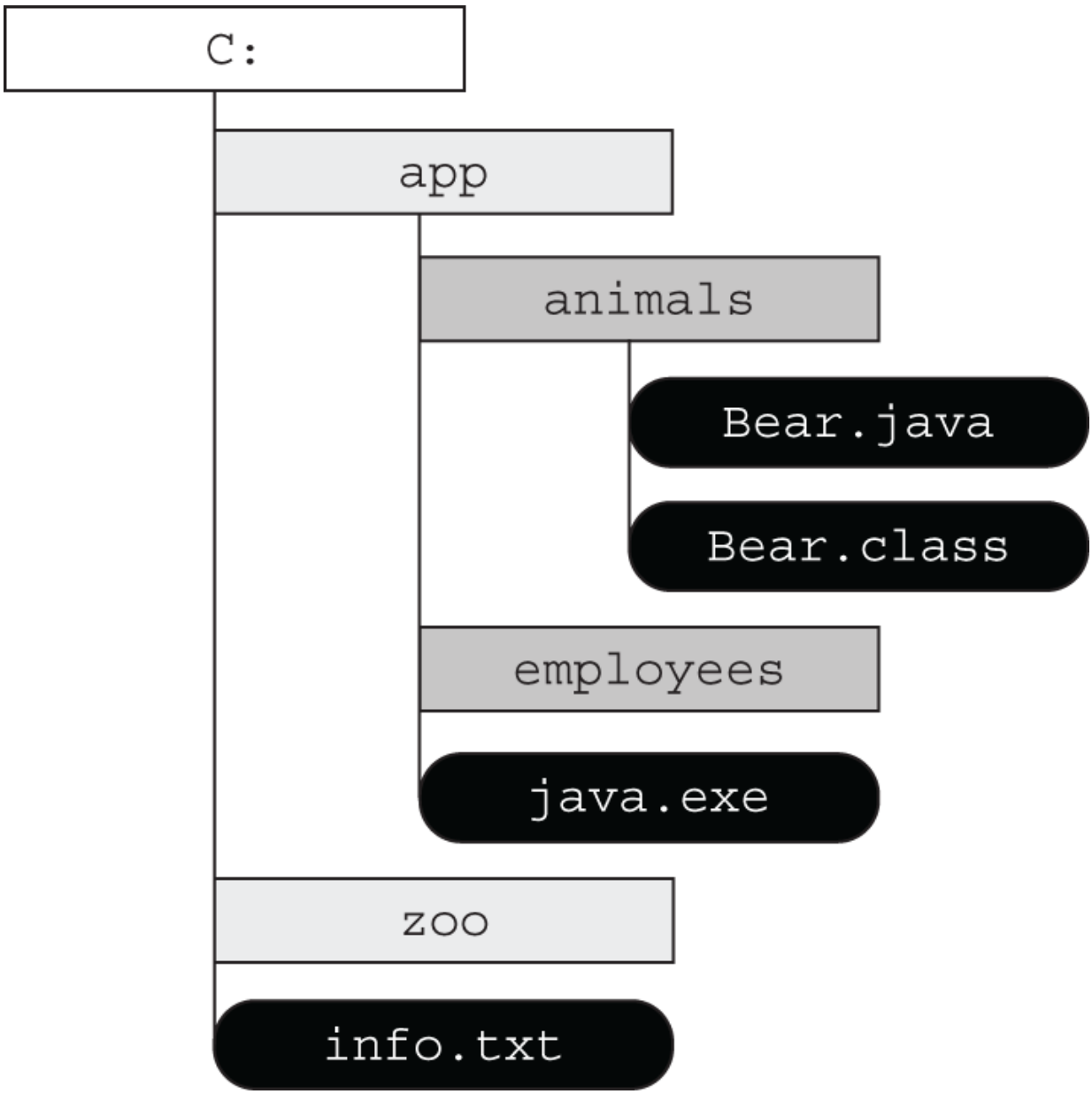

FIGURE 14.1 Directory and file hierarchy

This diagram shows the root directory, `C:`, as containing two directories, `app` and `zoo`, along with the file `info.txt`. Within the `app` directory, there are two more folders, `animals` and `employees`, along with the file `java.exe`. Finally, the `animals` directory contains two files, `Bear.java` and `Bear.class`.

We use both absolute and relative paths to the file or directory within the file system. The *absolute path* of a file or directory is the full path from the root directory to the file or directory, including all subdirectories that

contain the file or directory. Alternatively, the *relative path* of a file or directory is the path from the current working directory to the file or directory. For example, the following is an absolute path to the `Bear.java` file:

```
C:\app\animals\Bear.java
```

The following is a relative path to the same file, assuming the user's current directory is set to `C:\app`:

```
animals\Bear.java
```

Determining whether a path is relative or absolute is file system dependent. To match the exam, we adopt the following conventions:

- If a path starts with a forward slash (`/`), it is absolute, with `/` as the root directory, such as `/bird/parrot.png`.
- If a path starts with a drive letter (`C:`), it is absolute, with the drive letter as the root directory, such as `C :/bird/info`.
- Otherwise, it is a relative path, such as `bird/parrot.png`.

Absolute and relative paths can contain path symbols. A *path symbol* is one of a reserved series of characters with special meaning in some file systems. For the exam, there are two path symbols you need to know, as listed in Table 14.1.

TABLE 14.1 File system symbols

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

Looking at Figure 14.2, suppose the current directory is `/fish/shark/hammerhead`. In this case, `../swim.txt` is a valid relative path equivalent to `/fish/shark/swim.txt`. Likewise, `./play.png` refers to `play.png` in the current directory. These symbols can also be combined for greater effect. For example, `../../clownfish` is a relative path equivalent to `/fish/clownfish` within the file system.

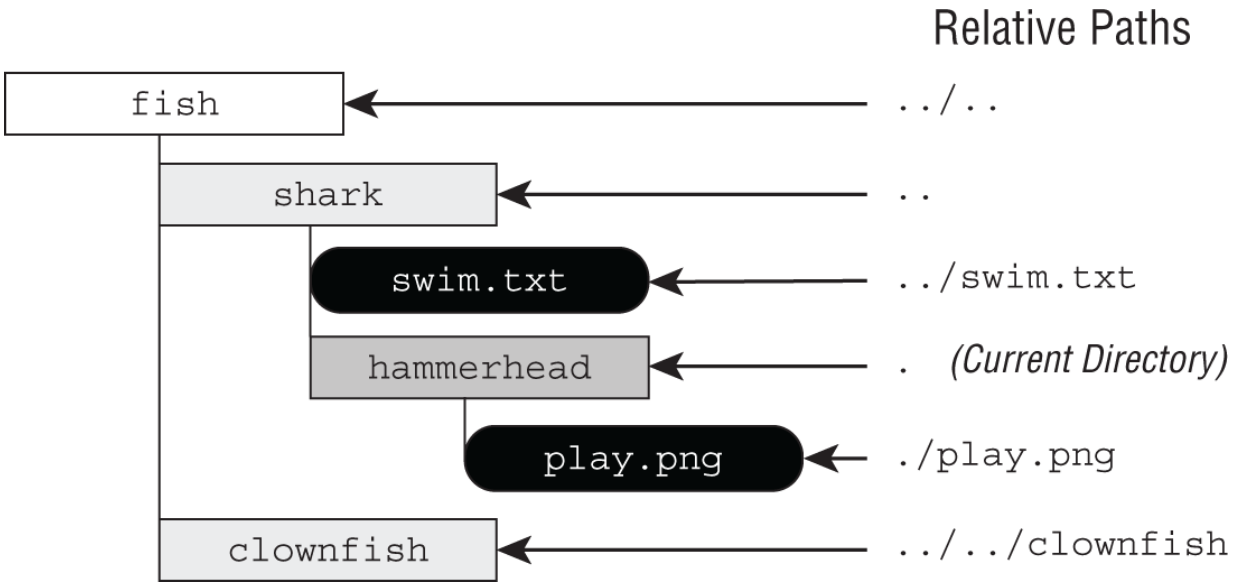

FIGURE 14.2 Relative paths using path symbols

Sometimes you'll see path symbols that are redundant or unnecessary. For example, the absolute path `/fish/clownfish/../shark/./swim.txt` can be simplified to `/fish/shark/swim.txt`. We see how to handle these redundancies later in the chapter when we cover `normalize()`.

A *symbolic link* is a special file within a file system that serves as a reference or pointer to another file or directory. Suppose we have a symbolic link from `/zoo/user/favorite` to `/fish/shark`. The `shark` folder and its elements can be accessed directly or via the symbolic link. For example, the following two paths reference the same file:

```
/fish/shark/swim.txt
/zoo/user/favorite/swim.txt
```

In general, symbolic links are transparent to the user, as the operating system takes care of resolving the reference to the actual file. While the I/O APIs do not support symbolic links, NIO.2 includes full support for creating, detecting, and navigating symbolic links within the file system.

Creating a *File* or *Path*

To do anything useful, you first need an object that represents the path to a particular file or directory on the file system. Using legacy I/O, this is the `java.io.File` class, whereas with NIO.2, it is the `java.nio.file.Path` interface. The `File` class and `Path` interface cannot read or write data within

a file, although they are passed as a reference to other classes, as you see in this chapter.

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

Creating a *File*

The `File` class is created by calling its constructor. This code shows three different constructors:

```
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. If we passed `null` as the parent to the final constructor, it would be ignored, and the method would behave the same way as the single `String` constructor. For fun, we also show how to tell if the file exists on the file system.

Creating a *Path*

Since `Path` is an interface, we can't create an instance directly. After all, interfaces don't have constructors! The most common way to create a `Path` is the following:

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

The following two examples reference the same file on disk:

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

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

The method allows passing a varargs parameter to pass additional path elements. The values are combined and automatically separated by the operating system–dependent file separator. We also show the `Files` helper class, which can check if the file exists on the file system. More on the `Files` class shortly!

Other Ways to Create a Path

There are three other ways to create a `Path` that might show up on the exam. An older way to create a `Path` is to use the `Paths` helper class.

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

This method has the exact same signature as the `Path.of()` method, which means it takes a varargs parameter. From the Javadoc, the `Paths` class may be deprecated in the future, so you should use `Path.of()` instead.

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.t
xt");
```

Finally, a `Path` can also be created using a uniform resource identifier (URI). It begins with a schema that indicates the resource type, followed by a path value, such as `file://` for local file systems and `https://` for remote file systems.

```
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. Luckily, Java makes this easy by providing methods to do just that:

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

Many older libraries use `File`, making it convenient to be able to get a `File` from a `Path` and vice versa. When working with newer applications, you should rely on NIO.2's `Path` interface, as it contains a lot more features.

Operating on *File* and *Path*

Now that we know how to create `File` and `Path` objects, we can start using them to do useful things. In this section, we explore the functionality available to us that involves directories.

Using Shared Functionality

Many operations can be done using both the I/O and NIO.2 libraries. We present many common APIs in Table 14.2 and Table 14.3. Although these tables may seem like a lot of methods to learn, many of them are self-explanatory. You can ignore the vararg parameters for now. We explain those later in the chapter.

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.C 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 `java.io.File` is the I/O class, while `Files` is an NIO.2 helper class. `Files` operates on `Path` instances, not `java.io.File` instances. We know this is confusing, but they are from completely different APIs!

Now let's try to use some of these APIs. The following is a sample program using only legacy I/O APIs. Given a file path, it outputs information about the file or directory, such as whether it exists, what files are contained within it, and so forth:

```
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 these examples, you see that the output of an I/O-based program is completely dependent on the directories and files available at runtime in the underlying file system.

On the exam, you might see paths that look like files but are directories or vice versa. For example, `/data/zoo.txt` could be a file or a directory, even though it has a file extension. Don't assume it is either unless the question tells you it is!

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.

Now, let's write that same program using only NIO.2 and see how it differs:

```
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`.

Second, lines 36–39 use a `Stream` and a lambda instead of a loop. Since streams use lazy evaluation, this means the method will load each path element as needed, rather than the entire directory at once.

Closing the Stream

Did you notice that in the last code sample, we put our `Stream` object inside a try-with-resources? 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.

If you assumed that a stream's terminal operation would automatically close the underlying file resources, you'd be wrong. There was a lot of debate about this behavior when it was first presented; in short, requiring developers to close the stream won out.

On the plus side, not all streams need to be closed: only those that open resources, like the ones found in NIO.2. For instance, you didn't need to close any of the streams you worked with in Chapter 10, "Streams."

Finally, the exam doesn't always properly close NIO.2 resources. To match the exam, we sometimes skip closing NIO.2 resources in review and practice questions. Always use try-with-resources statements with these NIO.2 methods in your own code.

For the remainder of this section, we only discuss the NIO.2 methods, because they are more important. There is also more to know about them, and they are more likely to come up on the exam.

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.

Methods that access or change files and directories, such as those in the `Files` class, often declare `IOException`. There are exceptions to this rule, as we will see. For example, the method `Files.exists()` does not declare `IOException`. If it did throw an exception when the file did not exist, it would never be able to return `false`! As a rule of thumb, if an NIO.2 method declares an `IOException`, it *usually* requires the paths it operates on to exist.

Providing NIO.2 Optional Parameters

Many of the NIO.2 methods in this chapter include a varargs that takes an optional list of values. Table 14.4 presents the arguments you should be familiar with for the exam.

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.

With the exceptions of `Files.copy()` and `Files.move()`, we won't discuss these varargs parameters each time we present a method. Their behavior should be straightforward, though. For example, can you figure out what the following call to `Files.exists()` with the `LinkOption` does in the following code snippet?

```
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.

Note that some of the enums in Table 14.4 inherit an interface. That means some methods accept a variety of enum types. For example, the `Files.move()` method takes a `CopyOption` vararg so it can take enums of different types, and more options can be added over time.

```
void move(Path source, Path target) throws IOException {
   Files.move(source, target,
      LinkOption.NOFOLLOW_LINKS,
      StandardCopyOption.ATOMIC_MOVE);
}
```

Interacting with NIO.2 Paths

Just like `String` values, `Path` instances are immutable. In the following example, the `Path` operation on the second line is lost since `p` is immutable:

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

Many of the methods available in the `Path` interface transform the path value in some way and return a new `Path` object, allowing the methods to be chained. We demonstrate chaining in the following example, the details of which we discuss in this section of the chapter:

```
Path.of("/zoo/../home").getParent().normalize().toAbsolutePath(
);
```

Viewing the Path

The `Path` interface contains three methods to retrieve basic information about the path representation. The `toString()` method returns a `String` representation of the entire path. In fact, it is the only method in the `Path` interface to return a `String`. Many of the other methods in the `Path` interface return `Path` instances.

The `getNameCount()` and `getName()` methods are often used together to retrieve the number of elements in the path and a reference to each element, respectively. These two methods do not include the root directory as part of the path.

```
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));
```

Notice that we didn't call `toString()` explicitly on the second line. Remember, Java calls `toString()` on any `Object` as part of string concatenation. We use this feature throughout the examples in this chapter.

The code prints the following:

```
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
```

Notice that if you try to call `getName()` with an invalid index, it will throw an exception at runtime.

Our examples print `/` as the file separator character because of the system we are using. Your actual output may vary throughout this chapter.

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`. This should sound familiar as it is how `String`'s `substring()` method works, as you saw in Chapter 4, "Core APIs."

The following code snippet shows how `subpath()` works. We also print the elements of the `Path` using `getName()` so that you can see how the indices are used.

```
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));
```

The output of this code snippet is the following:

```
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
```

Like `getNameCount()` and `getName()`, `subpath()` is zero-indexed and does not include the root. Also like `getName()`, `subpath()` throws an exception if invalid indices are provided.

```
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

The `Path` interface contains numerous methods for retrieving particular elements of a `Path`, returned as `Path` objects themselves. The `getFileName()` method returns the `Path` element of the current file or directory, while `getParent()` returns the full path of the containing directory. The `getParent()` method returns `null` if operated on the root path or at the top of a relative path. The `getRoot()` method returns the root element of the file within the file system, or `null` if the path is a relative path.

Consider the following method, which prints various `Path` elements:

```
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();
}
```

The `while` loop in the `printPathInformation()` method continues until `getParent()` returns `null`. We apply this method to the following three paths:

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

This sample application produces the following output:

```
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: .
```

Reviewing the sample output, you can see the difference in the behavior of `getRoot()` on absolute and relative paths. As you can see in the first and last examples, the `getParent()` method does not traverse relative paths outside the current working directory.

You also see that these methods do not resolve the path symbols and treat them as a distinct part of the path. While most of the methods in this part of the chapter treat path symbols as part of the path, we present one shortly that cleans up path symbols.

Resolving Paths

Suppose you want to concatenate paths in a manner similar to how we concatenate strings. 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`. Let's see what happens if we apply `resolve()` to an absolute path and a relative path:

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

The code snippet generates the following output:

```
/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");
System.out.println(path3.resolve("/tiger/cage"));
```

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

```
/tiger/cage
```

For the exam, you should be cognizant of mixing absolute and relative paths with the `resolve()` method. If an absolute path is provided as input to the method, that is the value returned. Simply put, you cannot combine two absolute paths using `resolve()`.

On the exam, when you see `resolve()`, think concatenation.

Relativizing a Path

The `Path` interface includes a `relativize()` method for constructing the relative path from one `Path` to another, often using path symbols. What do you think the following examples will print?

```
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));
```

The examples print the following:

```
../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.

```
var path1 = Path.of("/primate/chimpanzee");
var path2 = Path.of("bananas.txt");
path1.relativize(path2); // IllegalArgumentException
```

On Windows-based systems, it also requires that if absolute paths are used, both paths must have the same root directory or drive letter. For example, the following would also throw an `IllegalArgumentException` on a Windows-based system:

```
var path3 = Path.of("C:\\primate\\chimpanzee");
var path4 = Path.of("D:\\storage\\bananas.txt");
path3.relativize(path4); // IllegalArgumentException
```

Normalizing a Path

So far, we've presented a number of examples that included path symbols that were unnecessary. Luckily, Java provides the `normalize()` method to eliminate unnecessary redundancies in 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. Consider the following example:

```
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
```

The `equals()` method returns `true` if two paths represent the same value. In the first comparison, the path values are different. In the second comparison, the path values have both been reduced to the same normalized value, `/weather.txt`. This is the primary function of the `normalize()` method: to allow us to better compare different paths.

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.

Let's say that we have a file system in which we have a symbolic link from `/zebra` to `/horse`. What do you think the following will print, given a current working directory of `/horse/schedule`?

```
System.out.println(Path.of("/zebra/food.txt").toRealPath());
System.out.println(Path.of(".././food.txt").toRealPath());
```

The output of both lines is the following:

```
/horse/food.txt
```

In this example, the absolute and relative paths both resolve to the same absolute file, as the symbolic link points to a real file within the file system. We can also use the `toRealPath()` method to gain access to the current working directory as a `Path` object.

```
System.out.println(Path.of(".").toRealPath());
```

Reviewing NIO.2 Path APIs

We've covered a lot of instance methods on `Path` in this section. Table 14.5 lists them for review.

TABLE 14.5 `Path` API

Description	Path instance method
File path as string	`String` **`toString`**`()`
Single segment	`Path` **`getName`**`(int index)`
Number of segments	`int` **`getNameCount`**`()`
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 path	`Path` **`normalize`**`()`
Follow symbolic links to find path on file system	`Path` **`toRealPath`**`()`

Creating, Moving, and Deleting Files and Directories

Since creating, moving, and deleting have some nuance, we flesh them out in this section.

Making Directories

To create a directory, we use these `Files` methods:

```
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.

Both of these methods also accept an optional list of `FileAttribute<?>` values to apply to the newly created directory or directories. We discuss file attributes toward the end of the chapter.

The following shows how to create directories:

```
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

The `Files` class provides a method for copying files and directories within the file system.

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

The method copies a file or directory from one location to another using `Path` objects. The following shows an example of copying a file and a

directory:

```
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
   }
}
```

The method first copies the path, whether a file or a directory. If it is a directory, only a shallow copy is performed. Next, it checks whether the path is a directory and, if it is, performs a recursive copy of each of its elements.

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);
```

For the exam, you need to know that without the `REPLACE_EXISTING` option, this method will throw an exception if the file already exists.

Copying Files with IIO Streams

The `Files` class includes two `copy()` methods that operate 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.

The following are examples of each `copy()` method:

```
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);
```

While we used `FileInputStream` in the first example, the I/O stream could have been any valid I/O stream including website connections, in-memory stream resources, and so forth. The second example prints the contents of a file directly to the `System.out` stream.

Copying Files into a Directory

For the exam, it is important that you understand how the `copy()` method operates on both files and directories. 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);
```

If you said it would create a new file at `/enclosure/food.txt`, you're way off. 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.

This behavior applies to both the `copy()` and `move()` methods, the latter of which we cover next. In case you're curious, 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()*

The `Files` class provides a useful method for moving or renaming files and directories.

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

The following sample code uses the `move()` method:

```
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

Another enum value that you need to know for the exam when working with the `move()` method is the `StandardCopyOption` value `ATOMIC_MOVE`.

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

You may remember the atomic property from Chapter 13, "Concurrency," and the principle of an atomic move is similar. An atomic move is one in which a file is moved within the file system as a single indivisible operation. 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()*

The `Files` class includes two methods that delete a file or empty directory within the file system.

```
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 methods differ on how they handle a path that does not exist. 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.

Here we provide sample code that performs `delete()` operations:

```
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()*

Since a path may include path symbols and symbolic links within a file system, the `equals()` method can't be relied on to know if two `Path` instances refer to the same file. Luckily, there is the `isSameFile()` method. This method 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.

Assume that the file system exists, as shown in Figure 14.3, with a symbolic link from `/animals/snake` to `/animals/cobra`.

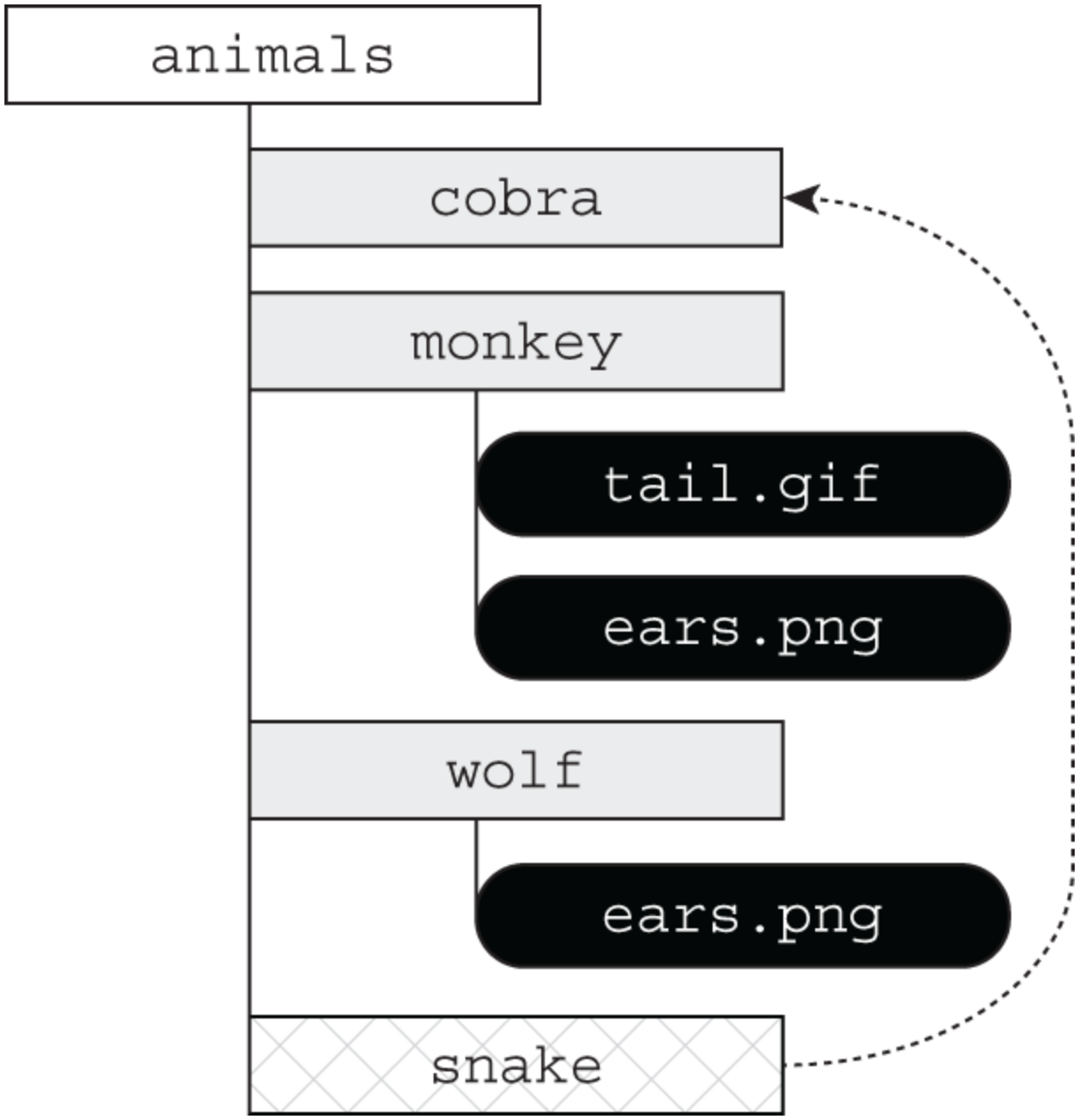

FIGURE 14.C Comparing file uniqueness

Given the structure defined in Figure 14.3, what does the following output?

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


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

Since `snake` is a symbolic link to `cobra`, the first example outputs `true`. In the second example, the paths refer to different files, so `false` is printed.

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. Given those values, what do you think this code prints?

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

The answer is the same as the previous example. The code prints `1` again. The `mismatch()` method is symmetric and returns the same result regardless of the order of the parameters.

Introducing IIO Streams

Now that we have the basics out of the way, let's move on to I/O streams, which are far more interesting. In this section, we show you how to use I/O streams to read and write data. The "I/O" refers to the nature of how data is accessed, either by reading the data from a resource (input) or by writing the data to a resource (output).

When we refer to *I/O streams* in this chapter, we are referring to the ones found in the `java.io` API. If we just say *streams*, it means the ones from Chapter 10. We agree that the naming can be a bit confusing!

Understanding IIO Stream Fundamentals

The contents of a file may be accessed or written via an I/O *stream*, which is a list of data elements presented sequentially. An I/O stream can be conceptually thought of as a long, nearly never-ending stream of water with data presented one wave at a time.

We demonstrate this principle in Figure 14.4. The I/O stream is so large that once we start reading it, we have no idea where the beginning or the end is. We just have a pointer to our current position in the I/O stream and read data one block at a time.

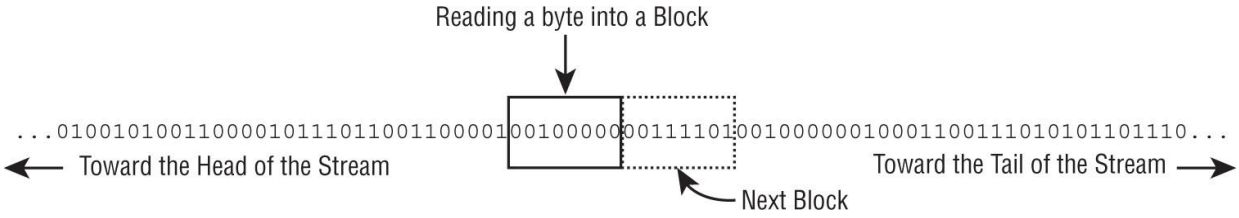

FIGURE 14.4 Visual representation of an I/O stream

Each type of I/O stream segments data into a wave or block in a particular way. For example, some I/O stream classes read or write data as individual bytes. Other I/O stream classes read or write individual characters or strings of characters. On top of that, some I/O stream classes read or write larger groups of bytes or characters at a time, specifically those with the word `Buffered` in their name.

Although the `java.io` API is full of I/O streams that handle characters, strings, groups of bytes, and so on, nearly all are built on top of reading or writing an individual byte or an array of bytes at a time. Higher-level I/O streams exist for convenience as well as performance.

Although I/O streams are commonly used with file I/O, they are more generally used to handle the reading/writing of any sequential data source. For example, you might construct a Java application that submits data to a website using an output stream and reads the result via an input stream.

IIO Streams Can Be Big

When writing code where you don't know what the I/O stream size will be at runtime, it may be helpful to visualize an I/O stream as being so large that all of the data contained in it could not possibly fit into memory. For example, a 1 TB file could not be stored entirely in memory by most computer systems (at the time this book is being written). The file can still be read and written by a program with very little memory, since the I/O stream allows the application to focus on only a small portion of the overall I/O stream at any given time.

Learning IIO Stream Nomenclature

The `java.io` API provides numerous classes for creating, accessing, and manipulating I/O streams—so many that it tends to overwhelm many new Java developers. Stay calm! We review the major differences between each I/O stream class and show you how to distinguish between them.

Even if you come across a particular I/O stream on the exam that you do not recognize, the name of the I/O stream often gives you enough information

to understand exactly what it does.

The goal of this section is to familiarize you with common terminology and naming conventions used with I/O streams. Don't worry if you don't recognize the particular stream class names used in this section or their function; we cover how to use them in detail in this chapter.

Storing Data as Bytes

Data is stored in a file system (and memory) as a `0` or `1`, called a *bit*. Since it's really hard for humans to read/write data that is just `0`s and `1`s, they are grouped into a set of 8 bits, called a *byte*.

What about the Java `byte` primitive type? As you learn later, when we use I/O streams, values are often read or written using `byte` values and arrays.

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 IIO 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.

It is important to remember that even though character I/O streams do not contain the word `Stream` in their class name, they are still I/O streams. The use of `Reader`/`Writer` in the name is just to distinguish them from byte streams.

Throughout the chapter, we refer to both `InputStream` and `Reader` as *input streams*, and we refer to both `OutputStream` and `Writer` as *output streams*.

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. For example, you can use a `Writer` class to output a `String` value to a file without necessarily having to worry about the underlying character encoding of the file.

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. Although this may sound simple, Java supports a wide variety of character encodings, ranging from ones that may use one byte for Latin characters, `UTF-8` and `ASCII` for example, to using two or more bytes per character, such as `UTF-16`. For the exam, you don't need to memorize the character encodings, but you should be familiar with the names.

Character Encoding in Java

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");
```

Java supports numerous character encodings, each specified by a different standard name value.

Input vs. Output Streams

Most `InputStream` classes have a corresponding `OutputStream` class, and vice versa. For example, the `FileOutputStream` class writes data that can be read by a `FileInputStream`. If you understand the features of a particular `Input` or `Output` stream class, you should naturally know what its complementary class does.

It follows, then, that most `Reader` classes have a corresponding `Writer` class. For example, the `FileWriter` class writes data that can be read by a `FileReader`.

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. We discuss these classes later in this chapter.

Low-Level vs. High-Level Streams

Another way that you can familiarize yourself with the `java.io` API is by segmenting I/O streams into low-level and 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.

Alternatively, 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. For example, take a look at the `FileReader` and `BufferedReader` objects in the following sample code:

```
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.

For the exam, the only low-level stream classes you need to be familiar with are the ones that operate on files. The rest of the nonabstract stream classes are all high-level streams.

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
```

The first two examples do not compile because they mix `Reader`/`Writer` classes with `InputStream`/`OutputStream` classes, respectively. The third example does not compile because we are mixing an `OutputStream` with an `InputStream`. Although it is possible to read data from an `InputStream` and write it to an `OutputStream`, wrapping the I/O stream is not the way to do so. As you see later in this chapter, the data must be copied over. Finally, the last example does not compile because `InputStream` is an abstract class, and therefore you cannot create an instance of it.

Like we saw with input vs output streams, there are a few exceptions. `InputStreamReader` is a `Reader` that takes an `InputStream`, while `OutputStreamWriter` is a `Writer` that takes an `OutputStream`. While unlikely to be on the exam, these convenience classes allow you to convert one stream type to another, which is why they have both types in their names. The `PrintWriter` class is also special in that it can take an `OutputStream` or a `Writer`. We'll cover `PrintWriter` more later in the chapter.

Decoding IIO Class Names

Pay close attention to the name of the I/O class on the exam, as decoding it often gives you context clues as to what the class does. For example, without needing to look it up, it should be clear that `FileReader` is a class that reads data from a file as characters or strings. Furthermore, `ObjectOutputStream` sounds like a class that writes object data to a byte stream.

Table 14.6 lists the abstract base classes that all I/O streams inherit from.

TABLE 14.6 The `java.io` abstract stream base classes

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

Table 14.7 lists the concrete I/O streams that you should be familiar with for the exam. Note that most of the information about each I/O stream, such as whether it is an input or output stream or whether it accesses data using bytes or characters, can be decoded by the name alone.

TABLE 14.7 The `java.io` concrete I/O stream classes

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
`PrintWriter`	High	Writes formatted representations of Java objects to character stream

Keep Table 14.6 and Table 14.7 handy as you learn more about I/O streams in this chapter. We discuss these in more detail, including examples of each.

Reading and Writing Files

There are a number of ways to read and write from a file. We show them in this section by copying one file to another.

Using IIO 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.

The following `copyStream()` methods show an example of reading all of the values of an `InputStream` and `Reader` and writing them to an `OutputStream` and `Writer`, respectively. In both examples, `-1` is used to indicate the end of 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);
   }
}
```

Hold on. We said we are reading and writing bytes, so why do the methods use `int` instead of `byte`? Remember, the `byte` data type has a range of 256 characters. They needed an extra value to indicate the end of an I/O stream. The authors of Java decided to use a larger data type, `int`, so that special values like `-1` would indicate the end of an I/O stream. The output stream classes use `int` as well, to be consistent with the input stream classes.

Reading and writing one byte at a time isn't a particularly efficient way of doing this. Luckily, there are overloaded methods for reading and writing multiple bytes at a time. 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:     }
```

Instead of reading the data one byte at a time, we read and write up to `1024` bytes at a time on lines 14–15. The return value `lengthRead` is critical for determining whether we are at the end of the stream and knowing how many bytes we should write into our output stream.

Unless our file happens to be a multiple of `1024` bytes, the last iteration of the `while` loop will write some value less than `1024` bytes. For example, if the buffer size is 1,024 bytes and the file size is 1,054 bytes, the last read will be only 30 bytes. If we ignored this return value and instead wrote 1,024 bytes, 994 bytes from the previous loop would be written to the end of the file.

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. It is not without cost, though. Each time it is used, it may cause a noticeable delay in the application, especially for large files. Unless the data that you are writing is extremely critical, the `flush()` method should be used only intermittently. For example, it should not necessarily be called after every write, as it is in this example.

Equivalent methods exist on `Reader` and `Writer`, but they use `char` rather than `byte`, making the equivalent `copyStream()` method very similar.

The previous example makes reading and writing a file look like a lot to think about. That's because it uses only low-level I/O streams. Let's try again using high-level streams.

```
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 also chose to use a `BufferedReader` and `BufferedWriter` so we can read a whole line at a time. This gives us the benefits of reading batches of characters on line 30 without having to write custom logic. Line 31 writes out the whole line of data at once. Since reading a line strips the line breaks, we add those back on line 32. Lines 27 and 28 demonstrate chaining constructors. The try-with-resources takes care of closing all the objects in the chain.

Now imagine that we wanted `byte` data instead of characters. We would need to choose different high-level classes: `BufferedInputStream`, `BufferedOutputStream`, `FileInputStream`, and `FileOuputStream`. We would call `readAllBytes()` instead of `readLine()` and store the result in a

`byte[]` instead of a `String`. Finally, we wouldn't need to handle new lines since the data is binary.

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);
      }
}
```

While we used a `String`, there are numerous overloaded versions of `println()`, which take everything from primitives and `String` values to objects. Under the covers, these methods often just perform `String.valueOf()`.

The print stream classes have the distinction of being the only I/O stream classes we cover that do not have corresponding input stream classes. And unlike other `OutputStream` classes, `PrintStream` does not have `Output` in its name.

It may surprise you that you've been regularly using a `PrintStream` throughout this book. Both `System.out` and `System.err` are `PrintStream` objects. Likewise, `System.in`, often useful for reading user input, is an `InputStream`.

Unlike the majority of the other I/O streams we've covered, the methods in the print stream classes do not throw any checked exceptions. If they did,

you would be required to catch a checked exception any time you called `System.out.print()`!

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*

The NIO.2 APIs provide even easier ways to read and write a file using the `Files` class. Let's start by looking at three ways of copying a file by reading in the data and writing it back:

```
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. Assuming that the input file `sharks.log` is as follows:

```
INFO:Server starting
DEBUG:Processes available = 10
WARN:No database could be detected
DEBUG:Processes available reset to 0
WARN:Performing manual recovery
INFO:Server successfully started
```

Then the sample output would be the following:

```
No database could be detected
Performing manual recovery
```

As you can see, we have the ability to manipulate files in complex ways, often with only a few short expressions.

Files.readAllLines() vs. *Files.lines()*

For the exam, you need to know the difference between `readAllLines()` and `lines()`. Both of these examples compile and run.

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

Files.lines(Path.of("birds.txt")).forEach(System.out::printl
n);
```

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.

You should also be aware of when they are mixing incompatible types on the exam. Do you see why the following does not compile?

```
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()*

Sometimes you need to mix I/O streams and NIO.2. Conveniently, `Files` includes two convenience methods for getting I/O streams.

```
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.

Reviewing Common Read and Write Methods

Table 14.8 reviews the `public` common I/O stream methods you should know for reading and writing. We also include `close()` and `flush()` since they are used when performing these actions. Table 14.9 does the same for common `public` NIO.2 read and write methods.

TABLE 14.8 Common I/O read and write instance methods

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)`	Reads up to `length` values into buffer starting from position `offset` and returns number of bytes or characters read.
`Reader`	`int` **`read`**`(char[] c, int offset, int length)`	
`All output streams`	`void` **`write`**`(int b)`	Writes single byte.
`OutputStream`	`void` **`write`**`(byte[] b)`	Writes array of values into stream.
`Writer`	`void` **`write`**`(char[] c)`	
`OutputStream`	`void` **`write`**`(byte[] b, int offset, int length)`	Writes `length` values from array into stream, starting with `offset` index.
`Writer`	`void` **`write`**`(char[] c, int offset, int length)`	

Class	Method name	Description
`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

Throughout this book, we have been managing our data model using classes, so it makes sense that we would want to save these objects between program executions. Data about our zoo animals' health wouldn't be particularly useful if it had to be entered every time the program runs!

You can certainly use the I/O stream classes you've learned about so far to store text and binary data, but you still have to figure out how to put the data in the I/O stream and then decode it later. There are various file

formats like XML and CSV you can standardize to, but you often have to build the translation yourself.

Alternatively, we can use serialization to solve the problem of how to convert objects to/from an I/O stream. *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.

Figure 14.5 shows a visual representation of serializing and deserializing a `Giraffe` object to and from a `giraffe.ser` file.

A process flow diagram of serializing and deserializing process. The serialization denotes file system of object giraffe.ser and deserialization denotes Java virtual machine of giraffe object.

FIGURE 14.5 Serialization process

In this section, we show you how Java provides built-in mechanisms for serializing and deserializing I/O streams of objects directly to and from disk, respectively.

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.

Since `Serializable` is a marker interface with no `abstract` members, why not just apply it to every class? Generally speaking, you should only mark data-oriented classes serializable. Process-oriented classes, such as the I/O streams discussed in this chapter or the `Thread` instances you learned about in Chapter 13, are often poor candidates for serialization, as the internal state of those classes tends to be ephemeral or short-lived.

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. All Java primitives and many of the built-in Java classes that you have worked with throughout this book are `Serializable`. For example, this class can be serialized:

```
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
}
```

In this example, the `Gorilla` class contains three instance members (`name`, `age`, `friendly`) that will be saved to an I/O stream if the class is serialized. Note that since `Serializable` is not part of the `java.lang` package, it must be imported or referenced with the package name.

What about the `favoriteFood` field that is marked `transient`? Any field that is marked `transient` will not be saved to an I/O stream when the class is serialized. We discuss that in more detail next.

Real World Scenario

Maintaining a *serialVersionUID*

It's a good practice to declare a `static serialVersionUID` variable in every class that implements `Serializable`. The version is stored with each object as part of serialization. Then, every time the class structure changes, this value is updated or incremented.

Perhaps our `Gorilla` class receives a new instance member `Double banana`, or maybe the `age` field is renamed. The idea is a class could have been serialized with an older version of the class and deserialized with a newer version of the class.

The `serialVersionUID` helps inform the JVM that the stored data may not match the new class definition. If an older version of the class is encountered during deserialization, a `java.io.InvalidClassException` may be thrown. Alternatively, some APIs support converting data between versions.

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. You see examples of this shortly when we present the object stream classes.

Marking `static` fields `transient` has little effect on serialization. Other than the `serialVersionUID`, only the instance members of a class are serialized. A `static` variable keeps it state as long as the JVM is running, so `transient` is ignored.

Ensuring That a Class Is Serializable

Since `Serializable` is a marker interface, you might think there are no rules to using it. Not quite! Any process attempting to serialize an object will throw a `NotSerializableException` if the class does not implement the `Serializable` interface properly.

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.

Be careful with the second rule. For a class to be serializable, we must apply the second rule recursively. Do you see why the following `Cat` class is not serializable?

```
public class Cat implements Serializable {
   private Tail tail = new Tail();
}

public class Tail implements Serializable {
   private Fur fur = new Fur();
}

public class Fur {}
```

`Cat` contains an instance of `Tail`, and both of those classes are marked `Serializable`, so no problems there. Unfortunately, `Tail` contains an instance of `Fur` that is not marked `Serializable`.

Either of the following changes fixes the problem and allows `Cat` to be serialized:

```
public class Tail implements Serializable {
   private transient Fur fur = new Fur();
}

public class Fur implements Serializable {}
```

We could also make our `tail` or `fur` instance members `null`, although this would make `Cat` serializable only for particular instances, rather than all instances.

Serializing Records

Do you think this record is serializable?

```
record Record(String name) {}
```

It is not serializable because it does not implement `Serializable`. A record follows the same rules as other types of classes with respect to whether it can be serialized. Therefore, this one can be:

```
record Record(String name) implements Serializable {}
```

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. While both of these classes contain a number of methods for built-in data types like primitives, the two methods you need to know for the exam are the ones related to working with objects.

```
// ObjectInputStream
public Object readObject() throws IOException,
ClassNotFoundException
```

```
// ObjectOutputStream
public void writeObject(Object obj) throws IOException
```

Note the parameters, return types, and exceptions thrown. We now provide a sample method that serializes a `List` of `Gorilla` objects to a file:

```
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);
   }
}
```

Pretty easy, right? Notice that we start with a file stream, wrap it in a buffered I/O stream to improve performance, and then wrap that with an object stream. Serializing the data is as simple as passing it to `writeObject()`.

Once the data is stored in a file, we can deserialize it by using the following method:

```
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;
}
```

Ah, not as simple as our save method, was it? When calling `readObject()`, `null` and `-1` do not have any special meaning, as someone might have serialized objects with those values. Unlike our earlier techniques for reading methods from an input stream, we need to use an infinite loop to

process the data, which throws an `EOFException` when the end of the I/O stream is reached.

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.

Since the return type of `readObject()` is `Object`, we need to check the type before obtaining access to our `Gorilla` properties. Notice that `readObject()` declares a checked `ClassNotFoundException` since the class might not be available on deserialization.

The following code snippet shows how to call the serialization methods:

```
var gorillas = new ArrayList<Gorilla>();
gorillas.add(new Gorilla("Grodd", 5, false));
gorillas.add(new Gorilla("Ishmael", 8, true));
File dataFile = new File("gorilla.ser");

saveToFile(gorillas, dataFile);
var gorillasFromDisk = readFromFile(dataFile);
System.out.print(gorillasFromDisk);
```

Assuming that the `toString()` method was properly overridden in the `Gorilla` class, this prints the following at runtime:

```
[[name=Grodd, age=5, friendly=false],
 [name=Ishmael, age=8, friendly=true]]
```

`ObjectInputStream` inherits an `available()` method from `InputStream` that you might think can be used to check for the end of the I/O stream rather than throwing an `EOFException`. Unfortunately, this only tells you the number of blocks that can be read without blocking another thread. In other words, it can return `0` even if there are more bytes to be read.

Understanding the Deserialization Creation Process

For the exam, you need to understand how a deserialized object is created. When you deserialize an object, *the constructor of the serialized class, along with any instance initializers, is not called when the object is created*. Java will call the no-arg constructor of the first nonserializable parent class it can find in the class hierarchy. In our `Gorilla` example, this would just be the no-arg constructor of `Object`.

As we stated earlier, 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.

Let's take a look at a new `Chimpanzee` class which has a parent class that is not `Serializable`. This time we do list the constructors to illustrate which is used on deserialization.

```
// Mammal.java
public class Mammal {
   private int id;

   public Mammal() {
      this.id = 4;
   }

   // Getters/Setters/toString() omitted
}

// Chimpanzee.java
```

```
import java.io.Serializable;
public class Chimpanzee extends Mammal implements Serializable
{
   private static final long serialVersionUID = 2L;
   private transient String name;
   private transient int age = 10;
   private static char type = 'C';
   { this.age = 14; }

   public Chimpanzee() {
      this("Unknown", 12, 'Q');
   }

   public Chimpanzee(String name, int age, char type) {
      this.name = name;
      this.age = age;
      this.type = type;
      setId(9);
   }

   // Getters/Setters/toString() omitted
}
```

Assuming we rewrite our previous serialization and deserialization methods to process a `Chimpanzee` object instead of a `Gorilla` object, what do you think the following prints?

```
var chimpanzees = List.of(new Chimpanzee("Ham", 2, 'A'),
   new Chimpanzee("Enos", 3, 'B'));
File dataFile = new File("chimpanzee.ser");
System.out.println("Original:   " + chimpanzees);

saveToFile(chimpanzees, dataFile);
var chimpanzeesFromDisk = readFromFile(dataFile);
System.out.println("From Disk: " + chimpanzeesFromDisk);
```

Think about it. Go on, we'll wait.

Ready for the answer? Well, for starters, none of the instance members is serialized to a file. The `name` and `age` variables are both marked `transient`, while the `type` variable is `static`. We purposely accessed the `type` variable using `this` to see whether you were paying attention.

Upon deserialization, none of the constructors in `Chimpanzee` is called. Even the no-arg constructor that sets the values [`name=Unknown, age=12,`

`type=Q, id=8`] is ignored. The instance initializer that sets `age` to `14` is also not executed.

In this case, the `name` variable is initialized to `null` since that's the default value for `String` in Java. Likewise, the `age` variable is initialized to `0`.

However, the superclass of `Mammal` is not `Serializable` so the default constructor is called and `id` is set to `4`. The program prints the following, assuming the `toString()` method is implemented:

```
Original:  [[name=Ham, age=2, type=B, id=9,
   [name=Enos, age=3, type=B, id=9]

From Disk: [[name=null, age=0, type=B, id=4,
   [name=null, age=0, type=B, id=4]
```

What about the `type` variable? Since it's `static`, it will display whatever value was set last. If the data is serialized and deserialized within the same execution, it will display `B`, since that was the last `Chimpanzee` we created. On the other hand, if the program performs the deserialization and print on startup, it will print `C`, since that is the value the class is initialized with.

For the exam, make sure you understand that the constructor and any instance initializations defined in the serialized class are ignored during the deserialization process. Java only calls the constructor of the first non-serializable parent class in the class hierarchy.

Finally, let's add a subclass:

```
public class BabyChimpanzee extends Chimpanzee {
   private static final long serialVersionUID = 3L;

   private String mother = "Mom";

   public BabyChimpanzee() { super(); }

   public BabyChimpanzee(String name, char type) {
      super(name, 0, type);
   }
   // Getters/Setters/toString() omitted
}
```

Notice that this subclass `BabyChimpanzee` is serializable because the superclass `Chimpanzee` has implemented `Serializable`. We now have an

additional instance variable. The code to serialize and deserialize remains the same. We can even still cast to `Chimpanzee` because this is a subclass.

Customizing Serialization

Normally, the rules that you get out of the box for serialization are good enough. However, you can control the serialization process in finer grained detail.

To customize serialization, just define `private` methods in your `Serializable` class with specific signatures. These methods are not actually overrides, but are called as part of the serialization process automatically. It is common to start by calling `defaultReadObject()` at the beginning of the `readObject()` method, and `defaultWriteObject()` at the end of the `writeObject()` method. These handle the default serialization behaviors.

```
public class Fish implements Serializable {
   private String name;
   private transient int fins;
   …
   private void readObject(ObjectInputStream in)
         throws ClassNotFoundException, IOException {
      in.defaultReadObject();

      // custom logic
      this.fins = 10;
   }
   private void writeObject(ObjectOutputStream out)
         throws IOException {
      // custom logic
      this.name = "Nemo";

      out.defaultWriteObject();
   } }
```

In this example, the `transient` variable `fins` is given a default value of `10` anytime it is read from disk. When writing to disk, the `name` value is replaced within the object with `"Nemo"`, which is then saved to disk.

Interacting with Users

Java includes numerous classes for interacting with the user. For example, you might want to write an application that asks a user to log in and then prints a success message. This section contains numerous techniques for handling and responding to user input.

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.

```
try (var in = new FileInputStream("zoo.txt")) {
   System.out.println("Found file!");
} catch (FileNotFoundException e) {
   System.err.println("File not found!");
}
```

How do they differ in practice? In part, that depends on what is executing the program. For example, if you are running from a command prompt, they will likely print text in the same format. On the other hand, if you are working in an integrated development environment (IDE), they might print the `System.err` text in a different color. Finally, if the code is being run on a server, the `System.err` stream might write to a different log file.

Real World Scenario

Using Logging APIs

While `System.out` and `System.err` are incredibly useful for debugging stand-alone or simple applications, they are rarely used in professional software development. Most applications rely on a logging service or API.

While many logging APIs are available, they tend to share a number of similar attributes. First you create a `static` logging object in each class. Then you log a message with an appropriate logging level: `debug()`, `info()`, `warn()`, or `error()`. The `debug()` and `info()` methods are useful as they allow developers to log things that aren't errors but may be useful. For example:

```
var logger = Logger.getLogger("errors");
logger.info("Code is running");
logger.warning("Code shouldn't have done that");
```

You can also use the `log()` method and provide the level programmatically:

```
logger.log(Level.SEVERE, "You should worry");
```

Reading Input as an IIO 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.

Closing System Streams

You might have noticed that we never created or closed `System.out`, `System.err`, and `System.in` when we used them. In fact, these are the only I/O streams in the entire chapter that we did not use a try-with-resources block on!

Because these are `static` objects, the `System` streams are shared by the entire application. The JVM creates and opens them for us. They can be used in a try-with-resources statement or by calling `close()`, although *closing them is not recommended*. Closing the `System` streams makes them permanently unavailable for all threads in the remainder of the program.

What do you think the following code snippet prints?

```
try (var out = System.out) {}
System.out.println("Hello");
```

Nothing. It prints nothing. The methods of `PrintStream` do not throw any checked exceptions and rely on the `checkError()` to report errors, so they fail silently.

What about this example?

```
try (var err = System.err) {}
System.err.println("Hello");
```

This one also prints nothing. Like `System.out`, `System.err` is a `PrintStream`. Even if it did throw an exception, we'd have a hard time seeing it since our I/O stream for reporting errors is closed! Closing `System.err` is a particularly bad idea, since the stack traces from all exceptions will be hidden.

Finally, what do you think this code snippet does?

```
var reader = new BufferedReader(new
InputStreamReader(System.in));
try (reader) {}
String data = reader.readLine();  // IOException
```

It prints an exception at runtime. Unlike the `PrintStream` class, most `InputStream` implementations will throw an exception if you try to operate on a closed I/O stream.

Acquiring Input with *Console*

The `java.io.Console` class is specifically designed to handle user interactions. After all, `System.in` and `System.out` are just raw streams, whereas `Console` is a class with numerous methods centered around user input.

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. For example, if you come across code on the exam such as the following, it does not compile, since the constructors are all `private`:

```
Console c = new Console();  // DOES NOT COMPILE
```

The following snippet shows how to obtain a `Console` and use it to retrieve user input:

```
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");
}
```

The `Console` object may not be available, depending on where the code is being called. If it is not available, `System.console()` returns `null`. It is imperative that you check for a `null` value before attempting to use a `Console` object!

This program first retrieves an instance of the `Console` and verifies that it is available, outputting a message to `System.err` if it is not. If it is available, the program retrieves a line of input from the user and prints the result. The `flush()` is to ensure the output gets written. As you might have noticed, this example is similar to our earlier example of reading user input with `System.in` and `System.out`.

Obtaining Underlying IIO Streams

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

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

Accessing these classes is analogous to calling `System.in` and `System.out` directly, although they use character streams rather than byte streams. In this manner, they are more appropriate for handling text data.

Formatting Console Data

In Chapter 4, you learned about the `format()` method on `String`; and in Chapter 11, "Exceptions and Localization," you worked with formatting using locales. Conveniently, each print stream class includes a `format()` method, which includes an overloaded version that takes a `Locale` to combine both of these:

```
// 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)
```

For convenience (as well as to make C developers feel more at home), Java includes `printf()` methods, which function identically to the `format()` methods. The only thing you need to know about these methods is that they are interchangeable with `format()`.

Let's take a look at using multiple methods to print information for the user:

```
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();
}
```

Assuming the `Console` is available at runtime, it prints the following:

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

`PrintStream` and `PrintWriter` are similar, as they both support a lot of the same overridden methods. One difference is that `PrintStream` will convert characters into bytes using the provided (or default charset) encoding. Also, only `PrintStream` supports working directly with the raw bytes.

Reading Console Data

The `Console` class includes four methods for retrieving regular text data from the user.

```
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`.

The first feature improves security by not showing the password on the screen if someone happens to be sitting next to you. The second feature involves preventing passwords from entering the `String` pool.

Reviewing Console Methods

The last code sample we present asks the user a series of questions and prints results based on this information using many of various methods we learned in this section:

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
   String name = console.readLine("Please enter your name: ");
   console.writer().format("Hi %s", name);
   console.writer().println();

   console.format("What is your address? ");
   String address = console.readLine();
```

```
   char[] password = console.readPassword("Enter a password "
      + "between %d and %d characters: ", 5, 10);
   char[] verify = console.readPassword("Enter the password
again: ");
   console.printf("Passwords "
      + (Arrays.equals(password, verify) ? "match" : "do not
match"));
   console.flush();
}
```

Assuming the `Console` is available, the output should resemble the following:

```
Please enter your name: Max
Hi Max
What is your address? Spoonerville
Enter a password between 5 and 10 characters:
Enter the password again:
Passwords match
```

Working with Advanced APIs

Files, paths, I/O streams: you've worked with a lot this chapter! In this final section, we cover some advanced features of I/O streams and NIO.2 that can be quite useful in practice—and have been known to appear on the exam from time to time!

Manipulating Input Streams

All input stream classes include the following methods to manipulate the order in which data is read from an I/O stream:

```
// 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.

Not all input stream classes support `mark()` and `reset()`. Make sure to call `markSupported()` on the I/O stream before calling these methods, or an exception will be thrown at runtime.

Marking Data

Assume that we have an `InputStream` instance whose next values are `LION`. Consider the following code snippet:

```
public void readData(InputStream is) throws IOException {
   System.out.print((char) is.read());     // L
   if (is.markSupported()) {
      is.mark(100);  // Marks up to 100 bytes
      System.out.print((char) is.read());  // I
      System.out.print((char) is.read());  // O
      is.reset();    // Resets stream to position before I
   }
   System.out.print((char)  is.read());   //  I
   System.out.print((char)  is.read());   //  O
   System.out.print((char) is.read());    // N
}
```

The code snippet will output `LIOION` if `mark()` is supported and `LION` otherwise. It's a good practice to organize your `read()` operations so that the I/O stream ends up at the same position regardless of whether `mark()` is supported.

What about the value of `100` that we passed to the `mark()` method? This value is called the `readLimit`. It instructs the I/O stream that we expect to call `reset()` after at most `100` bytes. If our program calls `reset()` after reading more than `100` bytes from calling `mark(100)`, it may throw an exception, depending on the I/O stream class.

In actuality, `mark()` and `reset()` are not putting the data back into the I/O stream but are storing the data in a temporary buffer in memory to be read again. Therefore, you should not call the `mark()` operation with too large a value, as this could take up a lot of memory.

Skipping Data

Assume that we have an `InputStream` instance whose next values are `TIGERS`. Consider the following code snippet:

```
System.out.print ((char)is.read()); // T
is.skip(2);  // Skips I and G
is.read();   // Reads E but doesn't output it
System.out.print((char)is.read());   // R
System.out.print((char)is.read());   // S
```

This code prints `TRS` at runtime. We skipped two characters, `I` and `G`. We also read `E` but didn't use it anywhere, so it behaved like calling `skip(1)`.

The return parameter of `skip()` tells us how many values were skipped. For example, if we are near the end of the I/O stream and call `skip(1000)`, the return value might be `20`, indicating that the end of the I/O stream was reached after `20` values were skipped. Using the return value of `skip()` is important if you need to keep track of where you are in an I/O stream and how many bytes have been processed.

Reviewing Manipulation APIs

Table 14.10 reviews these APIs related to manipulating I/O input streams. While you may not have used these in practice, you need to know them for the exam.

TABLE 14.1O 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

We begin our discussion by presenting the basic methods for reading file attributes. These methods are usable within any file system, although they may have limited meaning in some file systems.

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. Let's take a look at some sample code:

```
System.out.print(Files.isDirectory(Path.of("/canine/fur.jpg")))
;
System.out.print(Files.isSymbolicLink(Path.of("/canine/coyote")
));
System.out.print(Files.isRegularFile(Path.of("/canine/types.txt
")));
```

The first example prints `true` if `fur.jpg` is a directory or a symbolic link to a directory and `false` otherwise. The second example prints `true` if `/canine/coyote` is a symbolic link, regardless of whether the file or directory it points to exists. The third example prints `true` if `types.txt` points to a regular file or a symbolic link that points to a regular file.

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.

Here we present an example of each method:

```
System.out.print(Files.isHidden(Path.of("/walrus.txt")));
System.out.print(Files.isReadable(Path.of("/seal/baby.png")));
System.out.print(Files.isWritable(Path.of("dolphin.txt")));
System.out.print(Files.isExecutable(Path.of("whale.png")));
```

If the `walrus.txt` file exists and is hidden within the file system, the first example prints `true`. The second example prints `true` if the `baby.png` file exists and its contents are readable. The third example prints `true` if the `dolphin.txt` file can be modified. Finally, the last example prints `true` if the file can be executed within the operating system. Note that the file extension does not necessarily determine whether a file is executable. For example, an image file that ends in `.png` could be marked executable in some file systems.

With the exception of the `isHidden()` method, these methods do not declare any checked exceptions and return `false` if the file does not exist.

Improving Attribute Access

Up until now, we have been accessing individual file attributes with multiple method calls. While this is functionally correct, there is often a cost each time one of these methods is called. Put simply, it is far more efficient to ask the file system for all of the attributes at once rather than performing multiple round trips to the file system. Furthermore, some attributes are file system–specific and cannot be easily generalized for all file systems.

NIO.2 addresses both of these concerns by allowing you to construct views for various file systems with a single method call. A *view* is a group of related attributes for a particular file system type. That's not to say that the earlier attribute methods that we just finished discussing do not have their uses. If you need to read only one attribute of a file or directory, requesting a view is unnecessary.

Understanding Attribute and View Types

NIO.2 includes two methods for working with attributes in a single method call: a read-only attributes method and an updatable view method. For each method, you need to provide a file system type object, which tells the NIO.2 method which type of view you are requesting. By updatable view, we mean that we can both read and write attributes with the same object.

Table 14.11 lists the commonly used attributes and view types. For the exam, you only need to know about the basic file attribute types. The other views are for managing operating system–specific information.

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.

Retrieving Attributes

The `Files` class includes the following method to read attributes of a class in a read-only capacity:

```
public static <A extends BasicFileAttributes> A readAttributes(
   Path path,
   Class<A> type,
   LinkOption… options) throws IOException
```

Applying it requires specifying the `Path` and `BasicFileAttributes.class` parameters.

```
var path = Path.of("/turtles/sea.txt");
BasicFileAttributes data = Files.readAttributes(path,
   BasicFileAttributes.class);

System.out.println("Is a directory? " + data.isDirectory());
System.out.println("Is a regular file? " +
data.isRegularFile());
System.out.println("Is a symbolic link? " +
data.isSymbolicLink());
System.out.println("Size (in bytes): " + data.size());
System.out.println("Last modified: " +
data.lastModifiedTime());
```

The `BasicFileAttributes` class includes many values with the same name as the attribute methods in the `Files` class. The advantage of using this method, though, is that all of the attributes are retrieved at once for some operating systems.

Modifying Attributes

The following `Files` method returns an updatable view:

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

We can use the updatable view to increment a file's last modified date/time value by 10,000 milliseconds, or 10 seconds.

```
// Read file attributes
var path = Path.of("/turtles/sea.txt");
BasicFileAttributeView view = Files.getFileAttributeView(path,
```

```
   BasicFileAttributeView.class);
BasicFileAttributes attributes = view.readAttributes();

// Modify file last modified time
FileTime lastModifiedTime = FileTime.fromMillis(
   attributes.lastModifiedTime().toMillis() + 10_000);
view.setTimes(lastModifiedTime, null, null);
```

After the updatable view is retrieved, we need to call `readAttributes()` on the view to obtain the file metadata. From there, we create a new `FileTime` value and set it using the `setTimes()` method:

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

This method allows us to pass `null` for any date/time value that we do not want to modify. In our sample code, only the last modified date/time is changed.

Not all file attributes can be modified with a view. For example, you cannot set a property that changes a file into a directory. Likewise, you cannot change the size of the object without modifying its contents.

Traversing a Directory Tree

While the `Files.list()` method is useful, it traverses the contents of only a single directory. What if we want to visit all of the paths within a directory tree? Before we proceed, we need to review some basic concepts about file systems. Remember that a directory is organized in a hierarchical manner. For example, a directory can contain files and other directories, which can in turn contain other files and directories. Every record in a file system has exactly one parent, with the exception of the root directory, which sits atop everything.

A file system is commonly visualized as a tree with a single root node and many branches and leaves. In this model, a directory is a branch or internal node, and a file is a leaf node.

A common task in a file system is to iterate over the descendants of a path, either recording information about them or, more commonly, filtering them for a specific set of files. For example, you may want to search a folder and print a list of all of the `.java` files. Furthermore, file systems store file records in a hierarchical manner. Generally speaking, if you want to search for a file, you have to start with a parent directory, read its child elements, then read their children, and so on.

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. For example, if we're searching for a single file, we can end the search when the file is found or we've checked all files and come up empty. The starting path is usually a specific directory; after all, it would be time-consuming to search the entire file system on every request!

Selecting a Search Strategy

Two common strategies are associated with walking a directory tree: a depth-first search and a breadth-first search. A *depth-first search* traverses the structure from the root to an arbitrary leaf and then navigates back up toward the root, traversing fully any paths it skipped along the way. The *search depth* is the distance from the root to current node. To prevent endless searching, Java includes a search depth that is used to limit how many levels (or hops) from the root the search is allowed to go.

Alternatively, a *breadth-first search* starts at the root and processes all elements of each particular depth before proceeding to the next depth level. The results are ordered by depth, with all nodes at depth `1` read before all nodes at depth `2`, and so on. While a breadth-first search tends to be balanced and predictable, it also requires more memory since a list of visited nodes must be maintained.

For the exam, you don't have to understand the details of each search strategy that Java employs; you just need to be aware that the NIO.2 Stream

API methods use depth-first searching with a depth limit, which can be optionally changed.

Walking a Directory

That's enough background information; let's get to more Stream API methods. The `Files` class includes two methods for walking the directory tree using a depth-first search.

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

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

Like our other stream methods, `walk()` uses lazy evaluation and evaluates a `Path` only as it gets to it. This means that even if the directory tree includes hundreds or thousands of files, the memory required to process a directory tree is low. The first `walk()` method relies on a default maximum depth of `Integer.MAX_VALUE`, while the overloaded version allows the user to set a maximum depth. This is useful in cases where the file system might be large and we know the information we are looking for is near the root.

Rather than just printing the contents of a directory tree, we can again do something more interesting. The following `getPathSize()` method walks a directory tree and returns the total size of all the files in the directory:

```
private long getSize(Path p) {
   try {
      return Files.size(p);
   } catch (IOException e) {
      throw new UncheckedIOException(e);
   }
}

public long getPathSize(Path source) throws IOException {
   try (var s = Files.walk(source)) {
      return s.parallel()
            .filter(p -> !Files.isDirectory(p))
            .mapToLong(this::getSize)
            .sum();
   }
}
```

The `getSize()` helper method is needed because `Files.size()` declares `IOException`, and we'd rather not put a `try`/`catch` block inside a lambda expression. Instead, we wrap it in the unchecked exception class `UncheckedIOException`. We can print the data using the `format()` method:

```
var size = getPathSize(Path.of("/fox/data"));
System.out.format("Total Size: %.2f megabytes",
(size/1000000.0));
```

Depending on the directory you run this on, it will print something like this:

```
Total Size: 15.30 megabytes
```

Applying a Depth Limit

Let's say our directory tree is quite deep, so we apply a depth limit by changing one line of code in our `getPathSize()` method.

```
    try (var s = Files.walk(source, 5)) {
```

This new version checks for files only within five steps of the starting node. A depth value of `0` indicates the current path itself. Since the method calculates values only on files, you'd have to set a depth limit of at least `1` to get a nonzero result when this method is applied to a directory tree.

Avoiding Circular Paths

Many of our earlier NIO.2 methods traverse symbolic links by default, with a `NOFOLLOW_LINKS` used to disable this behavior. The `walk()` method is different in that it does *not* follow symbolic links by default and requires the `FOLLOW_LINKS` option to be enabled. We can alter our `getPathSize()` method to enable following symbolic links by adding the `FileVisitOption`:

```
    try (var s = Files.walk(source,
          FileVisitOption.FOLLOW_LINKS)) {
```

When traversing a directory tree, your program needs to be careful of symbolic links, if enabled. For example, if our process comes across a symbolic link that points to the root directory of the file system, every file in the system will be searched!

Worse yet, a symbolic link could lead to a cycle in which a path is visited repeatedly. A *cycle* is an infinite circular dependency in which an entry in a directory tree points to one of its ancestor directories. Let's say we had a directory tree as shown in Figure 14.6 with the symbolic link `/birds/robin/allBirds` that points to `/birds`.

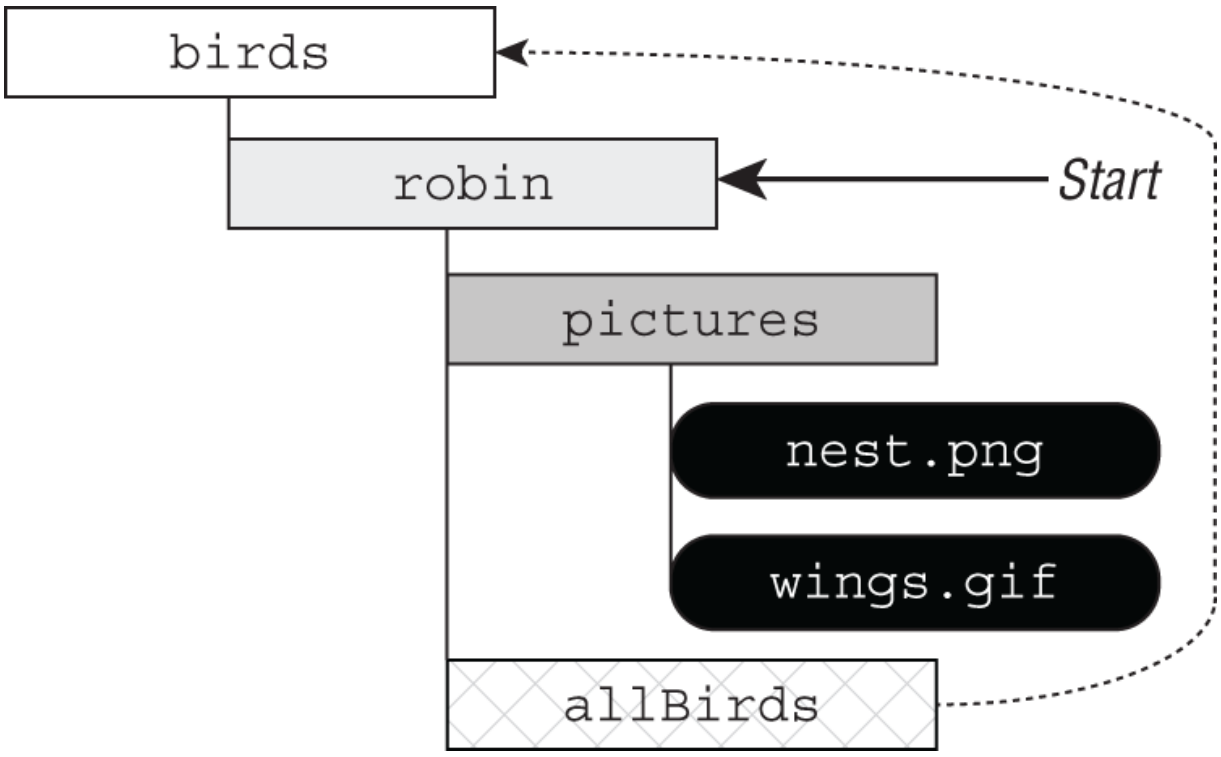

FIGURE 14.6 File system with cycle

What happens if we try to traverse this tree and follow all symbolic links, starting with `/birds/robin`? Table 14.12 shows the paths visited after walking a depth of `3`. For simplicity, we walk the tree in a breadth-first ordering, *although a cycle occurs regardless of the search strategy used*.

TABLE 14.12 Walking a directory with a cycle using breadth-first search

Depth	Path reached
0	`/birds/robin`
1	`/birds/robin/pictures`
1	`/birds/robin/allBirds` ➢ `/birds`
2	`/birds/robin/pictures/nest.png`
2	`/birds/robin/pictures/wings.gif`
2	**`/birds/robin/allBirds/robin`** ➢ /birds/robin
3	`/birds/robin/allBirds/robin/pictures` ➢ `/birds/robin/pictures`
3	`/birds/robin/allBirds/robin/allBirds` ➢ `/birds`

After walking a distance of `1` from the start, we hit the symbolic link `/birds/robin/allBirds` and go back to the top of the directory tree `/birds`. That's OK because we haven't visited `/birds` yet, so there's no cycle yet!

Unfortunately, at depth `2`, we encounter a cycle. We've already visited the `/birds/robin` directory on our first step, and now we're encountering it again. If the process continues, we'll be doomed to visit the directory over and over again.

Be aware that when the `FOLLOW_LINKS` option is used, the `walk()` method will track all of the paths it has visited, throwing a `FileSystemLoopException` if a path is visited twice.

Searching a Directory

In the previous example, we applied a filter to the `Stream<Path>` object to filter the results, although there is a more convenient method.

```
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.

The two parameters of the `BiPredicate` are a `Path` object and a `BasicFileAttributes` object, which you saw earlier in the chapter. In this manner, Java automatically retrieves the basic file information for you, allowing you to write complex lambda expressions that have direct access to this object. We illustrate this with the following example:

```
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`. While we could have accomplished this using the `walk()` method along with a call to `readAttributes()`, this implementation is a lot shorter and more convenient than those would have been. We also don’t have to worry about any methods within the lambda expression declaring a checked exception, as we saw in the `getPathSize()` example.

Using *DirectoryStream*

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);
}
```

It's better to use `Files.find()` in new code, but you might see either approach on the exam.

Review of Key APIs

Table 14.13 lists the key APIs you need to know for the exam. We know some of the classes look similar. You need to know this table really well before taking the exam.

TABLE 14.1C 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

Additionally, Figure 14.7 shows all of the I/O stream classes that you should be familiar with for the exam, with the exception of the filter streams. `FilterInputStream` and `FilterOutputStream` are high-level superclasses that filter or transform data. They are rarely used directly.

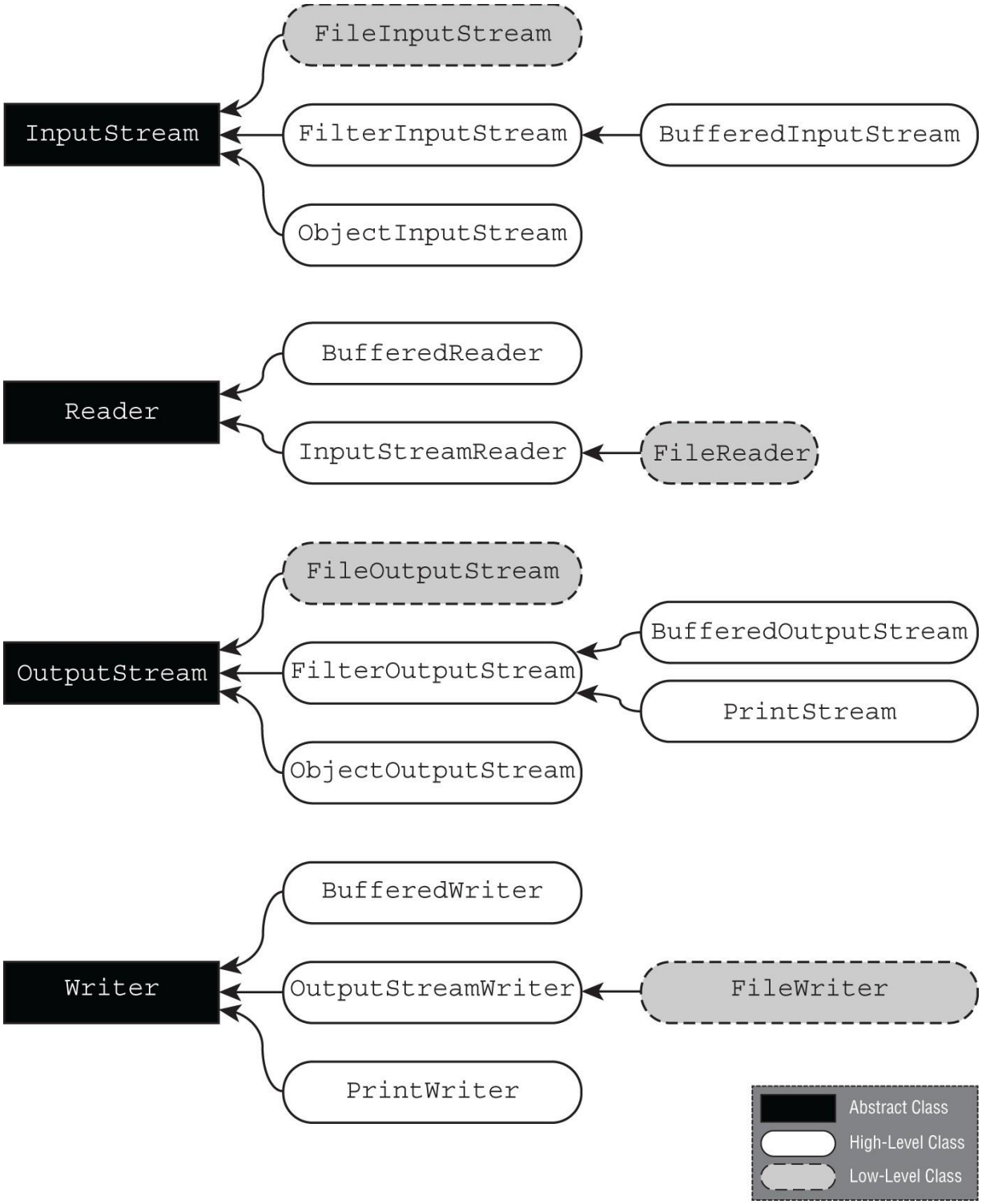

FIGURE 14.7 Diagram of I/O stream classes

The `InputStreamReader` and `OutputStreamWriter` are incredibly convenient and are also unique in that they are the only I/O stream classes

to have both `InputStream`/`OutputStream` and `Reader`/`Writer` in their name.

Summary

This chapter is all about reading and writing data. We started by showing you how to create `File` from I/O and `Path` from NIO.2. We then covered the functionality that works with both I/O and NIO.2 before getting into NIO.2-specific APIs. You should be familiar with how to combine or resolve `Path` objects with other `Path` objects. Additionally, NIO.2 includes Stream API methods that can be used to process files and directories. We discussed methods for listing a directory, walking a directory tree, searching a directory tree, and reading the lines of a file.

We spent time reviewing various methods available in the `Files` helper class. As discussed, the name of the function often tells you exactly what it does. We explained that most of these methods are capable of throwing an `IOException`, and many take optional varargs enum values.

We then introduced I/O streams and explained how they are used to read or write large quantities of data. While there are a lot of I/O streams, they differ on some key points:

- Byte versus character streams
- Input versus output streams
- Low-level versus high-level streams

Often, the name of the I/O stream can tell you a lot about what it does. We visited many of the I/O stream classes that you will need to know for the exam in increasing order of complexity. A common practice is to start with a low-level resource or file stream and wrap it in a buffered I/O stream to improve performance. You can also apply a high-level stream to manipulate the data, such as an object or print stream. We described what it means to be serializable in Java, and we showed you how to use the object stream classes to persist objects directly to and from disk.

We explained how to read input data from the user using both the system stream objects and the `Console` class. The `Console` class has many useful features, such as built-in support for passwords and formatting.

We also discussed how NIO.2 provides methods for reading and writing file metadata. NIO.2 includes two methods for retrieving all of the file system attributes for a path in a single call without numerous round trips to the operating system. One method returns a read-only attribute type, while the second method returns an updatable view type. It also allows NIO.2 to support operating system–specific file attributes.

Exam Essentials

Understand files and directories. Files are records that store data within a persistent storage device, such as a hard disk drive, that is available after the application has finished executing. Files are organized within a file system in directories, which in turn may contain other directories. The root directory is the topmost directory in a file system.

Be able to use `File` and `Path`. An I/O `File` instance is created by calling the constructor. It contains a number of instance methods for creating and manipulating a file or directory. An NIO.2 `Path` instance is an immutable object that is created from the factory method `Path.of()`. The `Path` interface includes many instance methods for reading and manipulating the path value.

Distinguish between types of I/O streams. I/O streams are categorized by byte/character, input/output, and low-level/high-level. Byte streams operate on binary data and have names that end with `Stream`, while character streams operate on text data and have names that end in `Reader` or `Writer`. The `InputStream` and `Reader` classes are the topmost abstract classes that receive data, while the `OutputStream` and `Writer` classes are the topmost abstract classes that send data. A low-level stream is one that operates directly on the underlying resource, such as a file or network connection. A high-level stream operates on a low-level or other high-level stream to filter data, convert data, or improve performance.

Understand how to use Java serialization. A class is considered serializable if it implements the `java.io.Serializable` interface and contains instance members that are either serializable or marked `transient`. All Java primitives and the `String` class are serializable. The

`ObjectInputStream` and `ObjectOutputStream` classes can be used to read and write a `Serializable` object from and to an I/O stream, respectively.

Be able to interact with the user. Be able to interact with the user using the system streams (`System.out`, `System.err`, and `System.in`) as well as the `Console` class. The `Console` class includes special methods for formatting data and retrieving complex input such as passwords.

Manage file attributes. The NIO.2 `Files` class includes many methods for reading single file attributes, such as its size or whether it is a directory, a symbolic link, hidden, etc. NIO.2 also supports reading all of the attributes in a single call. An attribute type is used to support operating system–specific views. Finally, NIO.2 supports updatable views for modifying selected attributes.