Chapter 20
NIO.2

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- I/O (Fundamentals and NIO2)
  - Use the Path interface to operate on file and directory paths
  - Use the Files class to check, delete, copy or move a file or directory
  - Use the Stream API with Files

In Chapter 19, “I/O,” we presented the `java.io` API and discussed how to use it to interact with files and streams. In this chapter, we focus on the `java.nio` version 2 API, or NIO.2 for short, to interact with files. NIO.2 is an acronym that stands for the second version of the Non-blocking Input/Output API, and it is sometimes referred to as the “New I/O.”

In this chapter, we will show how NIO.2 allows us to do a lot more with files and directories than the original `java.io` API. We'll also show you how to apply the Streams API to perform complex file and directory operations. We'll conclude this chapter by showing the various ways file attributes can be read and written using NIO.2.

NOTE

While Chapter 19 focused on I/O streams, we're back to using streams to refer to the Streams API that you learned about in Chapter 15, “Functional Programming.” For clarity, we'll use the phrase *I/O streams* to discuss the ones found in `java.io` from this point on.

Introducing NIO.2

At its core, NIO.2 is a replacement for the legacy `java.io.File` class you learned about in Chapter 19. The goal of the API is to provide a more intuitive, more feature-rich API for working with files and directories.

By *legacy*, we mean that the preferred approach for working with files and directories with newer software applications is to use NIO.2, rather than `java.io.File`. As you'll soon see, the NIO.2 provides many features and performance improvements than the legacy class supported.

WHAT ABOUT NIO?

This chapter focuses on NIO.2, not NIO. Java includes an NIO library that uses buffers and channels, in place of I/O streams. The NIO API was never popular, so much so that nothing from the original version of NIO will be on the OCP exam. Many Java developers continue to use I/O streams to manipulate byte streams, rather than NIO.

People sometimes refer to NIO.2 as just NIO, although for clarity and to distinguish it from the first version of NIO, we will refer to it as NIO.2 throughout the chapter.

Introducing *Path*

The cornerstone of NIO.2 is the `java.nio.file.Path` interface. A `Path` instance represents a hierarchical path on the storage system to a file or directory. You can think of a `Path` as the NIO.2 replacement for the `java.io.File` class, although how you use it is a bit different.

Before we get into that, let's talk about what's similar between these two implementations. Both `java.io.File` and `Path` objects may refer to an absolute path or relative path within the file system. In addition, both may refer to a file or a directory. As we did in Chapter 19 and continue to do in this chapter, we treat an instance that points to a directory as a file since it is stored in the file system with similar properties. For example, we can rename a file or directory with the same commands in both APIs.

Now for something completely different. Unlike the `java.io.File` class, the `Path` interface contains support for symbolic links. A *symbolic link* is a special file within a file system that serves as a reference or pointer to another file or directory. Figure 20.1 shows a symbolic link from `/zoo/favorite` to `/zoo/cats/lion`.

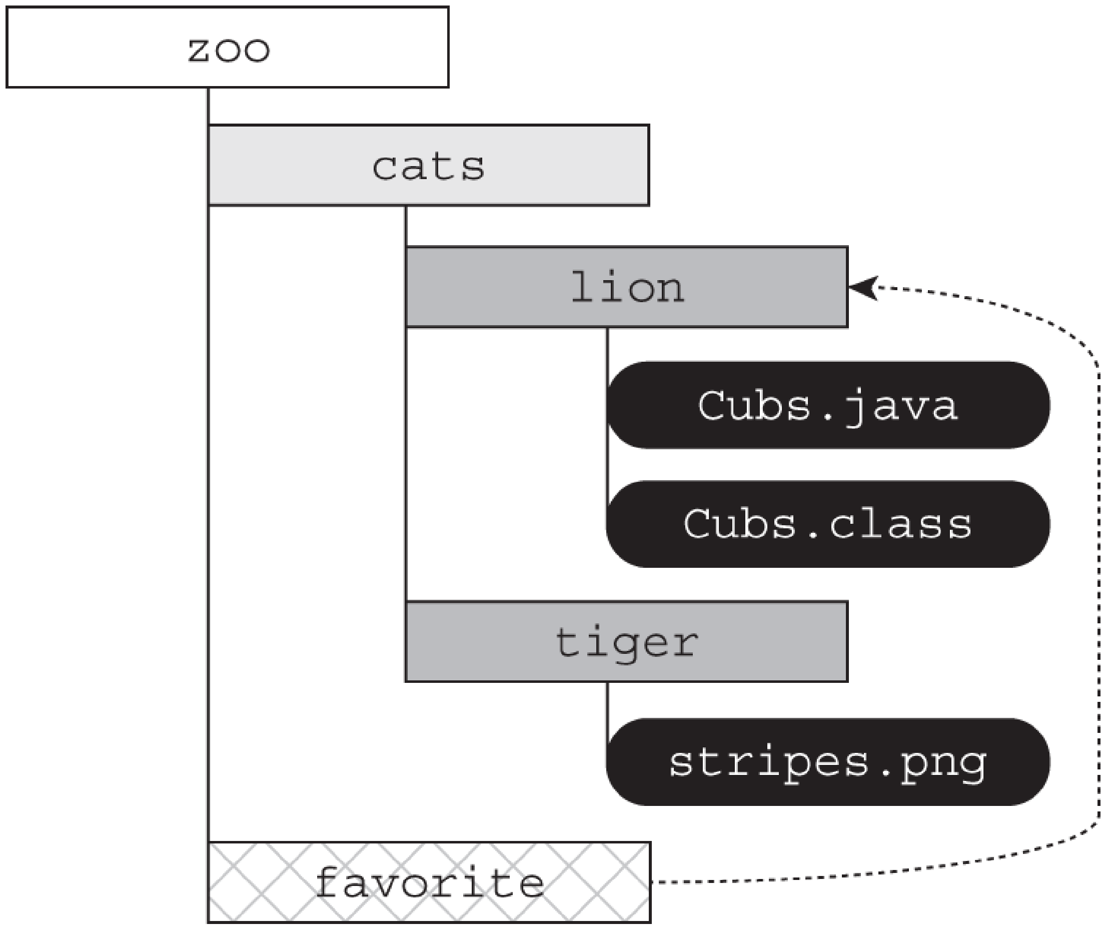

FIGURE 20.1 File system with a symbolic link

In Figure 20.1, the `lion` folder and its elements can be accessed directly or via the symbolic link. For example, the following paths reference the same file:

```
/zoo/cats/lion/Cubs.java
/zoo/favorite/Cubs.java
```

In general, symbolic links are transparent to the user, as the operating system takes care of resolving the reference to the actual file. NIO.2 includes full support for creating, detecting, and navigating symbolic links within the file system.

Creating Paths

Since `Path` is an interface, we can't create an instance directly. After all, interfaces don't have constructors! Java provides a number of classes and methods that you can use to obtain `Path` objects, which we will review in this section.

NOTE

You might wonder, why is `Path` an interface? When a `Path` is created, the JVM returns a file system–specific implementation, such as a Windows or Unix `Path` class. In the vast majority of circumstances, we want to perform the same operations on the `Path`, regardless of the file system. By providing `Path` as an interface using the factory pattern, we avoid having to write complex or custom code for each type of file system.

Obtaining a *Path* with the *Path* Interface

The simplest and most straightforward way to obtain a `Path` object is to use the `static` factory method defined within the `Path` interface.

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

It's easy to create `Path` instances from `String` values, as shown here:

```
Path path1 = Path.of("pandas/cuddly.png");
Path path2 = Path.of("c:\\zooinfo\\November\\employees.txt");
Path path3 = Path.of("/home/zoodirectory");
```

The first example creates a reference to a relative path in the current working directory. The second example creates a reference to an absolute file path in a Windows-based system. The third example creates a reference to an absolute directory path in a Linux or Mac-based system.

ABSOLUTE VS. RELATIVE PATHS

Determining whether a path is relative or absolute is actually 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. Examples: `/bird/parrot.png` and `/bird/../data/./info`
- If a path starts with a drive letter (`c:`), it is absolute, with the drive letter as the root directory. Examples: `c:/bird/parrot.png` and `d:/bird/../data/./info`
- Otherwise, it is a relative path. Examples: `bird/parrot.png` and `bird/../data/./info`

If you're not familiar with path symbols like `.` and `..` , don't worry! We'll be covering them in this chapter.

The `Path.of()` method also includes a varargs to pass additional path elements. The values will be combined and automatically separated by the operating system–dependent file separator you learned about in Chapter 19.

```
Path path1 = Path.of("pandas", "cuddly.png");
Path path2 = Path.of("c:", "zooinfo", "November", "employees.txt");
Path path3 = Path.of("/", "home", "zoodirectory");
```

These examples are just rewrites of our previous set of `Path` examples, using the parameter list of `String` values instead of a single `String` value. The advantage of the varargs is that it is more robust, as it inserts the proper operating system path separator for you.

Obtaining a *Path* with the *Paths* Class

The `Path.of()` method is actually new to Java 11. Another way to obtain a `Path` instance is from the `java.nio.file.Paths` factory class. Note the `s` at the end of the `Paths` class to distinguish it from the `Path` interface.

```
// Paths factory method
public static Path get(String first, String… more)
```

Rewriting our previous examples is easy.

```
Path path1 = Paths.get("pandas/cuddly.png");
Path path2 = Paths.get("c:\\zooinfo\\November\\employees.txt");
Path path3 = Paths.get("/", "home", "zoodirectory");
```

Since `Paths.get()` is older, the exam is likely to have both. We'll use both `Path.of()` and `Paths.get()` interchangeably in this chapter.

Obtaining a *Path* with a *URI* Class

Another way to construct a `Path` using the `Paths` class is with a URI value. A *uniform resource identifier* (URI) is a string of characters that identify a resource. It begins with a schema that indicates the resource type, followed by a path value. Examples of schema values include `file://` for local file systems, and `http://`, `https://`, and `ftp://` for remote file systems.

The java.net `.URI` class is used to create URI values.

```
// URI Constructor
public URI(String str) throws URISyntaxException
```

Java includes multiple methods to convert between `Path` and `URI` objects.

```
// URI to Path, using Path factory method
public static Path of(URI uri)
```

```
// URI to Path, using Paths factory method
public static Path get(URI uri)

// Path to URI, using Path instance method
public URI toURI()
```

The following examples all reference the same file:

```
URI a = new URI("file://icecream.txt");
Path b = Path.of(a);
Path c = Paths.get(a);
URI d = b.toUri();
```

Some of these examples may actually throw an `IllegalArgumentException` at runtime, as some systems require URIs to be absolute. The `URI` class does have an `isAbsolute()` method, although this refers to whether the URI has a schema, not the file location.

OTHER URI CONNECTION TYPES

A URI can be used for a web page or FTP connection.

```
    Path path5 = Paths.get(new URI("http://www.wiley.com"));
    Path path6 = Paths.get(new URI("ftp://username:secret@ftp.example.com"))
```

For the exam, you do not need to know the syntax of these types of URIs, but you should be aware they exist.

Obtaining a *Path* from the *FileSystem* Class

NIO.2 makes extensive use of creating objects with factory classes. As you saw already, the `Paths` class creates instances of the `Path` interface. Likewise, the `FileSystems` class creates instances of the abstract `FileSystem` class.

```
// FileSystems factory method
public static FileSystem getDefault()
```

The `FileSystem` class includes methods for working with the file system directly. In fact, both `Paths.get()` and `Path.of()` are actually shortcuts for this `FileSystem` method:

```
// FileSystem instance method
public Path getPath(String first, String… more)
```

Let's rewrite our three earlier examples one more time to show you how to obtain a `Path` instance “the long way.”

```
Path path1 = FileSystems.getDefault().getPath("pandas/cuddly.png");
Path path2 = FileSystems.getDefault()
   .getPath("c:\\zooinfo\\November\\employees.txt");
Path path3 = FileSystems.getDefault().getPath("/home/zoodirectory");
```

Real World Scenario

CONNECTING TO REMOTE FILE SYSTEMS

While most of the time we want access to a `Path` object that is within the local file system, the `FileSystems` class does give us the freedom to connect to a remote file system, as follows:

```
// FileSystems factory method
public static FileSystem getFileSystem(URI uri)
```

The following shows how such a method can be used:

```
FileSystem fileSystem = FileSystems.getFileSystem(
   new URI("http://www.selikoff.net"));
Path path = fileSystem.getPath("duck.txt");
```

This code is useful when we need to construct `Path` objects frequently for a remote file system. NIO.2 gives us the power to connect to both local and remote file systems, which is a major improvement over the legacy `java.io.File` class.

Obtaining a *Path* from the *java.io.File* Class

Last but not least, we can obtain `Path` instances using the legacy `java.io.File` class. In fact, we can also obtain a `java.io.File` object from a `Path` instance.

```
// Path to File, using Path instance method
public default File toFile()

// File to Path, using java.io.File instance method
public Path toPath()
```

These methods are available for convenience and also to help facilitate integration between older and newer APIs. The following shows examples of each:

```
File file = new File("husky.png");
Path path = file.toPath();
File backToFile = path.toFile();
```

When working with newer applications, though, you should rely on NIO.2's `Path` interface as it contains a lot more features.

Reviewing NIO.2 Relationships

By now, you should realize that NIO.2 makes extensive use of the factory pattern. You should become comfortable with this paradigm. Many of your interactions with NIO.2 will require two types: an abstract class or interface and a factory or helper class. Figure 20.2 shows the relationships among NIO.2 classes, as well as select `java.io` and java.net classes.

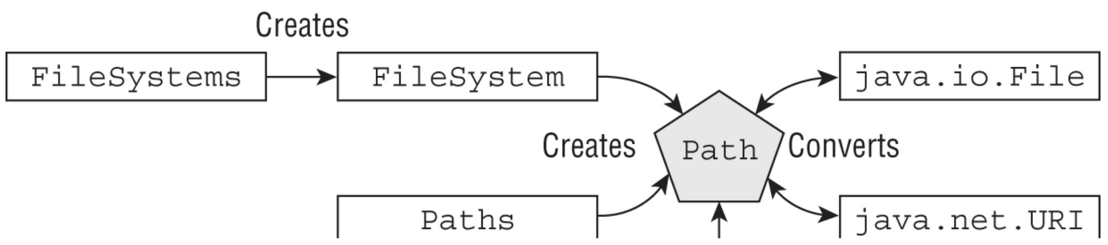

FIGURE 20.2 NIO.2 class and interface relationships

Review Figure 20.2 carefully. When working with NIO.2, keep an eye on whether the class name is singular or plural. The classes with plural names include methods to create or operate on class/interface instances with singular names. Remember, a `Path` can also be created from the `Path` interface, using the `static` factory `of()` method.

Included in Figure 20.2 is the class `java.nio.file.Files`, which we'll cover later in the chapter. For now, you just need to know that it is a helper or utility class that operates primarily on `Path` instances to read or modify actual files and directories.

NOTE

The `java.io.File` is the I/O class you worked with in Chapter 19, 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! For clarity, we often write out the full name of the `java.io.File` class in this chapter.

Understanding Common NIO.2 Features

Throughout this chapter, we introduce numerous methods you should know for the exam. Before getting into the specifics of each method, we present many of these common features in this section so you are not surprised when you see them.

Applying Path Symbols

Absolute and relative paths can contain path symbols. A *path symbol* is a reserved series of characters that have special meaning within some file systems. For the exam, there are two path symbols you need to know, as listed in Table 20.1.

TABLE 20.1 File system symbols

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

We illuminate using path symbols in Figure 20.3.

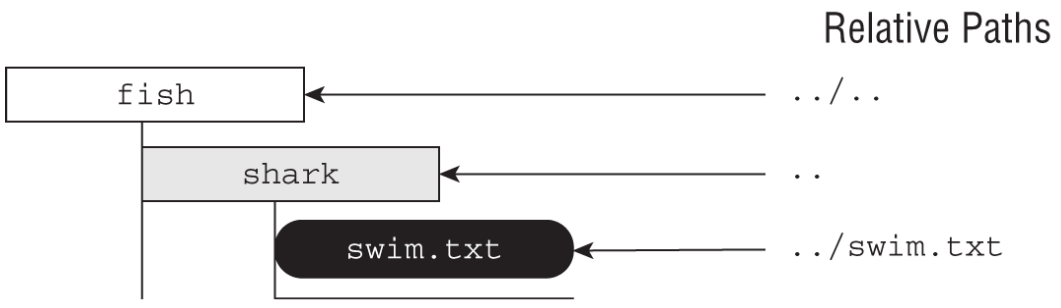

FIGURE 20.3 Relative paths using path symbols

In Figure 20.3, the current directory is `/fish/shark/hammerhead` . In this case, `../swim.txt` refers to the file `swim.txt` in the parent of the current directory. Likewise, `./play.png` refers to `play.png` in the current directory. These symbols can also be combined for greater effect. For example, `../../clownfish` refers to the directory that is two directories up from the current directory.

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

Providing Optional Arguments

Many of the methods in this chapter include a varargs that takes an optional list of values. Table 20.2 presents the arguments you should be familiar with for the exam.

TABLE 20.2 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, then append to the end.

Enum type	Interface inherited	Enum value	Details
		`CREATE`	Create a new file if it does not exist.
		`CREATE_NEW`	Create a new file only if it does not exist, fail otherwise.
		`READ`	Open for read access.
		`TRUNCATE_EXISTING`	If file is already open for write, then 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()` (which we'll cover later), we won't discuss these varargs parameters each time we present a method. The behavior of them 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 = Paths.get("schedule.xml");
boolean exists = Files.exists(path, LinkOption.NOFOLLOW_LINKS);
```

The `Files.exists()` simply checks whether a file exists. If the parameter is a symbolic link, though, then 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 20.2 inherit an interface. That means some methods accept a variety of enums types. For example, the `Files.move()` method takes a `CopyOption` vararg so it can take enums of different types.

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

NOTE

Many of the NIO.2 methods use a varargs for passing options, even when there is only one enum value available. The advantage of this approach, as opposed to say just passing a `boolean` argument, is future-proofing. These method signatures are insulated from changes in the Java language down the road when future options are available.

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 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.

In general, methods that operate on abstract `Path` values, such as those in the `Path` interface or `Paths` class, often do not throw any checked exceptions. On the other hand, methods that operate 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, then it would never be able to return `false`!

Interacting with Paths

Now that we've covered the basics of NIO.2, you might ask, what can we do with it? Short answer: a lot. NIO.2 provides a *rich plethora* of methods and classes that operate on `Path` objects—far more than were available in the `java.io` API. In this section, we present the `Path` methods you should know for the exam, organized by related functionality.

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'll discuss in this section of the chapter:

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

NOTE

If you start to feel overwhelmed by the number of methods available in the `Path` interface, just remember: the function of many of them can be inferred by their method name. For example, what do you think the `Path` method `getParent()` does? It returns the parent directory of a `Path`. Not so difficult, was it?

Many of the code snippets in this part of the chapter can be run without the paths they reference actually existing. The JVM communicates with the file system to determine the path components or the parent directory of a file, without requiring the file to actually exist. As rule of thumb, if the method declares an `IOException`, then it *usually* requires the paths it operates on to exist.

Viewing the Path with *toString()*, *getNameCount()*, and *getName()*

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

```
public String toString()

public int getNameCount()

public Path getName(int index)
```

The first method, `toString()`, 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 in conjunction 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 = Paths.get("/land/hippo/harry.happy");
System.out.println("The Path Name is: " + path);
for(int i=0; i<path.getNameCount(); i++) {
   System.out.println("   Element " + i + " is: " + path.getName(i));
}
```

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

The code prints the following:

```
The Path Name 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 as 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.

NOTE

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

Creating a New Path with *subpath()*

The `Path` interface includes a method to select portions of a path.

```
public Path subpath(int beginIndex, int endIndex)
```

The references are inclusive of the `beginIndex`, and exclusive of the `endIndex`. The `subpath()` method is similar to the previous `getName()` method, except that `subpath()` may return multiple path components, whereas `getName()` returns only one. Both return `Path` instances, though.

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 = Paths.get("/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 0-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 with *getFileName()*, *getParent()*, and *getRoot()*

The `Path` interface contains numerous methods for retrieving particular elements of a `Path`, returned as `Path` objects themselves.

```
public Path getFileName()

public Path getParent()

public Path getRoot()
```

The `getFileName()` returns the `Path` element of the current file or directory, while `getParent()` returns the full path of the containing directory. The `getParent()` 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);
   }
}
```

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()` 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 will treat path symbols as part of the path, we will present one shortly that cleans up path symbols.

Checking Path Type with *isAbsolute()* and *toAbsolutePath()*

The `Path` interface contains two methods for assisting with relative and absolute paths:

```
public boolean isAbsolute()

public Path toAbsolutePath()
```

The first method, `isAbsolute()`, returns `true` if the path the object references is absolute and `false` if the path the object references is relative. As discussed earlier in this chapter, whether a path is absolute or relative is often file system–dependent, although we, like the exam writers, adopt common conventions for simplicity throughout the book.

The second method, `toAbsolutePath()`, converts a relative `Path` object to an absolute `Path` object by joining it to the current working directory. If the `Path` object is already absolute, then the method just returns the `Path` object.

TIP

The current working directory can be selected from `System.getProperty("user.dir")`. This is the value that `toAbsolutePath()` will use when applied to a relative path.

The following code snippet shows usage of both of these methods when run on a Windows and Linux system, respectively:

```
var path1 = Paths.get("C:\\birds\\egret.txt");
System.out.println("Path1 is Absolute? " + path1.isAbsolute());
System.out.println("Absolute Path1: " + path1.toAbsolutePath());

var path2 = Paths.get("birds/condor.txt");
System.out.println("Path2 is Absolute? " + path2.isAbsolute());
System.out.println("Absolute Path2 " + path2.toAbsolutePath());
```

The output for the code snippet on each respective system is shown in the following sample output. For the second example, assume the current working directory is `/home/work`.

```
Path1 is Absolute? true
Absolute Path1: C:\birds\egret.txt

Path2 is Absolute? false
Absolute Path2 /home/work/birds/condor.txt
```

Joining Paths with *resolve()*

Suppose you want to concatenate paths in a similar manner as we concatenate strings. The `Path` interface provides two `resolve()` methods for doing just that.

```
public Path resolve(Path other)

public Path resolve(String other)
```

The first method takes a `Path` parameter, while the overloaded version is a shorthand form of the first that takes a `String` (and constructs the `Path` for you). 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 up to now, `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 `path3` 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, then that is the value that is returned. Simply put, you cannot combine two absolute paths using `resolve()` .

TIP

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

Deriving a Path with *relativize()*

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

```
public Path relativize()
```

What do you think the following examples using `relativize()` 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, then the `relativize()` method computes the paths as if they are in the same current working directory.

Alternatively, if both path values are absolute, then 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:

```
Path path3 = Paths.get("E:\\habitat");
Path path4 = Paths.get("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 code snippet works even if you do not have an `E:` in your system. Remember, most methods defined in the `Path` interface do not require the path to exist.

The `relativize()` method requires that both paths are absolute or both relative and throws an exception if the types are mixed.

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

On Windows-based systems, it also requires that if absolute paths are used, then 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:

```
Path path3 = Paths.get("c:\\primate\\chimpanzee");
Path path4 = Paths.get("d:\\storage\\bananas.txt");
path3.relativize(path4); // IllegalArgumentException
```

Cleaning Up a Path with *normalize()*

So far, we've presented a number of examples that included path symbols that were unnecessary. Luckily, Java provides a method to eliminate unnecessary redundancies in a path.

```
public Path normalize()
```

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 = Paths.get("/pony/../weather.txt");
var p2 = Paths.get("/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 File System Path with *toRealPath()*

While working with theoretical paths is useful, sometimes you want to verify the path actually exists within the file system.

```
public Path toRealPath(LinkOption… options) throws IOException
```

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 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(Paths.get("/zebra/food.txt").toRealPath());
System.out.println(Paths.get(".././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(Paths.get(".").toRealPath());
```

Reviewing *Path* Methods

We conclude this section with Table 20.3, which shows the `Path` methods that you should know for the exam.

TABLE 20.3 Path methods

`Path of(String, String…)`	`Path getParent()`
`URI toURI()`	`Path getRoot()`
`File toFile()`	`boolean isAbsolute()`
`String toString()`	`Path toAbsolutePath()`
`int getNameCount()`	`Path relativize()`
`Path getName(int)`	`Path resolve(Path)`
`Path subpath(int, int)`	`Path normalize()`
`Path getFileName()`	`Path toRealPath(LinkOption…)`

Other than the `static` method `Path.of()`, all of the methods in Table 20.3 are instance methods that can be called on any `Path` instance. In addition, only `toRealPath()` declares an `IOException`.

Operating on Files and Directories

Most of the methods we covered in the `Path` interface operate on theoretical paths, which are not required to exist within the file system. What if you want to rename a directory, copy a file, or read the contents of a file?

Enter the NIO.2 `Files` class. The `Files` helper class is capable of interacting with real files and directories within the system. Because of this, most of the methods in this part of the chapter take optional parameters and throw an `IOException` if the path does not exist. The `Files` class also replicates numerous methods found in the `java.io.File`, albeit often with a different name or list of parameters.

NOTE

Many of the names for the methods in the NIO.2 `Files` class are a lot more straightforward than what you saw in the `java.io.File` class. For example, the `java.io.File` methods `renameTo()` and `mkdir()` have been changed to `move()` and `createDirectory()`, respectively, in the `Files` class.

Checking for Existence with *exists()*

The first `Files` method we present is the simplest. It just checks whether the file exists.

```
public static boolean exists(Path path, LinkOption… options)
```

Let's take a look at some sample code that operates on a `features.png` file in the `/ostrich` directory.

```
var b1 = Files.exists(Paths.get("/ostrich/feathers.png"));
System.out.println("Path " + (b1 ? "Exists" : "Missing"));

var b2 = Files.exists(Paths.get("/ostrich"));
System.out.println("Path " + (b2 ? "Exists" : "Missing"));
```

The first example checks whether a file exists, while the second example checks whether a directory exists. This method does not throw an exception if the file does not exist, as doing so would prevent this method from ever returning `false` at runtime.

TIP

Remember, a file and directory may both have extensions. In the last example, the two paths could refer to two files or two directories. Unless the exam tells you whether the path refers to a file or directory, do not assume either.

Testing Uniqueness with *isSameFile()*

Since a path may include path symbols and symbolic links within a file system, it can be difficult to know if two `Path` instances refer to the same file. Luckily, there's a method for that in the `Files` class:

```
public static boolean isSameFile(Path path, Path path2)
   throws IOException
```

The 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 usages 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()`, then the method will just return `true` without checking whether the file exists.

Assume the file system exists as shown in Figure 20.4 with a symbolic link from `/animals/snake` to `/animals/cobra`.

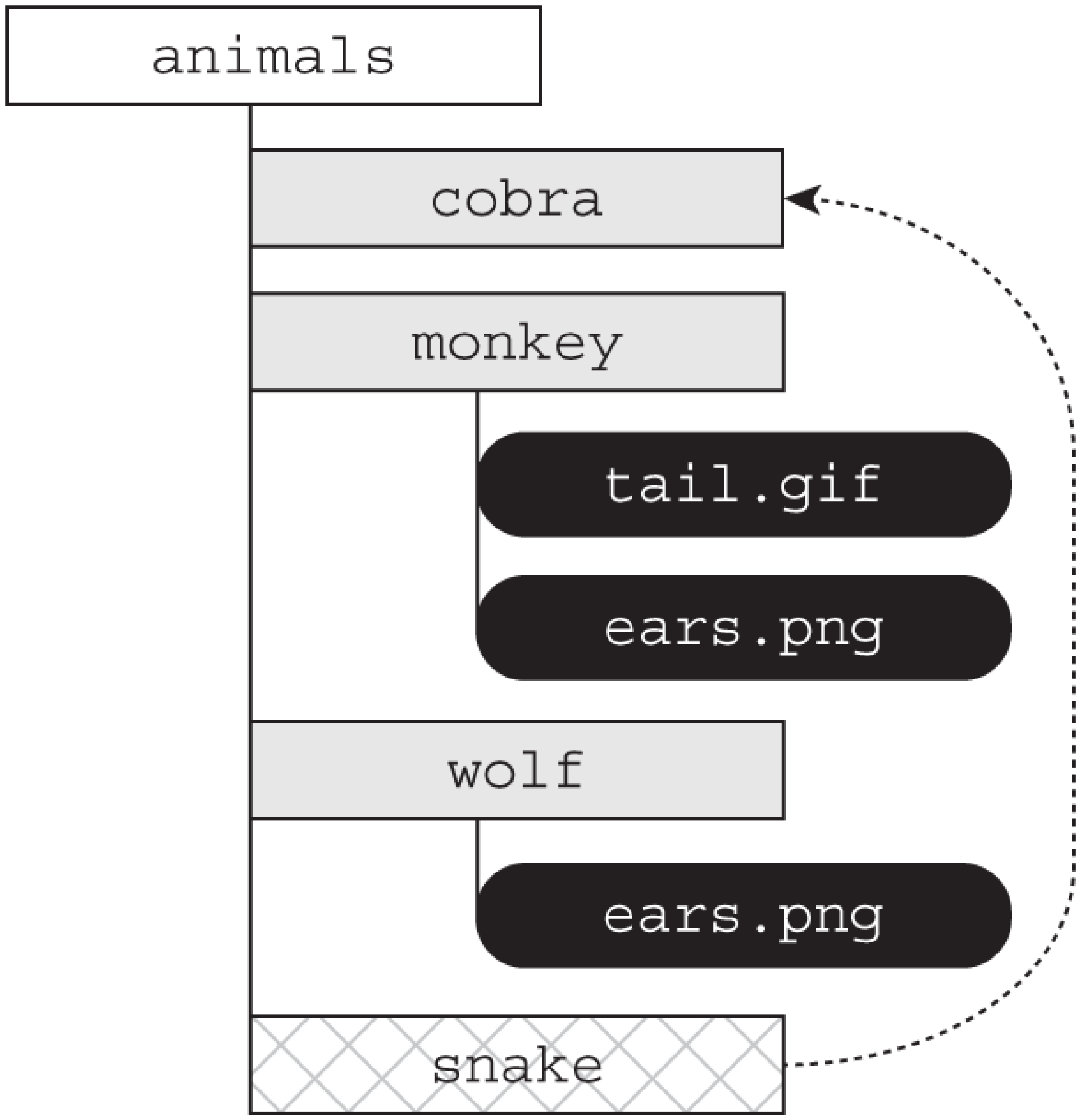

FIGURE 20.4 Comparing file uniqueness

Given the structure defined in Figure 20.4, 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 cobra is a symbolic link to snake, the first example outputs true. In the second example, the paths refer to different files, so false is printed.

NOTE

This `isSameFile()` method does not compare the contents of the files. Two files may have identical names, content, and attributes, but if they are in different locations, then this method will return `false`.

Making Directories with *createDirectory()* and *createDirectories()*

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()` will create a directory and throw an exception if it already exists or the paths leading up to the directory do not exist.

The `createDirectories()` works just like the `java.io.File` method `mkdirs()`, in that it 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 will discuss file attributes more later in the chapter.

The following shows how to create directories in NIO.2:

```
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; or else 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 with *copy()*

The NIO.2 `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(Paths.get("/panda/bamboo.txt"),
   Paths.get("/panda-save/bamboo.txt"));

Files.copy(Paths.get("/turtle"), Paths.get("/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. We'll show how to perform a deep copy of a directory tree using streams later in the chapter.

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(Paths.get("book.txt"), Paths.get("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 I/O 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 a stream and writes the output to a file. The second method reads the contents of a file and writes the output to a stream. They 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 stream data to a file
   Files.copy(is, Paths.get("/mammals/wolf.txt"));
}

Files.copy(Paths.get("/fish/clown.xsl"), System.out);
```

While we used `FileInputStream` in the first example, the streams 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 = Paths.get("food.txt");
var directory = Paths.get("/enclosure");
Files.copy(file, directory);
```

If you said it would create a new file at `/enclosure/food.txt`, then you're way off. It actually 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, then it would create a new file with the contents of `food.txt`, but it 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 the `move()` methods, the latter of which we will be covering next. In case you're curious, the correct way to copy the file into the directory would be to do the following:

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

You also define `directory` using the `resolve()` method we saw earlier, which saves you from having to write the filename twice.

```
var directory = Paths.get("/enclosure").resolve(file.getFileName());
```

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 is some sample code that 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 it renames it to `addresses2.txt`.

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

Like `copy()`, `move()` requires `REPLACE_EXISTING` to overwrite the target if it exists, else 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 18, “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, then 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, and `false` otherwise. Similar to `createDirectories()`, `deleteIfExists()` is useful in situations where you want to ensure a path does not exist, and delete it if it does.

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

```
Files.delete(Paths.get("/vulture/feathers.txt"));
Files.deleteIfExists(Paths.get("/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, then the second line will not throw an exception.

Reading and Writing Data with *newBufferedReader()* and *newBufferedWriter()*

NIO.2 includes two convenient methods for working with I/O streams.

```
public static BufferedReader newBufferedReader(Path path)
   throws IOException

public static BufferedWriter newBufferedWriter(Path path,
   OpenOption… options) throws IOException
```

You can wrap I/O stream constructors to produce the same effect, although it's a lot easier to use the factory method.

NOTE

There are overloaded versions of these methods that take a `Charset`. You may remember that we briefly discussed character encoding and `Charset` in Chapter 19. For this chapter, you just need to know that characters can be encoded in bytes in a variety of ways.

The first method, `newBufferedReader()` , reads the file specified at the `Path` location using a `BufferedReader` object.

```
var path = Path.of("/animals/gopher.txt");
try (var reader = Files.newBufferedReader(path)) {
   String currentLine = null;
   while((currentLine = reader.readLine()) != null)
      System.out.println(currentLine);
}
```

This example reads the lines of the files using a `BufferedReader` and outputs the contents to the screen. As you shall see shortly, there are other methods that do this without having to use an I/O stream.

The second method, `newBufferedWriter()` , writes to a file specified at the `Path` location using a `BufferedWriter` .

```
var list = new ArrayList<String>();
list.add("Smokey");
list.add("Yogi");

var path = Path.of("/animals/bear.txt");
try (var writer = Files.newBufferedWriter(path)) {
   for(var line : list) {
      writer.write(line);
      writer.newLine();
   }
}
```

This code snippet creates a new file with two lines of text in it. Did you notice that both of these methods use buffered streams rather than low-level file streams? As we mentioned in Chapter 19, the buffered stream classes are much more performant, especially when working with files.

Reading a File with *readAllLines()*

The `Files` class includes a convenient method for turning the lines of a file into a `List`.

```
public static List<String> readAllLines(Path path) throws IOException
```

The following sample code reads the lines of the file and outputs them to the user:

```
var path = Path.of("/animals/gopher.txt");
final List<String> lines = Files.readAllLines(path);
lines.forEach(System.out::println);
```

Be aware that the entire file is read when `readAllLines()` is called, with the resulting `List<String>` storing all of the contents of the file in memory at once. If the file is significantly large, then you may trigger an `OutOfMemoryError` trying to load all of it into memory. Later in the chapter, we will revisit this method and present a stream-based NIO.2 method that can operate with a much smaller memory footprint.

Reviewing *Files* Methods

Table 20.4 shows the `static` methods in the `Files` class that you should be familiar with for the exam.

TABLE 20.4 *Files* methods

`boolean exists(Path, LinkOption…)`	`Path move(Path, Path, CopyOption…)`
`boolean isSameFile(Path, Path)`	`void delete(Path)`
`Path createDirectory(Path, FileAttribute<?>…)`	`boolean deleteIfExists(Path)`
`Path createDirectories(Path, FileAttribute<?>…)`	`BufferedReader newBufferedReader(Path)`
`Path copy(Path, Path, CopyOption…)`	`BufferedWriter newBufferedWriter( Path, OpenOption…)`
`long copy(InputStream, Path, CopyOption…)`	`List<String> readAllLines(Path)`
`long copy(Path, OutputStream)`	

All of these methods except `exists()` declare `IOException` .

Managing File Attributes

The `Files` class also provides numerous methods for accessing file and directory metadata, referred to as *file attributes*. A file attribute is data about a file within the system, such as its size and visibility, that is not part of the file contents. In this section, we'll show how to read file attributes individually or as a single streamlined call.

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.

Reading Common Attributes with *isDirectory()*, *isSymbolicLink()*, and *isRegularFile()*

The `Files` class includes three methods for determining type of a `Path`.

```
public static boolean isDirectory(Path path, LinkOption… options)

public static boolean isSymbolicLink(Path path)

public static boolean isRegularFile(Path path, LinkOption… options)
```

The `isDirectory()` and `isSymbolicLink()` methods should be self-explanatory, although `isRegularFile()` warrants some discussion. Java defines a *regular file* as one that can contain content, as opposed to a symbolic link, directory, resource, or other nonregular file that may be present in some operating systems. If the symbolic link points to an actual file, Java will perform the check on the target of the symbolic link. In other words, it is possible for `isRegularFile()` to return `true` for a symbolic link, as long as the link resolves to a regular file.

Let's take a look at some sample code.

```
System.out.print(Files.isDirectory(Paths.get("/canine/fur.jpg")));
System.out.print(Files.isSymbolicLink(Paths.get("/canine/coyote")));
System.out.print(Files.isRegularFile(Paths.get("/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 alternatively a symbolic link that points to a regular file.

NOTE

While most methods in the `Files` class declare `IOException`, these three methods do not. They return `false` if the path does not exist.

Checking File Accessibility with *isHidden()*, *isReadable()*, *isWritable()*, and *isExecutable()*

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.

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

```
public static boolean isReadable(Path path)
```

```
public static boolean isWritable(Path path)
```

```
public static boolean isExecutable(Path path)
```

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 sample usage of each method:

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

If the `walrus.txt` exists and is hidden within the file system, then 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 is able to be modified. Finally, the last example prints `true` if the file is able to 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 isHidden(), these methods do not declare any checked exceptions and return `false` if the file does not exist.

Reading File Size with *size()*

The `Files` class includes a method to determine the size of the file in bytes.

```
public static long size(Path path) throws IOException
```

The size returned by this method represents the conceptual size of the data, and this may differ from the actual size on the persistent storage device. The following is a sample call to the `size()` method:

```
System.out.print(Files.size(Paths.get("/zoo/animals.txt")));
```

The `Files.size()` method is defined only on files. Calling `Files.size()` on a directory is undefined, and the result depends on the file system. If you need to determine the size of a directory and its contents, you'll need to walk the directory tree. We'll show you how to do this later in the chapter.

Checking for File Changes with *getLastModifiedTime()*

Most operating systems support tracking a last-modified date/time value with each file. Some applications use this to determine when the file's contents should be read again. In the majority of circumstances, it is a lot faster to check a single file metadata attribute than to reload the entire contents of the file.

The `Files` class provides the following method to retrieve the last time a file was modified:

```
public static FileTime getLastModifiedTime(Path path,
   LinkOption… options) throws IOException
```

The method returns a `FileTime` object, which represents a timestamp. For convenience, it has a `toMillis()` method that returns the epoch time, which is the number of milliseconds since 12 a.m. UTC on January 1, 1970.

The following shows how to print the last modified value for a file as an epoch value:

```
final Path path = Paths.get("/rabbit/food.jpg");
System.out.println(Files.getLastModifiedTime(path).toMillis());
```

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, then 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 20.5 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 20.5 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 with *readAttributes()*

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 = Paths.get("/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.

Modifying Attributes with *getFileAttributeView()*

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 = Paths.get("/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.

NOTE

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.

Applying Functional Programming

We saved the best for last! In this part of the chapter, we'll combine everything we've presented so far with functional programming to perform extremely powerful file operations, often with only a few lines of code. The `Files` class includes some incredibly useful Stream API methods that operate on files, directories, and directory trees.

Listing Directory Contents

Let's start with a simple Stream API method. The following `Files` method lists the contents of a directory:

```
public static Stream<Path> list(Path dir) throws IOException
```

The `Files.list()` is similar to the `java.io.File` method `listFiles()`, except that it returns a `Stream<Path>` rather than a `File[]`. Since streams use lazy evaluation, this means the method will load each path element as needed, rather than the entire directory at once.

Printing the contents of a directory is easy.

```
try (Stream<Path> s = Files.list(Path.of("/home"))) {
   s.forEach(System.out::println);
}
```

Let's do something more interesting, though. Earlier, we presented the `Files.copy()` method and showed that it only performed a shallow copy of a directory. We can use the `Files.list()` to perform a deep copy.

```
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 it be a file or a directory. If it is a directory, then 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. What if the method comes across a symbolic link? Don't worry, we'll address that topic in the next section. For now, you just need to know the JVM will not follow symbolic links when using this stream method.

CLOSING THE STREAM

Did you notice that in the last two code samples, we put our `Stream` objects inside a try-with-resources method? The NIO.2 stream-based methods open a connection to the file system *that must be properly closed*, else 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 completed.

If you assumed 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, but 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 15.

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

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. When we originally described a directory in Chapter 19, we mentioned that it was 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, as shown in Figure 20.5. In this model, a directory is a branch or internal node, and a file is a leaf node.

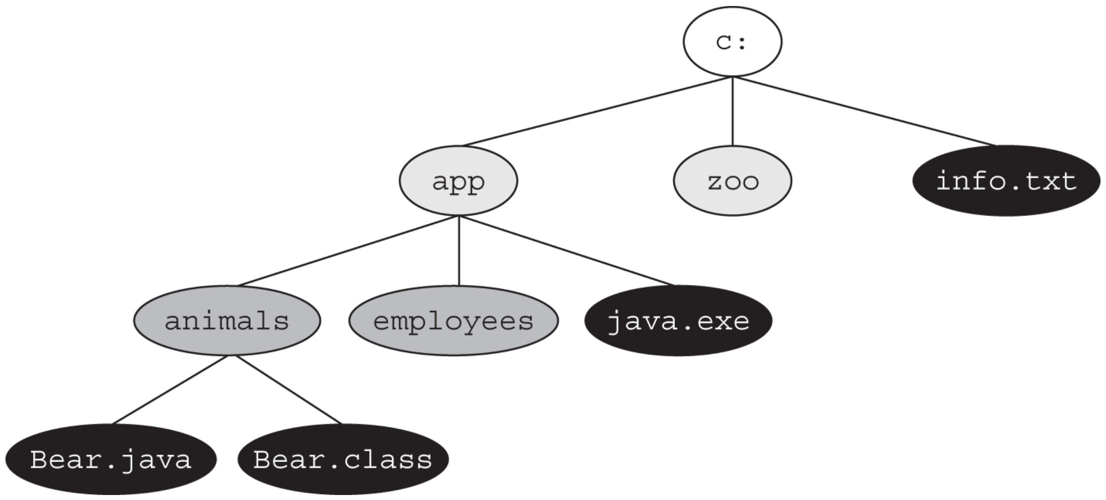

FIGURE 20.5 File and directory as a tree structure

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 when 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!

DON'T USE *DIRECTORYSTREAM* AND *FILEVISITOR*

While browsing the NIO.2 Javadocs, you may come across methods that use the `DirectoryStream` and `FileVisitor` classes to traverse a directory. These methods predate the existence of the Streams API and were even required knowledge for older Java certification exams. Even worse, despite its name, `DirectoryStream` is not a Stream API class.

The best advice we can give you is to not use them. The newer Stream API–based methods are superior and accomplish the same thing often with much less code.

Selecting a Search Strategy

There are two common strategies 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 down 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 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 Streams API methods use depth-first searching with a depth limit, which can be optionally changed.

Walking a Directory with *walk()*

That's enough background information; let's get to more Steam 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.

NOTE

Java uses an `int` for its maximum depth rather than a `long` because most file systems do not support path values deeper than what can be stored in an `int`. In other words, using `Integer.MAX_VALUE` is effectively like using an infinite value, since you would be hard-pressed to find a stable file system where this limit is exceeded.

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) {
      // Handle exception
   }
   return 0L;
}

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. We can print the data using the `format()` method we saw in the previous chapter:

```
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 Directory Tree Size: 15.30 megabytes
```

Applying a Depth Limit

Let's say our directory tree was 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 `5` 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, then every file in the system would 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 20.6, with the symbolic link `/birds/robin/allBirds` that points to `/birds`.

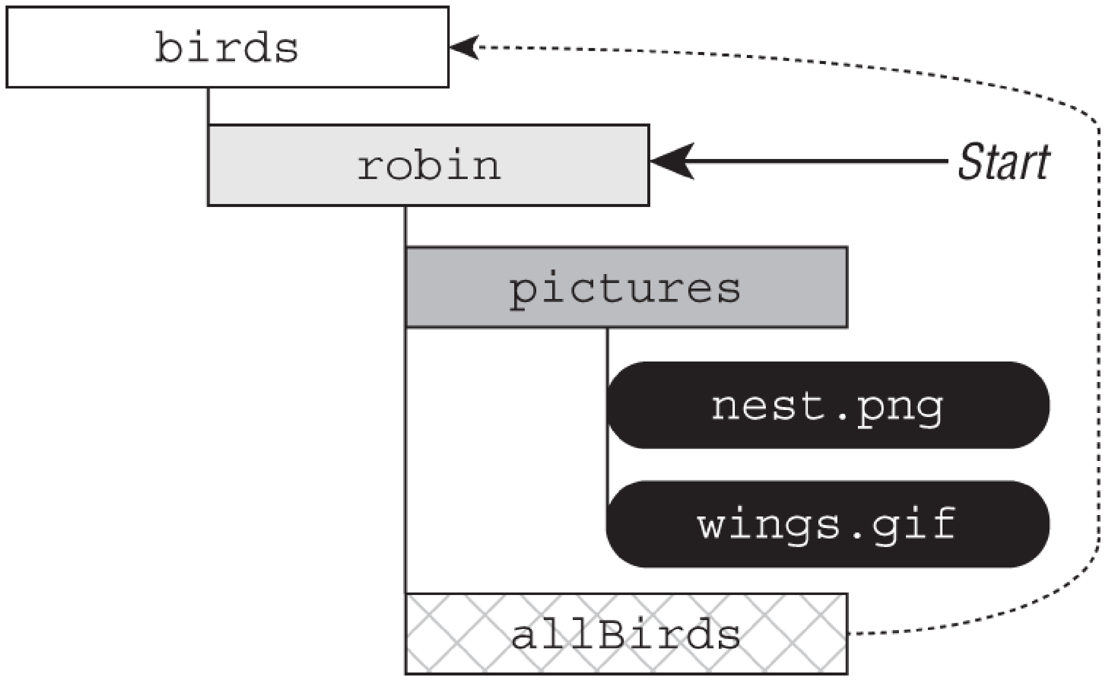

FIGURE 20.6 File system with cycle

What happens if we try to traverse this tree and follow all symbolic links, starting with `/birds/robin`? Table 20.6 shows the paths visited after walking a depth of `3`. For simplicity, we'll walk the tree in a breadth-first ordering, *although a cycle occurs regardless of the search strategy used.*

TABLE 20.6 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/nest.gif`
2	`/birds/robin/allBirds/robin` **➢** `/birds/robin`
3	`/birds/robin/allBirds/robin/pictures` ➢ `/birds/robin/pictures`
3	`/birds/robin/allBirds/robin/pictures/allBirds` ➢ `/birds/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 with *find()*

In the previous example, we applied a filter to the `Stream<Path>` object to filter the results, although NIO.2 provides 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 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, NIO.2 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:

```
Path path = Paths.get("/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.

Reading a File with *lines()*

Earlier in the chapter, we presented `Files.readAllLines()` and commented that using it to read a very large file could result in an `OutOfMemoryError` problem. Luckily, NIO.2 solves this problem with a Stream API method.

```
public static Stream<String> lines(Path path) throws IOException
```

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.

```
Path path = Paths.get("/fish/sharks.log");
try (var s = Files.lines(path)) {
   s.forEach(System.out::println);
}
```

Taking things one step further, we can leverage other stream methods for a more powerful example.

```
Path path = Paths.get("/fish/sharks.log");
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, functional programming plus NIO.2 gives us 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(Paths.get("birds.txt")).forEach(System.out::println);
Files.lines(Paths.get("birds.txt")).forEach(System.out::println);
```

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

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(Paths.get("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.

Comparing Legacy java.io.File and NIO.2 Methods

We conclude this chapter with Table 20.7, which shows a comparison between some of the legacy `java.io.File` methods described in Chapter 19 and the new NIO.2 methods described in this chapter. In this table, `file` refers to an instance of the `java.io.File` class, while `path` and `otherPath` refer to instances of the NIO.2 `Path` interface.

TABLE 20.7 Comparison of *java.io.File* and NIO.2 methods

Legacy I/O *File* method	NIO.2 method
`file.delete()`	`Files.delete(path)`
`file.exists()`	`Files.exists(path)`
`file.getAbsolutePath()`	`path.toAbsolutePath()`
`file.getName()`	`path.getFileName()`
`file.getParent()`	`path.getParent()`
`file.isDirectory()`	`Files.isDirectory(path)`
`file.isFile()`	`Files.isRegularFile(path)`
`file.lastModified()`	`Files.getLastModifiedTime(path)`
`file.length()`	`Files.size(path)`
`file.listFiles()`	`Files.list(path)`
`file.mkdir()`	`Files.createDirectory(path)`
`file.mkdirs()`	`Files.createDirectories(path)`
`file.renameTo(otherFile)`	`Files.move(path,otherPath)`

Bear in mind that a number of methods and features are available in NIO.2 that are not available in the legacy API, such as support for symbolic links, setting file system–specific attributes, and so on. The NIO.2 is a more developed, much more powerful API than the legacy `java.io.File` class.

Summary

This chapter introduced NIO.2 for working with files and directories using the `Path` interface. For the exam, you need to know what the NIO.2 `Path` interface is and how it differs from the legacy `java.io.File` class. You should be familiar with how to create and use `Path` objects, including how to combine or resolve them with other `Path` objects.

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 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 requires a read-only attribute type, while the second method requires an updatable view type. It also allows NIO.2 to support operating system–specific file attributes.

Finally, 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.

Exam Essentials

- **Understand how to create *Path* objects.** An NIO.2 `Path` instance is an immutable object that is commonly created from the factory method `Paths.get()` or `Path.of()`. It can also be created from `FileSystem`, java.net `.URI`, or `java.io.File` instances. The `Path` interface includes many instance methods for reading and manipulating the abstract path value.
- **Be able to manipulate *Path* objects.** NIO.2 includes numerous methods for viewing, manipulating, and combining `Path` values. It also includes methods to eliminate redundant or unnecessary path symbols.

Most of the methods defined on `Path` do not declare any checked exceptions and do not require the path to exist within the file system, with the exception of `toRealPath()`.

- **Be able to operate on files and directories using the *Files* class.** The NIO.2 `Files` helper class includes methods that operate on real files and directories within the file system. For example, it can be used to check whether a file exists, copy/move a file, create a directory, or delete a directory. Most of these methods declare `IOException`, since the path may not exist, and take a variety of varargs options.
- **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 modified select attributes.
- **Be able to operate on directories using functional programming.** NIO.2 includes the Stream API for manipulating directories. The `Files.list()` method iterates over the contents of a single directory, while the `Files.walk()` method lazily traverses a directory tree in a depth-first manner. The `Files.find()` method also traverses a directory but requires a filter to be applied. Both `Files.walk()` and `Files.find()` support a search depth limit. Both methods will also throw an exception if they are directed to follow symbolic links and detect a cycle.
- **Understand the difference between *readAllLines()* and *lines()*.** The `Files.readAllLines()` method reads all the lines of of a file into memory and returns the result as a `List<String>`. The `Files.lines()` method lazily reads the lines of a file, instead returning a functional programming `Stream<Path>` object. While both methods will correctly read the lines of a file, `lines()` is considered safer for larger files since it does not require the entire file to be stored in memory.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. What is the output of the following code?

```
4: var path = Path.of("/user/./root","../kodiacbear.txt");
5: path.normalize().relativize("/lion");
6: System.out.println(path);
```

   A. `../../lion`
   B. `/user/kodiacbear.txt`
   C. `../user/kodiacbear.txt`
   D. `/user/./root/../kodiacbear.txt`
   E. The code does not compile.
   F. None of the above

2. For which values of `path` sent to this method would it be possible for the following code to output `Success`? (Choose all that apply.)

```
public void removeBadFile(Path path) {
   if(Files.isDirectory(path))
      System.out.println(Files.deleteIfExists(path)
         ? "Success": "Try Again");
}
```

   A. `path` refers to a regular file in the file system.
   B. `path` refers to a symbolic link in the file system.
   C. `path` refers to an empty directory in the file system.
   D. `path` refers to a directory with content in the file system.
   E. `path` does not refer to a record that exists within the file system.
   F. The code does not compile.

3. What is the result of executing the following code? (Choose all that apply.)

```
4: var p = Paths.get("sloth.schedule");
5: var a = Files.readAttributes(p, BasicFileAttributes.class);
6: Files.mkdir(p.resolve(".backup"));
```

```
7: if(a.size()>0 && a.isDirectory()) {
8:    a.setTimes(null,null,null);
9: }
```

A. It compiles and runs without issue.
B. The code will not compile because of line 5.
C. The code will not compile because of line 6.
D. The code will not compile because of line 7.
E. The code will not compile because of line 8.
F. None of the above

4. If the current working directory is `/user/home` , then what is the output of the following code?

```
var p = Paths.get("/zoo/animals/bear/koala/food.txt");
System.out.print(p.subpath(1,3).getName(1).toAbsolutePath());
```

A. `bear`
B. `animals`
C. `/user/home/bear`
D. `/user/home/animals`
E. `/user/home/food.txt`
F. The code does not compile.
G. The code compiles but throws an exception at runtime.

5. Assume `/kang` exists as a symbolic link to the directory `/mammal/kangaroo` within the file system. Which of the following statements are correct about this code snippet? (Choose all that apply.)

```
var path = Paths.get("/kang");
if(Files.isDirectory(path) && Files.isSymbolicLink(path))
   Files.createDirectory(path.resolve("joey"));
```

A. A new directory will always be created.
B. A new directory may be created.
C. If the code creates a directory, it will be reachable at `/kang/joey` .

D. If the code creates a directory, it will be reachable at `/mammal/joey` .
E. The code does not compile.
F. The code will compile but always throws an exception at runtime.

6. Assume that the directory `/animals` exists and is empty. What is the result of executing the following code?

```
Path path = Path.of("/animals");
try (var z = Files.walk(path)) {
   boolean b = z
      .filter((p,a) -> a.isDirectory() && !path.equals(p)) // x
      .findFirst().isPresent();  // y
   System.out.print(b ? "No Sub": "Has Sub");
}
```

A. It prints `No Sub` .
B. It prints `Has Sub` .
C. The code will not compile because of line `x` .
D. The code will not compile because of line `y` .
E. The output cannot be determined.
F. It produces an infinite loop at runtime.

7. If the current working directory is `/zoo` and the path `/zoo/turkey` does not exist, then what is the result of executing the following code? (Choose all that apply.)

```
Path path = Paths.get("turkey");
if(Files.isSameFile(path,Paths.get("/zoo/turkey"))) // z1
   Files.createDirectories(path.resolve("info"));     // z2
```

A. The code compiles and runs without issue, but it does not create any directories.
B. The directory `/zoo/turkey` is created.
C. The directory `/zoo/turkey/info` is created.
D. The code will not compile because of line `z1` .
E. The code will not compile because of line `z2` .
F. It compiles but throws an exception at runtime.

8. Which of the following correctly create `Path` instances? (Choose all that apply.)
   - A. `new Path("jaguar.txt")`
   - B. `FileSystems.getDefault() .getPath("puma.txt")`
   - C. `Path.get("cats","lynx.txt")`
   - D. `new java.io.File("tiger.txt").toPath()`
   - E. `new FileSystem().getPath("lion")`
   - F. `Paths.getPath("ocelot.txt")`
   - G. `Path.of(Path.of(".").toUri())`
9. What is the output of the following code?

```
var path1 = Path.of("/pets/../cat.txt");
var path2 = Paths.get("./dog.txt");
System.out.println(path1.resolve(path2));
System.out.println(path2.resolve(path1));
```

   - A. `/cats.txt`
     `/dog.txt`
   - B. `/cats.txt/dog.txt`
     `/cat.txt`
   - C. `/pets/../cat.txt/./dog.txt`
     `/pets/../cat.txt`
   - D. `/pets/../cat.txt/./dog.txt`
     `./dog.txt/pets/../cat.txt`
   - E. None of the above
10. What are some advantages of using `Files.lines()` over `Files.readAllLines()` ? (Choose all that apply.)
   - A. It is often faster.
   - B. It can be run with little memory available.
   - C. It can be chained with functional programming methods like `filter()` and `map()` directly.
   - D. It does not modify the contents of the file.
   - E. It ensures the file is not read-locked by the file system.
   - F. There are no differences, because one method is a pointer to the other.

11. Assume `monkey.txt` is a file that exists in the current working directory. Which statements about the following code snippet are correct? (Choose all that apply.)

```
Files.move(Path.of("monkey.txt"), Paths.get("/animals"),
   StandardCopyOption.ATOMIC_MOVE,
   LinkOption.NOFOLLOW_LINKS);
```

A. If `/animals/monkey.txt` exists, then it will be overwritten at runtime.
B. If `/animals` exists as an empty directory, then `/animals/monkey.txt` will be the new location of the file.
C. If `monkey.txt` is a symbolic link, then the file it points to will be moved at runtime.
D. If the move is successful and another process is monitoring the file system, then it will not see an incomplete file at runtime.
E. The code will always throw an exception at runtime.
F. None of the above

12. What are some advantages of NIO.2 over the legacy `java.io.File` class for working with files? (Choose three.)
A. NIO.2 supports file system–dependent attributes.
B. NIO.2 includes a method to list the contents of a directory.
C. NIO.2 includes a method to traverse a directory tree.
D. NIO.2 includes a method to delete an entire directory tree.
E. NIO.2 includes methods that are aware of symbolic links.
F. NIO.2 supports sending emails.

13. For the `copy()` method shown here, assume that the source exists as a regular file and that the target does not. What is the result of the following code?

```
var p1 = Path.of(".","/","goat.txt").normalize(); // k1
var p2 = Path.of("mule.png");
Files.copy(p1, p2, StandardCopyOption.COPY_ATTRIBUTES); //k2
System.out.println(Files.isSameFile(p1, p2));
```

A. It will output `false`.
B. It will output `true`.
C. It does not compile because of line `k1`.
D. It does not compile because of line `k2`.
E. None of the above

14. Assume `/monkeys` exists as a directory containing multiple files, symbolic links, and subdirectories. Which statement about the following code is correct?

```
var f = Path.of("/monkeys");
try (var m =
      Files.find(f, 0, (p,a) -> a.isSymbolicLink())) { // y1
   m.map(s -> s.toString())
      .collect(Collectors.toList())
      .stream()
      .filter(s -> s.toString().endsWith(".txt")) // y2
      .forEach(System.out::println);
}
```

A. It will print all symbolic links in the directory tree ending in `.txt`.
B. It will print the target of all symbolic links in the directory ending in `.txt`.
C. It will print nothing.
D. It does not compile because of line `y1`.
E. It does not compile because of line `y2`.
F. It compiles but throws an exception at runtime.

15. Which NIO.2 method is most similar to the legacy `java.io.File` method `listFiles`?
A. `Path.listFiles()`
B. `Files.dir()`
C. `Files.ls()`
D. `Files.files()`
E. `Files.list()`
F. `Files.walk()`

16. What are some advantages of using NIO.2's `Files.readAttributes()` method rather than reading attributes individually from a file?

(Choose all that apply.)

A. It can be used on both files and directories.

B. For reading a single attribute, it is often more performant.

C. It allows you to read symbolic links.

D. It makes fewer round-trips to the file system.

E. It can be used to access file system–dependent attributes.

F. For reading multiple attributes, it is often more performant.

17. Assuming the `/fox/food-schedule.csv` file exists with the specified contents, what is the expected output of calling `printData()` on it?

```
/fox/food-schedule.csv
6am,Breakfast
9am,SecondBreakfast
12pm,Lunch
6pm,Dinner

void printData(Path path) throws IOException {
   Files.readAllLines(path) // r1
      .flatMap(p -> Stream.of(p.split(","))) // r2
      .map(q -> q.toUpperCase())  // r3
      .forEach(System.out::println);
}
```

A. The code will not compile because of line `r1`.

B. The code will not compile because of line `r2`.

C. The code will not compile because of line `r3`.

D. It throws an exception at runtime.

E. It does not print anything at runtime.

F. None of the above

18. What are some possible results of executing the following code? (Choose all that apply.)

```
var x = Path.of("/animals/fluffy/..");
Files.walk(x.toRealPath().getParent()) // u1
   .map(p -> p.toAbsolutePath().toString()) // u2
   .filter(s -> s.endsWith(".java")) // u3
```

```
      .collect(Collectors.toList())
      .forEach(System.out::println);
```

A. It prints some files in the root directory.

B. It prints all files in the root directory.

C. `FileSystemLoopException` is thrown at runtime.

D. Another exception is thrown at runtime.

E. The code will not compile because of line `u1`.

F. The code will not compile because of line `u2`.

19. Assuming the directories and files referenced exist and are not symbolic links, what is the result of executing the following code?

```
var p1 = Path.of("/lizard",".")
   .resolve(Path.of("walking.txt"));
var p2 = new File("/lizard/././actions/../walking.txt")
   .toPath();
System.out.print(Files.isSameFile(p1,p2));
System.out.print(" ");
System.out.print(p1.equals(p2));
System.out.print(" ");
System.out.print(p1.normalize().equals(p2.normalize()));
```

A. `true true true`

B. `false false false`

C. `false true false`

D. `true false true`

E. `true false false`

F. The code does not compile.

20. Assuming the current directory is `/seals/harp/food`, what is the result of executing the following code?

```
final Path path = Paths.get(".").normalize();
int count = 0;
for(int i=0; i<path.getNameCount(); ++i) {
   count++;
}
System.out.println(count);
```

A. `0`
B. `1`
C. `2`
D. `3`
E. `4`
F. The code compiles but throws an exception at runtime.

21. Assume the `source` instance passed to the following method represents a file that exists. Also, assume `/flip/sounds.txt` exists as a file prior to executing this method. When this method is executed, which statement correctly copies the file to the path specified by `/flip/sounds.txt`?

```
void copyIntoFlipDirectory(Path source) throws IOException {
   var dolphinDir = Path.of("/flip");
   dolphinDir = Files.createDirectories(dolphinDir);
   var n = Paths.get("sounds.txt");
   ______________________;
}
```

A. `Files.copy(source, dolphinDir)`
B. `Files.copy(source, dolphinDir.resolve(n), StandardCopyOption.REPLACE_EXISTING)`
C. `Files.copy(source, dolphinDir, StandardCopyOption.REPLACE_EXISTING)`
D. `Files.copy(source, dolphinDir.resolve(n))`
E. The method does not compile, regardless of what is placed in the blank.
F. The method compiles but throws an exception at runtime, regardless of what is placed in the blank.

22. Assuming the path referenced by `m` exists as a file, which statements about the following method are correct? (Choose all that apply.)

```
void duplicateFile(Path m, Path x) throws Exception {
   var r = Files.newBufferedReader(m);
   var w = Files.newBufferedWriter(x,
            StandardOpenOption.APPEND);
        String currentLine = null;
        while ((currentLine = r.readLine()) != null)
            w.write(currentLine);
    }
```

A. If the path referenced by `x` does not exist, then it correctly copies the file.
B. If the path referenced by `x` does not exist, then a new file will be created.
C. If the path referenced `x` does not exist, then an exception will be thrown at runtime.
D. If the path referenced `x` exists, then an exception will be thrown at runtime.
E. The method contains a resource leak.
F. The method does not compile.