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

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

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

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

Referencing Files and Directories

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

Conceptualizing the File System

We start with the basics. Data is stored on persistent storage devices, such as hard disk drives and memory cards. A *file* within the storage device holds data. Files are organized into hierarchies using directories. A *directory* is a location that can contain files as well as other directories. When working with directories in Java, we often treat them like files. In fact, we use many of the same classes and interfaces to operate on files and directories. For example, a file and directory both can be renamed with the same Java method. Note that we often say *file* to mean *file or directory* in this chapter.

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

Next, the *root directory* is the topmost directory in the file system, from which all files and directories inherit. In Windows, it is denoted with a drive letter such as `C:\`, while on Linux, it is denoted with a single forward slash, `/`.

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

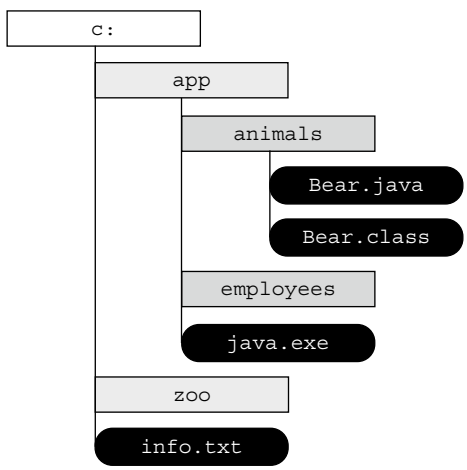
Operating System File Separators

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

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

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

FIGURE 14.1 Directory and file hierarchy



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

We use both absolute and relative paths to the file or directory within the file system. The *absolute path* of a file or directory is the full path from the root directory to the file or directory, including all subdirectories that contain the file or directory. Alternatively, the *relative path* of a file or directory is the path from the current working directory to the file or directory. For example, the following is an absolute path to the `Bear.java` file:

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

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

```
animals\Bear.java
```

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

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

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

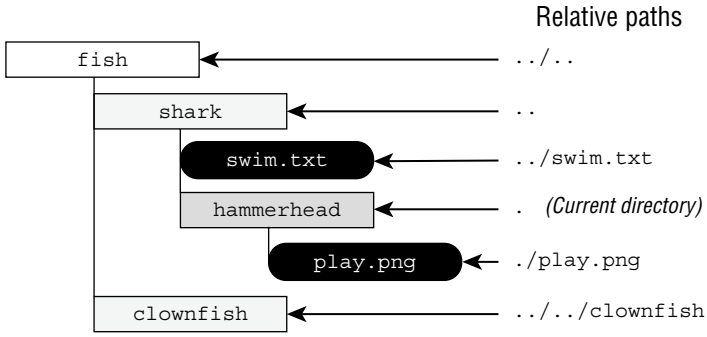
TABLE 14.1 File-system symbols

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

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

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

FIGURE 14.2 Relative paths using path symbols



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

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

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

Creating a *File* or *Path*

In order to do anything useful, you first need an object that represents the path to a particular file or directory on the file system. Using legacy I/O, this is the `java.io.File` class, whereas with NIO.2, it is the `java.nio.file.Path` interface. The `File` class and `Path` interface cannot read or write data within a file, although they are passed as a reference to other classes, as you see in this chapter.

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

Creating a *File*

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

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

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

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

All three create a `File` object that points to the same location on disk. If we passed `null` as the parent to the final constructor, it would be ignored, and the method would behave the same way as the single `String` constructor. For fun, we also show how to tell if the file exists on the file system.

Creating a *Path*

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

The simplest and most straightforward way to obtain a `Path` object is to use a `static` factory method defined on `Path` or `Paths`. All four of these examples point to the same reference on disk:

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

Path zooPath3 = Paths.get("/home/tiger/data/stripes.txt");
Path zooPath4 = Paths.get("/home", "tiger", "data", "stripes.txt");

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

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

As you can see, there are two ways of doing the same thing here. The `Path.of()` method was introduced in Java 11 as a `static` method on the interface. The `Paths` factory class also provides a `get()` method to do the same thing. Note the `s` at the end of the `Paths` class to distinguish it from the `Path` interface. We use `Path.of()` and `Paths.get()` interchangeably in this chapter.

You might notice that both the I/O and NIO.2 classes can interact with a URI. A uniform resource identifier (URI) is a string of characters that identifies a resource. It begins with a schema that indicates the resource type, followed by a path value such as `file://` for local file systems and `http://`, `https://`, and `ftp://` for remote file systems.

Switching between *File* and *Path*

Since `File` and `Path` both reference locations on disk, it is helpful to be able to convert between them. Luckily, Java makes this easy by providing methods to do just that:

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

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

Obtaining a *Path* from the *FileSystems* Class

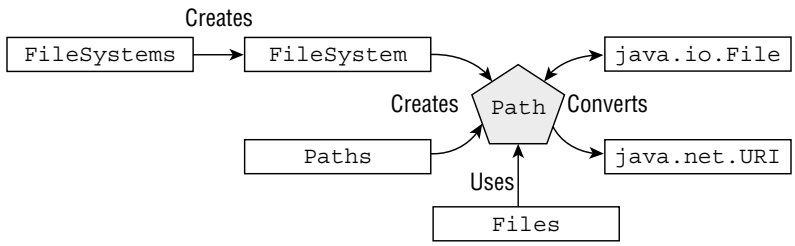
NIO.2 makes extensive use of creating objects with factory classes. The `FileSystems` class creates instances of the abstract `FileSystem` class. The latter includes methods for working with the file system directly. Both `Paths.get()` and `Path.of()` are shortcuts for this `FileSystem` method. Let's rewrite our earlier examples one more time to see how to obtain a `Path` instance the long way:

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

Reviewing I/O and NIO.2 Relationships

The model for I/O is smaller, and you only need to understand the `File` class. In contrast, NIO.2 has more features and makes extensive use of the factory pattern. You should become comfortable with this approach. Many of your interactions with NIO.2 will require two types: an abstract class or interface and a factory or helper class. Figure 14.3 shows the relationships among the classes and interface we have used in this chapter so far.

FIGURE 14.3 I/O and NIO.2 class and interface relationships



Review Figure 14.3 carefully. In particular, keep an eye on whether the class name is singular or plural. Classes with plural names include methods to create or operate on class/interface instances with singular names. Remember, as a convenience (and source of confusion), a `Path` can also be created from the `Path` interface using the `static` factory `of()` method.

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

Table 14.2 reviews the APIs we have covered for creating `java.io.File` and `java.nio.file.Path` objects. When reading the table, remember that `static` methods operate on the class/interface, while instance methods require an instance of an object. Be sure you know this well before proceeding with the rest of the chapter.

TABLE 14.2 Options for creating `File` and `Path`

Creates	Declared in	Method or Constructor

`File`	`File`	`public` **`File`**`(String pathname)`
		`public` **`File`**`(File parent, String child)`
		`public` **`File`**`(String parent, String child)`

`File`	`Path`	`public default File` **`toFile`**`()`
`Path`	`File`	`public Path` **`toPath`**`()`
`Path`	`Path`	`public static Path` **`of`**`(String first, String... more)` `public static Path` **`of`**`(URI uri)`
`Path`	`Paths`	`public static Path` **`get`**`(String first, String... more)` `public static Path` **`get`**`(URI uri)`
`Path`	`FileSystem`	`public Path` **`getPath`**`(String first, String... more)`
`FileSystem`	`FileSystems`	`public static FileSystem` **`getDefault`**`()`