Chapter 20 - NIO.2

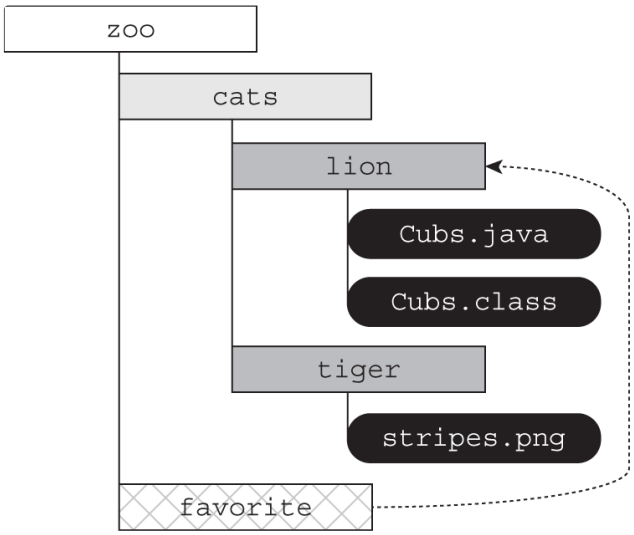
Introducing NIO.2

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.

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.

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`.



following paths reference the same file:

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

Creating Paths

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

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 `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");
```

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)
```

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

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.

```
// 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.

Obtaining a *Path* from the *FileSystem* Class

`FileSystems` class creates instances of the abstract `FileSystem` class.

```
// FileSystems factory method
```

```
2 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:

```
1 // FileSystem instance method
2 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."

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

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:

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

The following shows how such a method can be used:

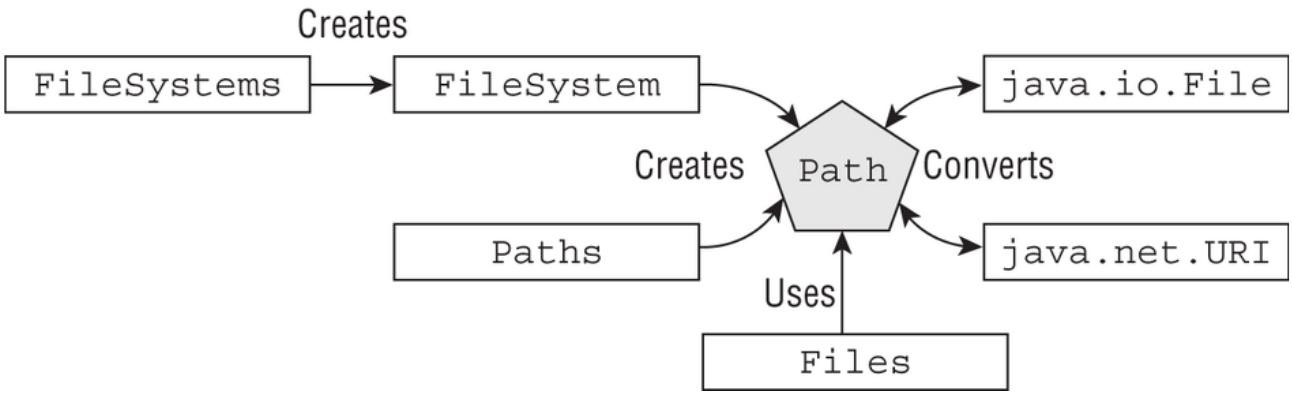
```
1    FileSystem fileSystem = FileSystems.getFileSystem(
2       new URI("http://www.selikoff.net"));
3    Path path = fileSystem.getPath("duck.txt");
```

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.

```
1 // Path to File, using Path instance method
2 public default File toFile()
3
4 // File to Path, using java.io.File instance method
5 public Path toPath()
```

Reviewing NIO.2 Relationships



Included in figure 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.

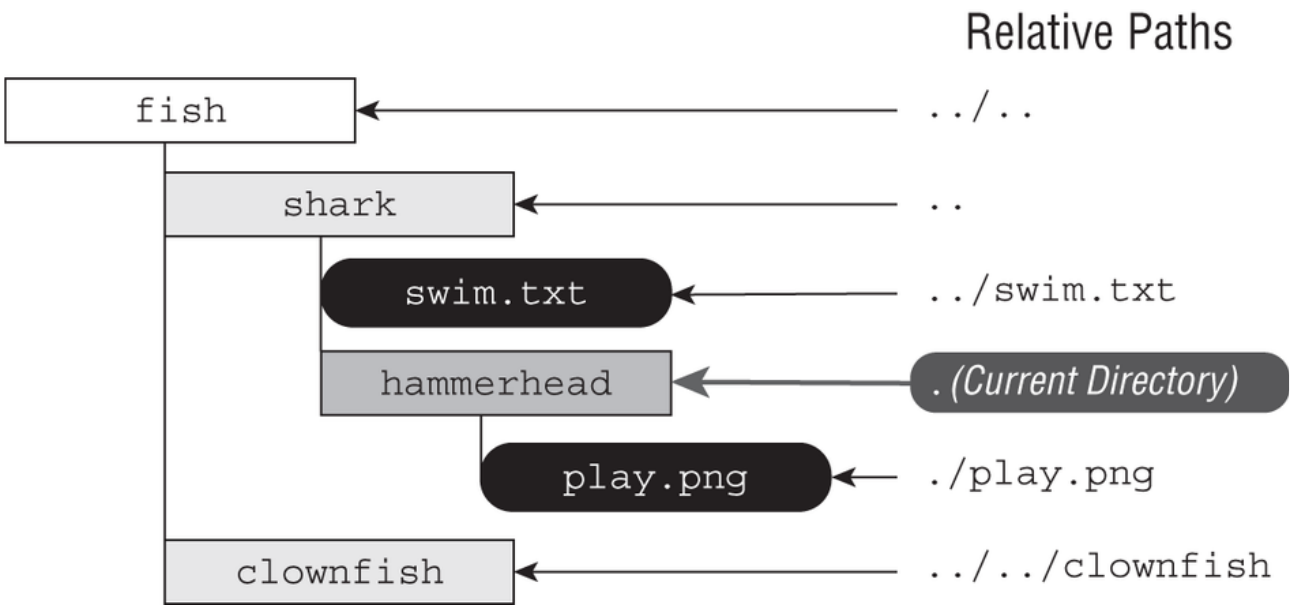
Understanding Common NIO.2 Features

Applying Path Symbols

A *path symbol* is a reserved series of characters that have special meaning within some file systems.

TABLE 20.1 File system symbols

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



Providing Optional Arguments

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.
		`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.

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.

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.

Interacting with Paths

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 `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));
}
```

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
```

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
```

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`

```
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
```

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()
```

```
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);
   }
}
```

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: .
```

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 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.

```
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());
```

```
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 object on which the `resolve()` method is invoked becomes the basis of the new `Path` object, with the input argument being appended onto the `Path`

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

The code snippet generates the following output:

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

```
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
```

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
```

If both path values are relative, then the `relativize()` method computes the paths as if they are in the same current working directory.

```
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

```
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()
```

```
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
```

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());
```

`/horse/food.txt`

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());
```

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…)`

Operating on Files and Directories

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.

Checking for Existence with *exists()*

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

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

```
1  var b1 = Files.exists(Paths.get("/ostrich/feathers.png"));
2  System.out.println("Path " + (b1 ? "Exists" : "Missing"));
3
4  var b2 = Files.exists(Paths.get("/ostrich"));
5  System.out.println("Path " + (b2 ? "Exists" : "Missing"));
```

The first example checks whether a file exists, while the second example checks whether a directory exists.

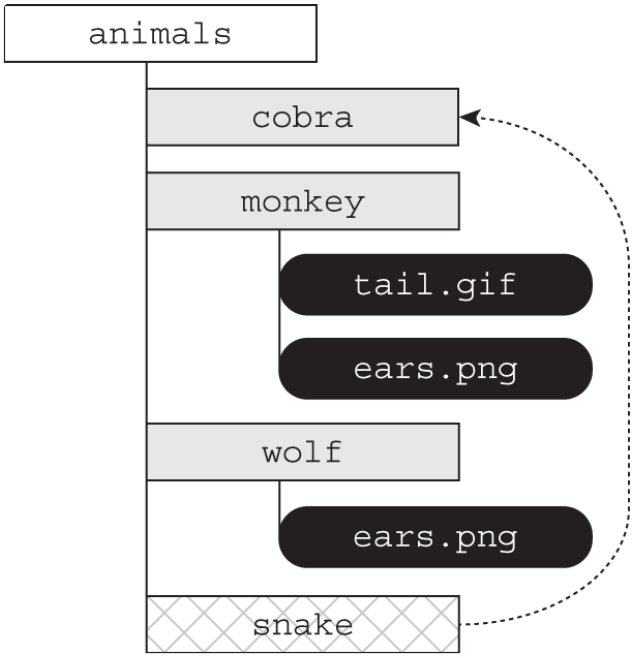
Remember, a file and directory may both have extensions. In the last example, the two paths could refer to two files or two directories. Unless the exam tells you whether the path refers to a file or directory, do not assume either.

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:

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

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.




```
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.

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.

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
```

```
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);
```

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
```

```
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);
```

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");
```

```
3 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:

```
1 var file = Paths.get("food.txt");
2 var directory = Paths.get("/enclosure/food.txt");
3 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.

```
1 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.

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

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

```
1 Files.move(Path.of("c:\\zoo"), Path.of("c:\\zoo-new"));
2
3 Files.move(Path.of("c:\\user\\addresses.txt"),
4    Path.of("c:\\zoo-new\\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`.

```
1 Files.move(Path.of("mouse.txt"), Path.of("gerbil.txt"),
2    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"));
```

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
```

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);
}
```

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();
   }
}
```

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

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)
```

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.

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)
```

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.

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 `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()*

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 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

A *view* is a group of related attributes for a particular file system type.

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.

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());
```

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);
```

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

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

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

Applying Functional Programming

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
```

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.

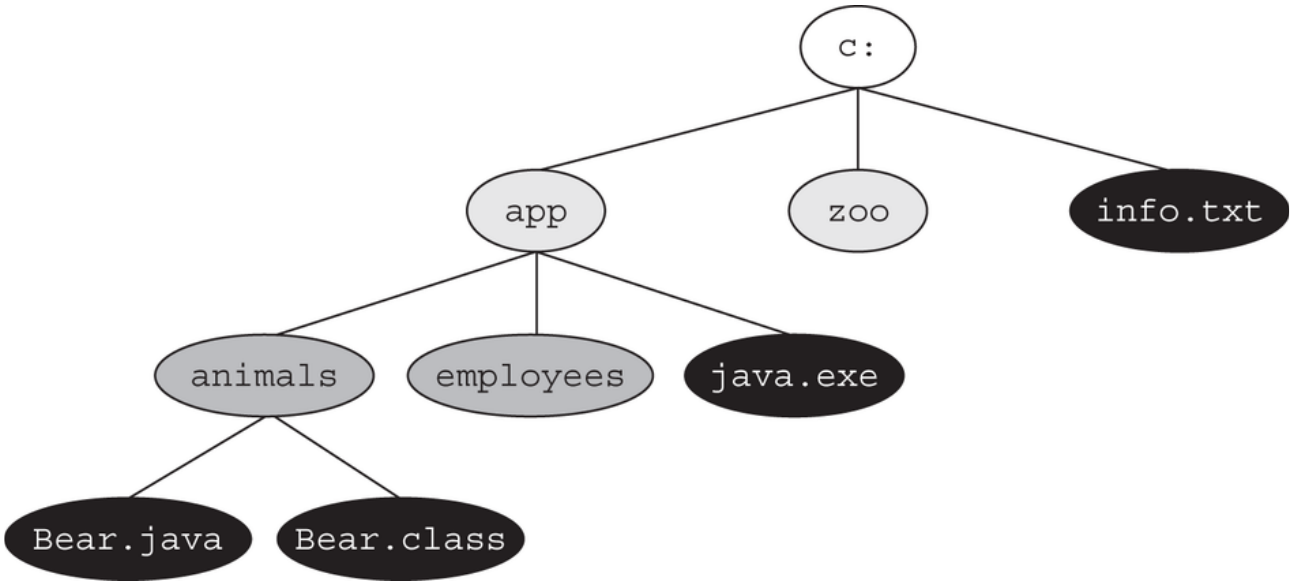
```
public void copyPath(Path source, Path target) {
   try {
      Files.copy(source, target);
      if(Files.isDirectory(source))
         try (Stream<Path> s = Files.list(source)) {
            s.forEach(p -> copyPath(p,
               target.resolve(p.getFileName())));
         }
   } catch(IOException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

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.

Traversing a Directory Tree

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.



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

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.

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
```

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.

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();
   }
}
```

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)) {
```

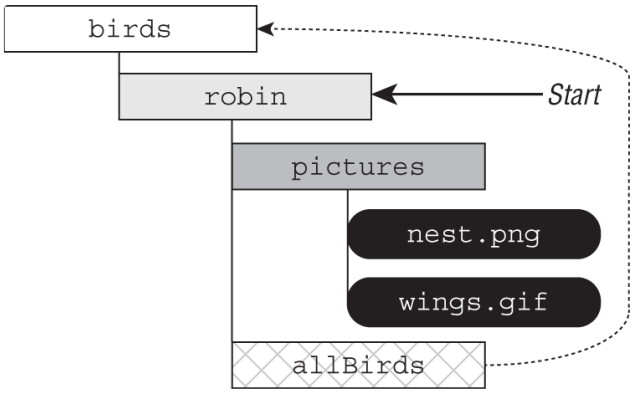
This new version checks for files only within `5` steps of the starting node. A depth value of `0` indicates the current path itself. Since the method calculates values only on files, you'd have to set a depth limit of at least `1` to get a nonzero result when this method is applied to a directory tree.

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)) {
```

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`.



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);
}
```

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.

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)`