5

I/O and NIO

CERTIFICATION OBJECTIVES

- Read and Write Data from the Console
- Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io Package
- Use Path Interface to Operate on File and Directory Paths
- Use Files Class to Check, Read, Delete, Copy, Move, Manage Metadata of a File or Directory
- Use Stream API with NIO.2

✓ Two-Minute Drill

Q&A Self Test

I/O (input/output) has been around since the beginning of Java. You could read and write files along with some other common operations. Then with Java 1.4, Java added more I/O functionality and cleverly named it NIO. That stands for "new I/O." Don't worry—you won't be asked about those Java 1.4 additions on the exam.

The APIs prior to Java 7 still had a few limitations when you had to write applications that focused heavily on files and file manipulation. Trying to write a little routine listing all the files created in the past day within a directory tree would have given you some headaches. There was no support for navigating directory trees, and just reading attributes of a file was also quite hard. As of Java 7, this whole routine is fewer than 15 lines of code!

Now what to name yet another I/O API? The name "new I/O" was taken, and "new new I/O" would just sound silly. Since the Java 7 functionality was added to package names that began with `java.nio`, the new name was NIO.2. For the purposes of this chapter and the exam, NIO is shorthand for NIO.2.

Since NIO (or NIO.2 if you like) builds on the original I/O, some of those concepts are still tested on the exam in addition to the new parts. Fortunately, you won't have to become a total I/O or NIO guru to do well on the exam. The intention of the exam team

was to include just the basic aspects of these technologies, and in this chapter, we cover *more* than you'll need to get through these objectives on the exam.

CERTIFICATION OBJECTIVE

File Navigation and I/O (OCP Objectives 8.1 and 8.2)

8.1 Read and write data from the console.
8.2 Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.

I/O has had a strange history with the OCP certification. It was included in all the versions of the exam, up to and including 1.2, then removed from the 1.4 exam, reintroduced for Java 5, extended for Java 6, and extended still more for Java 7 and 8.

I/O is a huge topic in general, and the Java APIs that deal with I/O in one fashion or another are correspondingly huge. A general discussion of I/O could include topics such as file I/O, console I/O, thread I/O, high-performance I/O, byte-oriented I/O, character-oriented I/O, I/O filtering and wrapping, serialization, and more. Luckily for us, the I/O topics included in the Java 8 exam are fairly well restricted to file I/O for characters and serialization.

Here's a summary of the I/O classes you'll need to understand for the exam:

- File The API says that the `File` class is "an abstract representation of file and directory pathnames." The `File` class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.
- FileReader This class is used to read character files. Its `read()` methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. `FileReader`s are usually *wrapped* by higher-level objects such as `BufferedReader`s, which improve performance and provide more convenient ways to work with the data.
- BufferedReader This class is used to make lower-level `Reader` classes like `FileReader` more efficient and easier to use. Compared to `FileReader`s, `BufferedReader`s read relatively large chunks of data from a file at once and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of times that time-intensive file-read operations are performed. In addition, `BufferedReader` provides more convenient methods, such as `readLine()`, that allow you to get the next line of characters from a file.

- FileWriter This class is used to write to character files. Its `write()` methods allow you to write character(s) or strings to a file. `FileWriters` are usually *wrapped* by higher-level `Writer` objects, such as `BufferedWriters` or `PrintWriters`, which provide better performance and higher-level, more flexible methods to write data.
- BufferedWriter This class is used to make lower-level classes like `FileWriters` more efficient and easier to use. Compared to `FileWriters`, `BufferedWriters` write relatively large chunks of data to a file at once, minimizing the number of times that slow file-writing operations are performed. The `BufferedWriter` class also provides a `newLine()` method to create platform-specific line separators automatically.
- PrintWriter This class has been enhanced significantly in Java 5. Because of newly created methods and constructors (like building a `PrintWriter` with a `File` or a `String`), you might find that you can use `PrintWriter` in places where you previously needed a `Writer` to be wrapped with a `FileWriter` and/or a `BufferedWriter`. New methods like `format()`, `printf()`, and `append()` make `PrintWriters` quite flexible and powerful.
- FileInputStream This class is used to read `bytes` from files and can be used for binary as well as text. Like `FileReader`, the `read()` methods are low-level, allowing you to read single `bytes`, a stream of `bytes`, or a fixed number of `bytes`. We typically use `FileInputStream` with higher-level objects such as `ObjectInputStream`.
- FileOutputStream This class is used to write `bytes` to files. We typically use `FileOutputStream` with higher-level objects such as `ObjectOutputStream`.
- ObjectInputStream This class is used to read an input stream and deserialize objects. We use `ObjectInputStream` with lower-level classes like `FileInputStream` to read from a file. `ObjectInputStream` works at a higher level so that you can read objects rather than characters or bytes. This process is called *deserialization.*

Classes with "Stream" in their name are used to read and write bytes, and Readers and Writers are used to read and write characters.

- ObjectOutputStream This class is used to write objects to an output stream and is used with classes like `FileOutputStream` to write to a file. This is called *serialization.* Like `ObjectInputStream`, `ObjectOutputStream` works at a higher level to write objects, rather than characters or bytes.
- Console This Java 6 convenience class provides methods to read input from the

console and write formatted output to the console.

Creating Files Using the File Class

Objects of type `File` are used to represent the actual files (but not the data in the files) or directories that exist on a computer's physical disk. Just to make sure we're clear, when we talk about an object of type `File`, we'll say `File`, with a capital *F*. When we're talking about what exists on a hard drive, we'll call it a file with a lowercase *f* (unless it's a variable name in some code). Let's start with a few basic examples of creating files, writing to them, and reading from them. First, let's create a new file and write a few lines of data to it:

```
import java.io.*;                                  // The section 8 objectives
                                                   // focus on classes from
                                                   // java.io
class Writer1 {
  public static void main(String [] args) {
    File file = new File("fileWrite1.txt");        // There's no
                                                   // file yet!
  }
}
```

If you compile and run this program, when you look at the contents of your current directory, you'll discover absolutely no indication of a file called `fileWrite1.txt`. When you make a new instance of the class `File`, *you're not yet making an actual file; you're just creating a filename.* Once you have a `File` *object,* there are several ways to make an actual file. Let's see what we can do with the `File` object we just made:

```
import java.io.*;

class Writer1 {
 public static void main(String [] args) {
   try {                                // warning: exceptions possible
     boolean newFile = false;
     File file = new File               // it's only an object
                 ("fileWrite1.txt");
     System.out.println(file.exists()); // look for a real file
     newFile = file.createNewFile();    // maybe create a file!
     System.out.println(newFile);       // already there?
     System.out.println(file.exists()); // look again
   } catch(IOException e) { }
 }
}
```

This produces the output

```
false
true
true
```

And also produces an empty file in your current directory. If you run the code a *second* time, you get the output

```
true
false
true
```

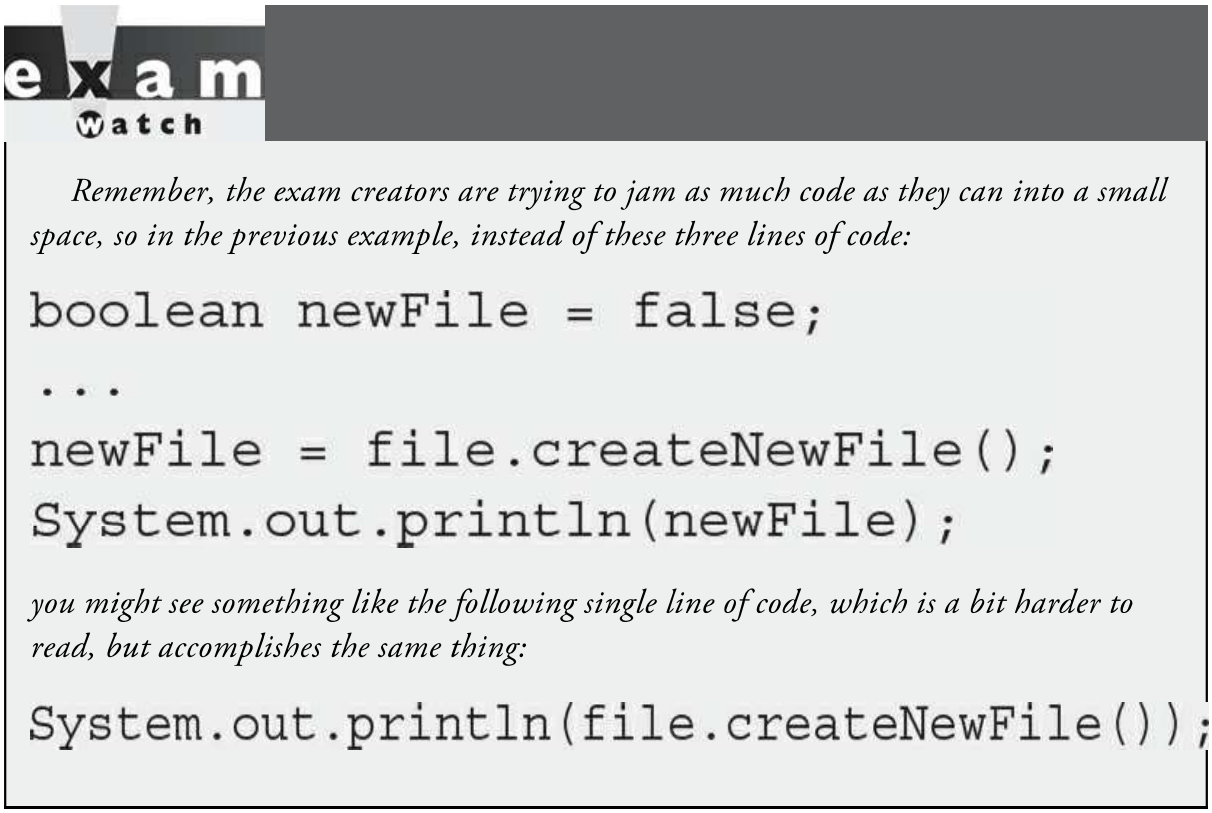
Let's examine these sets of output:

- First execution The first call to `exists()` returned `false`, which we expected… remember, `new File()` doesn't create a file on the disk! The `createNewFile()`

method created an actual file and returned `true`, indicating that a new file was created and that one didn't already exist. Finally, we called `exists()` again, and this time it returned `true`, indicating the file existed on the disk.

- Second execution The first call to `exists()` returns `true` because we built the file during the first run. Then the call to `createNewFile()` returns `false` since the method didn't create a file this time through. Of course, the last call to `exists()` returns `true`.

A couple of other new things happened in this code. First, notice that we had to put our file creation code in a try/catch. This is true for almost all of the file I/O code you'll ever write. I/O is one of those inherently risky things. We're keeping it simple for now and ignoring the exceptions, but we still need to follow the handle-or-declare rule, since most I/O methods declare checked exceptions. We'll talk more about I/O exceptions later. We used a couple of `File`'s methods in this code:



- boolean exists() This method returns `true` if it can find the actual file.
- boolean createNewFile() This method creates a new file if it doesn't already exist.

Using FileWriter and FileReader

In practice, you probably won't use the `FileWriter` and `FileReader` classes without wrapping them (more about "wrapping" very soon). That said, let's go ahead and do a little "naked" file I/O:

```
import java.io.*;
class Writer2 {
  public static void main(String [] args) {
    char[] in = new char[50];          // to store input
    int size = 0;
    try {
      File file = new File(            // just an object
                  "fileWrite2.txt");
      FileWriter fw =
                new FileWriter(file);  // create an actual file
                                       // & a FileWriter obj
      fw.write("howdy\nfolks\n");      // write characters to
                                       // the file
      fw.flush();                      // flush before closing
      fw.close();                      // close file when done
      FileReader fr =
                new FileReader(file);  // create a FileReader
                                       // object
      size = fr.read(in);              // read the whole file!
      System.out.print(size + " ");    // how many characters read
      for(char c : in)                 // print the array
        System.out.print(c);
      fr.close();                      // again, always close
    } catch(IOException e) { }
  }
}
```

which produces the output:

```
12 howdy
folks
```

Here's what just happened:

1. `FileWriter fw = new FileWriter(file)` did three things:
   a. It created a `FileWriter` reference variable, `fw`.
   b. It created a `FileWriter` object and assigned it to `fw`.
   c. It created an actual empty file out on the disk (and you can prove it).
2. We wrote 12 characters to the file with the `write()` method, and we did a `flush()` and a `close()`.
3. We made a new `FileReader` object, which also opened the file on disk for reading.
4. The `read()` method read the whole file, a character at a time, and put it into the `char[] in`.
5. We printed out the number of characters we read in `size`, and we looped through the `in` array, printing out each character we read, and then we closed the file.

Before we go any further, let's talk about `flush()` and `close()`. When you write data out to a stream, some amount of buffering will occur, and you never know for sure exactly when the last of the data will actually be sent. You might perform many write operations on a stream before closing it, and invoking the `flush()` method guarantees that the last of the data you thought you had already written actually gets out to the file. Whenever you're done using a file, either reading it or writing to it, you should invoke the `close()` method. When you are doing file I/O, you're using expensive and limited operating system resources, and so when you're done, invoking `close()` will free up those resources.

Now, back to our last example. This program certainly works, but it's painful in a couple of different ways:

1. When we were writing data to the file, we manually inserted line separators (in this case `\n`) into our data.
2. When we were reading data back in, we put it into a character array. It being an array and all, we had to declare its size beforehand, so we'd have 257been in trouble if we hadn't made it big enough! We could have read the data in one character at a time, looking for the end of the file after each `read()`, but that's pretty painful too.

Because of these limitations, we'll typically want to use higher-level I/O classes like `BufferedWriter` or `BufferedReader` in combination with `FileWriter` or

`FileReader`.

Using FileInputStream and FileOutputStream

Using `FileInputStream` and `FileOutputStream` is similar to using `FileReader` and `FileWriter`, except you're working with byte data instead of character data. That means you can use `FileInputStream` and `FileOutputStream` to read and write binary data as well as text data.

We've rewritten the previous example to use `FileInputStream` and `FileOutputStream`; the code does exactly the same thing, but because we're working with bytes instead of characters, we made a few small modifications, which we'll point out:

```
import java.io.*;
class Writer3 {
   public static void main(String [] args) {
      byte[] in = new byte[50];              // bytes, not chars!
      int size = 0;
      FileOutputStream fos = null;
      FileInputStream fis = null;
      File file = new File("fileWrite3.txt");
      try {
         fos = new FileOutputStream(file);   // create a FileOutputStream
         String s = "howdy\nfolks\n";
         fos.write(s.getBytes("UTF-8"));     // write characters (bytes)
                                             // to the file
         fos.flush();                        // flush before closing
         fos.close();                        // close file when done

         fis = new FileInputStream(file);    // create a FileInputStream
         size = fis.read(in);                // read the file into in
         System.out.print(size + " ");       // how many bytes read
         for(byte b : in) {                  // print the array
            System.out.print((char)b);
         }
         fis.close();                        // again, always close
      } catch(IOException e) {
         e.printStackTrace();
      }
   }
}
```

As you can see, this example is almost exactly like the previous one, except we're using bytes rather than chars. That means we convert the `String` we write to the file to bytes for the `write()` method, and when we read, we read into an array of `byte`s, rather than an array of `char`s, and convert each `byte` to a `char` before we print it.

And like the previous example, this one is painful in the same ways. You'll typically find you use higher-level I/O classes like `ObjectInputStream` and `ObjectOutputStream`, rather than `FileInputStream` and `FileOutputStream`, unless you really need to read binary data byte by byte.
We talk about `ObjectInputStream` and `ObjectOutputStream` in the section on serialization later in the chapter.

Combining I/O Classes

Java's entire I/O system was designed around the idea of using several classes in combination. Combining I/O classes is sometimes called *wrapping* and sometimes called *chaining*. The `java.io` package contains about 50 classes, 10 interfaces, and 15 exceptions. Each class in the package has a specific purpose (i.e., highly specialized), and the classes are designed to be combined with each other in countless ways to handle a wide variety of situations.

When it's time to do some I/O in real life, you'll undoubtedly find yourself poring over the `java.io` API, trying to figure out which classes you'll need and how to hook them together. For the exam, you'll need to do the same thing, but Oracle artificially reduced the API (phew!). In terms of studying for Exam Objective 8.2, we can imagine that the entire `java.io` package—consisting of the classes listed in Exam Objective 8.2 and summarized in Table 5-1—is our mini I/O API.

TABLE 5-1 `java.io` Mini API

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format(), printf() print(), println() write()
FileOutputStream	OutputStream	File String	close() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
FileInputStream	InputStream	File String	read() close()

Now let's say we want to find a less painful way to write data to a file and read the file's contents back into memory. Starting with the task of writing data to a file, here's a process for determining what classes we'll need and how we'll hook them together:

1. We know that ultimately we want to hook to a `File` object. So whatever other class or classes we use, one of them must have a constructor that takes an object of type `File`.
2. Find a method that sounds like the most powerful, easiest way to accomplish the task. When we look at Table 5-1 we can see that `BufferedWriter` has a `newLine()` method. That sounds a little better 259than having to manually embed a separator after each line, but if we look further, we see that `PrintWriter` has a method called `println()`. That sounds like the easiest approach of all, so we'll go with it.
3. When we look at `PrintWriter`'s constructors, we see that we can build a `PrintWriter` object if we have an object of type `File`, so all we need to do to create a `PrintWriter` object is the following:

```
File file = new File("fileWrite2.txt");   // create a File
PrintWriter pw = new PrintWriter(file);   // pass file to
                                          // the PrintWriter
                                          // constructor
```

Okay, time for a pop quiz. Prior to Java 5, `PrintWriter` did not have constructors that took either a `String` or a `File`. If you were writing some I/O code in Java 1.4, how would you get a `PrintWriter` to write data to a file? Hint: You can figure this out by studying the mini I/O API in Table 5-1.

Here's one way to go about solving this puzzle: First, we know that we'll create a `File` object on one end of the chain and that we want a `PrintWriter` object on the other end. We can see in Table 5-1 that a `PrintWriter` can also be built using a `Writer` object. Although `Writer` isn't a *class* we see in the table, we can see that several other classes extend `Writer`, which, for our purposes, is just as good; any class that extends `Writer` is a candidate. Looking further, we can see that `FileWriter` has the two attributes we're looking for:

- It can be constructed using a `File`.
- It extends `Writer`.

Given all of this information, we can put together the following code (remember, this is a Java 1.4 example):

```
File file = new File("fileWrite2.txt");  // create a File object
FileWriter fw = new FileWriter(file);    // create a FileWriter
                                         // that will send its
                                         // output to a File

PrintWriter pw = new PrintWriter(fw);    // create a PrintWriter
                                         // that will send its
                                         // output to a Writer

pw.println("howdy");                     // write the data
pw.println("folks");
```

At this point, it should be fairly easy to put together the code to more easily read data from the file back into memory. Again, looking through the table, we see a method called `readLine()` that sounds like a much better way to read data. Going through a similar process, we get the following code:

```
File file =
      new File("fileWrite2.txt");  // create a File object AND
                                   // open "fileWrite2.txt"
FileReader fr =
         new FileReader(file);     // create a FileReader to get
                                   // data from 'file'
BufferedReader br =
        new BufferedReader(fr);    // create a BufferReader to
                                   // get its data from a Reader
String data = br.readLine();       // read some data
```

You're almost certain to encounter exam questions that test your knowledge of how I/O classes can be chained. If you're not totally clear on this last section, we recommend using Table 5-1 as a reference and writing code to experiment with which chaining combinations are legal and which are illegal.

Working with Files and Directories

Earlier, we touched on the fact that the `File` class is used to create files and directories. In addition, `File`'s methods can be used to delete files, rename files, determine whether files exist, create temporary files, change a file's attributes, and differentiate between files and directories. A point that is often confusing is that an object of type `File` is used to represent *either a file or a directory.* We'll talk about both cases next.

We saw earlier that the statement

```
File file = new File("foo");
```

always creates a `File` object and then does one of two things:

1. If `"foo"` does NOT exist, no actual file is created.
2. If `"foo"` *does* exist, the new `File` object refers to the existing file.

Notice that `File file = new File("foo");` NEVER creates an actual file. There are two ways to create a file:

1. Invoke the `createNewFile()` method on a `File` object. For example:

```
File file = new File("foo");      // no file yet
file.createNewFile();             // make a file, "foo" which
                                  // is assigned to 'file'
```

2. Create a `Writer` or a `Stream`. Specifically, create a `FileWriter`, a `PrintWriter`, or a `FileOutputStream`. Whenever you create an instance of one of these classes, you automatically create a file, unless one already exists, for instance:

```
File file = new File("foo");       // no file yet
PrintWriter pw =
        new PrintWriter(file);     // make a PrintWriter object AND
                                   // make a file, "foo" to which
                                   // 'file' is assigned, AND assign
                                   // 'pw' to the PrintWriter
```

Creating a directory is similar to creating a file. Again, we'll use the convention of referring to an object of type `File` that represents an actual directory as a `Directory` object, with a capital *D* (even though it's of type `File`). We'll call an actual directory on a computer a directory, with a small *d*. Phew! As with creating a file, creating a directory is a two-step process; first we create a `Directory` (`File`) object; then we create an actual directory using the following `mkdir()` method:

```
File myDir = new File("mydir");   // create an object
myDir.mkdir();                    // create an actual directory
```

Once you've got a directory, you put files into it and work with those files:

```
File myFile = new File(myDir, "myFile.txt");
myFile.createNewFile();
```

This code is making a new file in a subdirectory. Since you provide the subdirectory to the constructor, from then on, you just refer to the file by its reference variable. In this case, here's a way that you could write some data to the file `myFile`:

```
PrintWriter pw = new PrintWriter(myFile);
pw.println("new stuff");
pw.flush();
pw.close();
```

Be careful when you're creating new directories! As we've seen, constructing a `Writer` or a `Stream` will often create a file for you automatically if one doesn't exist, but that's not true for a directory.

```
File myDir = new File("mydir");
// myDir.mkdir();                    // call to mkdir() omitted!
File myFile = new File(
              myDir, "myFile.txt");
myFile.createNewFile();              // exception if no mkdir!
```

This will generate an exception that looks something like

```
java.io.IOException: No such file or directory
```

You can refer a `File` object to an existing file or directory. For example, assume we already have a subdirectory called `existingDir` in which an existing file `existingDirFile.txt` resides. This file contains several lines of text. When we run the following code:

```
File existingDir = new File("existingDir");     // assign a dir
System.out.println(existingDir.isDirectory());

File existingDirFile = new File(
          existingDir, "existingDirFile.txt");  // assign a file
System.out.println (existingDirFile.isFile());
FileReader fr = new FileReader(existingDirFile);
BufferedReader br = new BufferedReader(fr);     // make a Reader

String s;
while( (s = br.readLine()) != null)             // read data
  System.out.println(s);

br.close();
```

the following output will be generated:

```
true
true
existing sub-dir data
line 2 of text
line 3 of text
```

Take special note of what the `readLine()` method returns. When there is no more data to read, `readLine()` returns a `null`—this is our signal to stop reading the file. Also, notice that we didn't invoke a `flush()` method. When reading a file, no flushing is required, so you won't even find a `flush()` method in a `Reader` kind of class.

In addition to creating files, the `File` class lets you do things like renaming and deleting files. The following code demonstrates a few of the most common ins and outs of deleting files and directories (via `delete()`) and renaming files and directories (via `renameTo()`):

```
File delDir = new File("deldir");       // make a directory
delDir.mkdir();

File delFile1 = new File(
          delDir, "delFile1.txt");      // add file to directory
delFile1.createNewFile();

File delFile2 = new File(
          delDir, "delFile2.txt");      // add file to directory
delFile2.createNewFile();
delFile1.delete();                      // delete a file
System.out.println("delDir is "
                + delDir.delete());     // attempt to delete
                                        // the directory
File newName = new File(
          delDir, "newName.txt");       // a new object
delFile2.renameTo(newName);             // rename file

File newDir = new File("newDir");       // rename directory
delDir.renameTo(newDir);
```

This outputs

```
delDir is false
```

and leaves us with a directory called `newDir` that contains a file called `newName.txt`. Here are some rules that we can deduce from this result:

- delete() You can't delete a directory if it's not empty, which is why the invocation `delDir.delete()` failed.
- renameTo() You must give the existing `File` object a valid new `File` object with the new name that you want. (If `newName` had been `null`, we would have gotten a

NullPointerException.)

- renameTo() It's okay to rename a directory, even if it isn't empty.

There's a lot more to learn about using the java.io package, but as far as the exam goes, we only have one more thing to discuss, and that is how to search for a file. Assuming we have a directory named searchThis that we want to search through, the following code uses the File.list() method to create a String array of files and directories. We then use the enhanced for loop to iterate through and print.

```
String[] files = new String[100];
File search = new File("searchThis");
files = search.list();                    // create the list

for(String fn : files)                    // iterate through it
  System.out.println("found " + fn);
```

On our system, we got the following output:

```
found dir1
found dir2
found dir3
found file1.txt
found file2.txt
```

Your results will almost certainly be different!

In this section, we've scratched the surface of what's available in the java.io package. Entire books have been written about this package, so we're obviously covering only a very small (but frequently used) portion of the API. On the other hand, if you understand everything we've covered in this section, you will be in great shape to handle any java.io questions you encounter on the exam, except for the Console class, which we'll cover next.

The java.io.Console Class

Java 6 added the java.io.Console class. In this context, the *console* is the physical device with a keyboard and a display (like your Mac or PC). If you're running Java SE 6 from the command line, you'll typically have access to a console object, to which you can get a reference by invoking System.console(). Keep in mind that it's possible for your Java

program to be running in an environment that doesn't have access to a console object, so be sure that your invocation of `System.console()` actually returns a valid console reference and not null.

The `Console` class makes it easy to accept input from the command line, both echoed and nonechoed (such as a password), and makes it easy to write formatted output to the command line. It's a handy way to write test engines for unit testing or if you want to support a simple but secure user interaction and you don't need a GUI.

On the input side, the methods you'll have to understand are `readLine` and `readPassword`. The `readLine` method returns a string containing whatever the user keyed in—that's pretty intuitive. However, the `readPassword` method doesn't return a string; it returns a character array. Here's the reason for this: Once you've got the password, you can verify it and then absolutely remove it from memory. If a string was returned, it could exist in a pool somewhere in memory, and perhaps some nefarious hacker could find it.

Let's take a look at a small program that uses a console to support testing another class:

```
import java.io.Console;

public class NewConsole {
  public static void main(String[] args) {
    String name = "";
    Console c = System.console();           // #1: get a Console
    char[] pw;
    pw = c.readPassword("%s", "pw: ");      // #2: return a char[]
    for(char ch: pw)
      c.format("%c ", ch);                  // #3: format output
    c.format("\n");

    MyUtility mu = new MyUtility();
    while(true) {
      name = c.readLine("%s", "input?: ");  // #4: return a String

      c.format("output: %s \n", mu.doStuff(name));
    }
  }
}
class MyUtility {                           // #5: class to test
  String doStuff(String arg1) {
    // stub code
    return "result is " + arg1;
  }
}
```

Let's review this code:

- At line 1, we get a new `Console` object. Remember that we can't say this:

```
Console c = new Console();
```

- At line 2, we invoke `readPassword`, which returns a `char[]`, not a string. You'll notice when you test this code that the password you enter isn't echoed on the screen.
- At line 3, we're just manually displaying the password you keyed in, separating each character with a space. Later on in this chapter, you'll read about the `format()` method, so stay tuned.
- At line 4, we invoke `readLine`, which returns a string.
- At line 5 is the class that we want to test. We recommend that you use something like `NewConsole` to test the concepts that you're learning.

The `Console` class has more capabilities than are covered here, but if you understand everything discussed so far, you'll be in good shape for the exam.

CERTIFICATION OBJECTIVE

Files, Path, and Paths (OCP Objectives 9.1 and 9.2)

9.1 Use Path interface to operate on file and directory paths.
9.2 Use Files class to check, read, delete, copy, move, manage metadata of a file or directory.

The OCP 8 exam has two sections devoted to I/O. The previous section Oracle refers to as "Java I/O Fundamentals" (which we've referred to as the 8.*x* objectives), and it was focused on the `java.io` package. Now we're going to look at the set of objectives Oracle calls "Java File I/O (NIO.2)," whose specific objectives we'll refer to as 9.*x*. The term *NIO.2* is a bit loosely defined, but most people (and the exam creators) define NIO.2 as being the key new features introduced in Java 7 that reside in two packages:

- `java.nio.file`
- `java.nio.file.attribute`

We'll start by looking at the important classes and interfaces in the `java.nio.file` package, and then we'll move to the `java.nio.file.attribute` package later in the chapter.

As you read earlier in the chapter, the `File` class represents a file or directory at a high level. NIO.2 adds three new central classes that you'll need to understand well for the

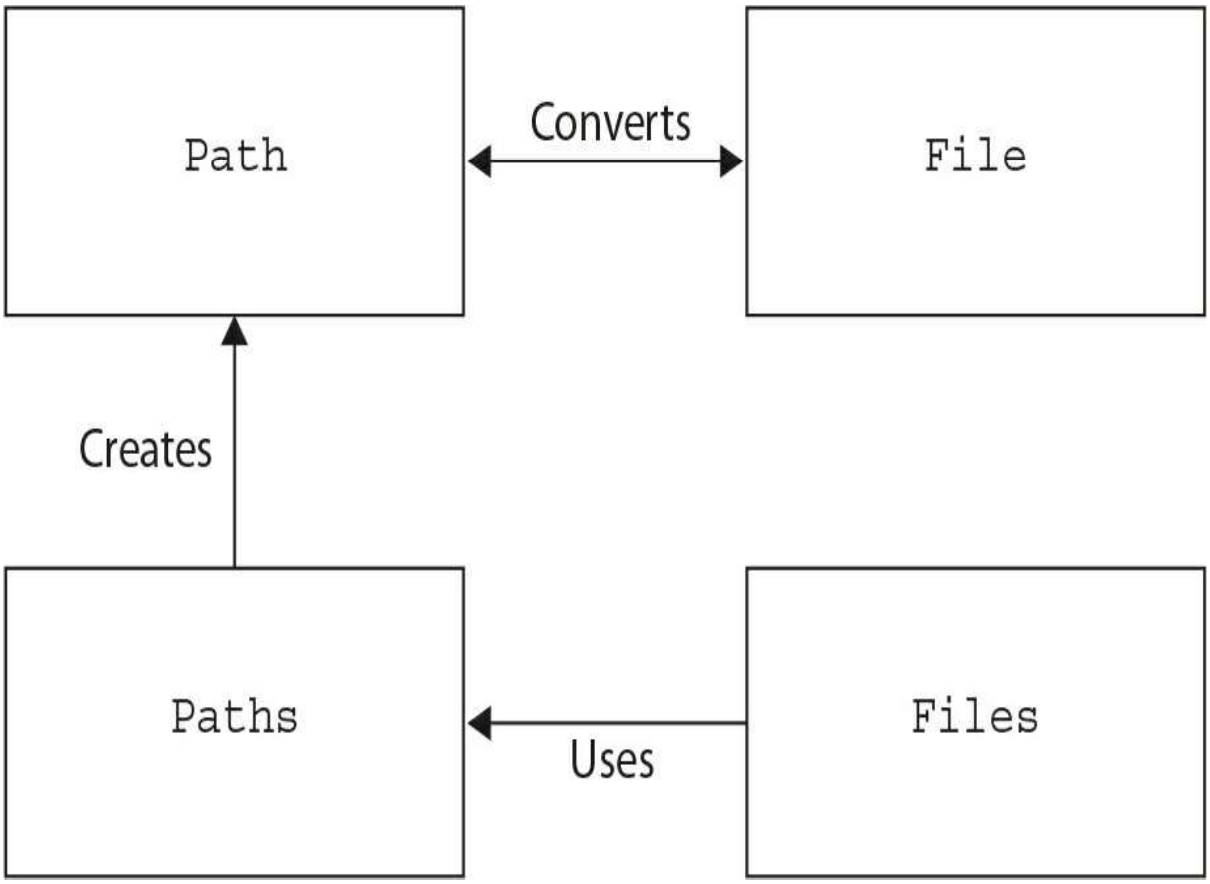
exam:

- Path This interface replaces `File` as the representation of a file or a directory when working in NIO.2. It is a lot more powerful than a `File`.
- Paths This class contains static methods that create `Path` objects.
- Files This class contains static methods that work with `Path` objects. You'll find basic operations in here like copying or deleting files.

The interface `java.nio.file.Path` is one of the key classes of file-based I/O under NIO.2. Just like the good old `java.io.File`, a `Path` represents only a location in the file system, like `C:\java\workspace\ocpjp7` (a Windows directory) or `/home/nblack/docs` (the `docs` directory of user `nblack` on UNIX). When you create a `Path` to a new file, that file does not exist until you actually create the file using `Files.createFile(Path target)`. The `Files` utility class will be covered in depth in the next section.

The difference between File, Files, Path, and Paths is really important. Read carefully on the exam. A one-letter difference can mean a big difference in what the class does.

Let's take a look at these relationships another way. The `Paths` class is used to create a class implementing the `Path` interface. The `Files` class uses `Path` objects as parameters. All three of these were introduced in Java 7. Then there is the `File` class. It's been around since the beginning. `File` and `Path` objects know how to convert to the other. This lets any older code interact with the new APIs in `Files`. But notice what is missing. In the figure, there is no line between `File` and `Files`. Despite the similarity in name, these two classes do not know about each other.



To make sure you know the difference between these key classes backward and forward, make sure you can fill in the four rightmost columns in Table 5-2.

TABLE 5-2 Comparing the Core Classes

	File	Files	Path	Paths
Existed in Java 6?	Yes	No	No	No
Concrete class or interface?	Concrete class	Concrete class	Interface	Concrete class
Create using "new"	Yes	No	No	No
Contains only static methods	No	Yes	No	Yes

Creating a Path

A `Path` object can be easily created by using the `get` methods from the `Paths` helper class. Remember you are calling `Paths.get()` and not `Path.get()`. If you don't remember why, study the last section some more. It's important to have this down cold.

Taking a look at two simple examples, we have:

```
Path p1 = Paths.get("/tmp/file1.txt");   // on UNIX
Path p2 = Paths.get("c:\\temp\\test");   // On Windows
```

The actual method we just called is `Paths.get(String first, String... more)`. This means we can write it out by separating the parts of the path.

```
Path p3 = Paths.get("/tmp", "file1.txt");   // same as p1
Path p4 = Paths.get("c:", "temp", "test");  // same as p2
Path p5 = Paths.get("c:\\temp", "test") ;   // also same as p2
```

As you can see, you can separate out folder and filenames as much or as little as you want when calling `Paths.get()`. For Windows, that is particularly cool because you can make the code easier to read by getting rid of the backslash and escape character.

Be careful when creating paths. The previous examples are absolute paths since they begin with the root (/ on UNIX or c: on Windows). When you don't begin with the root, the `Path` is considered a relative path, which means Java looks from the current directory. Which `file1.txt` do you think `p6` has in mind?

```
Path p6 = Paths.get("tmp", "file1.txt");  // relative path - NOT same as p1
/ (root)
   |-- tmp
         | - file1.txt
         | - tmp
               | - file1.txt
```

It depends. If the program is run from the root, it is the one in `/tmp/file1.txt`. If the program is run from `/tmp`, it is the one in `/tmp/tmp/file1.txt`. If the program is run from anywhere else, `p6` refers to a file that does not exist.

One more thing to watch for. If you are on Windows, you might deal with a URL that looks like `file:///c:/temp`. The `file://` is a protocol just like `http://` is. This syntax allows you to browse to a folder in Internet Explorer. Your program might have to

deal with such a `String` that a user copied/pasted from the browser. No problem, right? We learned to code:

```
Path p = Paths.get("file:///c:/temp/test");
```

Unfortunately, this doesn't work, and you get an exception about the colon being invalid that looks something like this:

```
Exception in thread "main" java.nio.file.InvalidPathException:
Illegal char <:>
at index 4: file:///c:/temp
```

`Paths` provides another method that solves this problem. `Paths.get(URI uri)` lets you (indirectly) convert the `String` to a `URI` (Uniform Resource Identifier) before trying to create a `Path`:

```
Path p = Paths.get(URI.create("file:///C:/temp"));
```

The last thing you should know is that the `Paths.get()` method we've been discussing is really a shortcut. You won't need to code the longer version, but it is good to understand what is going on under the hood. First, Java finds out what the default file system is. For example, it might be `WindowsFileSystemProvider`. Then Java gets the path using custom logic for that file system. Luckily, this all goes on without us having to write any special code or even think about it.

```
Path short = Paths.get("c:", "temp");
Path longer = FileSystems.getDefault()      // get default file system
      .getPath("c:", "temp");               // then get the Path
```

Now that you know how to create a `Path` instance, you can manipulate it in various ways. We'll get back to that in a bit.

As far as the exam is concerned, *Paths.get()* is how to create a *Path* initially. There is another way that is useful when working with code that was written before Java 7:

```
Path  convertedPath = file.toPath();
File  convertedFile = path.toFile();
```

If you are updating older code that uses *File*, you can convert it to a *Path* and start calling the new classes. And if your newer code needs to call older code, it can convert back to a *File*.

Creating Files and Directories

With I/O, we saw that a file doesn't exist just because you have a `File` object. You have to call `createNewFile()` to bring the file into existence and `exists()` to check if it exists. Rewriting the example from earlier in the chapter to use NIO.2 methods, we now have:

```
Path path = Paths.get("fileWrite1.txt");   // it's only an object
System.out.println(Files.exists(path));    // look for a real file
Files.createFile(path);                    // create a file!
System.out.println(Files.exists(path));    // look again
```

NIO.2 has equivalent methods with two differences:

- You call static methods on `Files` rather than instance methods on `File`.
- Method names are slightly different.

See Table 5-3 for the mapping between old class/method names and new ones. You can still continue to use the older I/O approach if you happen to be dealing with `File` objects.

TABLE 5-3 I/O vs. NIO.2

Description	I/O Approach	NIO.2 Approach
Create an empty file	`File file = new File("test");` `file.createNewFile();`	`Path path = Paths.get("test");` `Files.createFile(path);`
Create an empty directory	`File file = new File("dir");` `file.mkdir()`	`Path path = Paths.get("dir");` `Files.createDirectory(path);`
Create a directory, including any missing parent directories	`File file = new File("/a/b/c");` `file.mkdirs();`	`Path path = Paths.get("/a/b/c");` `Files.createDirectories(path);`
Check if a file or directory exists	`File file = new File("test");` `file.exists();`	`Path path = Paths.get("test");` `Files.exists(path);`

The method *Files.notExists()* supplements *Files.exists()*. In some incredibly rare situations, Java won't have enough permissions to know whether the file exists. When this happens, both methods return false.

You can also create directories in Java. Suppose we have a directory named `/java` and we want to create the file `/java/source/directory/Program.java`. We could do this one at a time:

```
Path path1 = Paths.get("/java/source");
Path path2 = Paths.get("/java/source/directory");
Path file = Paths.get("/java/source/directory/Program.java");
Files.createDirectory(path1);       // create first level of directory
Files.createDirectory(path2);       // create second level of directory
Files.createFile(file);             // create file
```

Or we could create all the directories in one go:

```
Files.createDirectories(path2);    // create all levels of directories
Files.createFile(file);            // create file
```

Although both work, the second is clearly better if you have a lot of directories to create. And remember that the directory needs to exist by the time the file is created.

Copying, Moving, and Deleting Files

We often copy, move, or delete files when working with the file system. Up until Java 7, this was hard to do. Now, however, each is one line. Let's look at some examples:

```
Path source = Paths.get("/temp/test1.txt");  // exists
Path target = Paths.get("/temp/test2.txt");  // doesn't yet exist
Files.copy(source, target);                  // now two copies of the file
Files.delete(target);                        // back to one copy
Files.move(source, target);                  // still one copy
```

This is all pretty self-explanatory. We copy a file, delete the copy, and then move the file. Now, let's try another example:

```
Path one = Paths.get("/temp/test1.txt");    // exists
Path two = Paths.get("/temp/test2.txt");    // exists
Path targ = Paths.get("/temp/test23.txt"); // doesn't yet exist
Files.copy(one, targ);                      // now two copies of the file
Files.copy(two, targ);                      // oops,
                                            // FileAlreadyExistsException
```

Java sees it is about to overwrite a file that already exists. Java doesn't want us to lose the file, so it "asks" if we are sure by throwing an exception. `copy()` and `move()` actually take an optional third parameter—zero or more `CopyOptions`. The most useful option you can pass is `StandardCopyOption.REPLACE_EXISTING`.

```
Files.copy(two,  target,                    // ok. You know what
  StandardCopyOption.REPLACE_EXISTING);     // you are doing
```

We have to think about whether a file exists when deleting the file too. Let's say we wrote this test code:

```
Path path = Paths.get("/java/out.txt");
try {
   methodUnderTest();          // might throw an exception
   Files.createFile(path);     // file only gets created
                               // if methodUnderTest() succeeds
} finally {
  Files.delete(path);          // NoSuchFileException if no file
}
```

We don't know whether `methodUnderTest` works properly yet. If it does, the code works fine. If it throws an exception, we never create the file and `Files.delete()` throws a `NoSuchFileException`. This is a problem, as we only want to delete the file if it was created so we aren't leaving stray files around. There is an alternative. `Files.deleteIfExists(path)` returns true and deletes the file only if it exists. If not, it just quietly returns false. Most of the time, you can ignore this return value. You just want the file to not be there. If it never existed, mission accomplished.

If you have to work on pre-Java 7 code, you can use the *FileUtils* class in Apache Commons IO (http://commons.apache.org/io). It has methods similar to many of the copy, move, and delete methods that are now built into Java.

To review, Table 5-4 lists the methods on `Files` that you are likely to come across on the exam. Luckily, the exam doesn't expect you to know all 30 methods in the API. The important thing to remember is to check the `Files` JavaDoc when you find yourself dealing with files.

TABLE 5-4 `Files` Methods

Method	Description
`Path copy(Path source, Path target, CopyOption... options)`	Copy the file from source to target and return target
`Path move(Path source, Path target, CopyOption... options)`	Move the file from source to target and return target
`void delete(Path path)`	Delete the file and throw an exception if it does not exist
`boolean deleteIfExists(Path path)`	Delete the file if it exists and return whether file was deleted
`boolean exists(Path path, LinkOption... options)`	Return true if file exists
`boolean notExists(Path path, LinkOption... options)`	Return true if file does not exist

Retrieving Information about a Path

The `Path` interface defines a bunch of methods that return useful information about the path that you're dealing with. In the following code listing, a `Path` is created referring to a directory and then we output information about the `Path` instance:

```
Path path = Paths.get("C:/home/java/workspace");
System.out.println("getFileName: "  + path.getFileName());
System.out.println("getName(1): " +  path.getName(1));
System.out.println("getNameCount: "  + path.getNameCount());
System.out.println("getParent: " +  path.getParent());
System.out.println("getRoot: " +  path.getRoot());
System.out.println("subpath(0, 2): " + path.subpath(0, 2));
System.out.println("toString: " +  path.toString());
```

When you execute this code snippet on Windows, the following output is printed:

```
getFileName: workspace
getName(1): java
getNameCount: 3
getParent: C:\home\java
getRoot: C:\
subpath(0, 2): home\java
toString: C:\home\java\workspace
```

Based on this output, it is fairly simple to describe what each method does. Table 5-5 does just that.

TABLE 5-5 `Path` Methods

Method	Description
`String getFileName()`	Returns the filename or the last element of the sequence of name elements.
`Path getName(int index)`	Returns the path element corresponding to the specified index. The 0th element is the one closest to the root. (On Windows, the root is usually C:\ and on UNIX, the root is /.)
`int getNameCount()`	Returns the number of elements in this path, excluding the root.
`Path getParent()`	Returns the parent path, or `null` if this path does not have a parent.
`Path getRoot()`	Returns the root of this path, or `null` if this path does not have a root.
`Path subpath(int beginIndex, int endIndex)`	Returns a subsequence of this path (not including a root element) as specified by the beginning (included) and ending (not included) indexes.
`String toString()`	Returns the string representation of this path.

Here is yet another interesting fact about the `Path` interface: It extends from `Iterable<Path>`. At first sight, this seems anything but interesting. But every class that (correctly) implements the `Iterable<?>` interface can be used as an expression in the enhanced `for` loop. You know you can iterate through an array or a `List`, but you can iterate through a `Path` as well. That's pretty cool!

Using this functionality, it's easy to print the hierarchical tree structure of a file (or

directory), as the following example shows:

```
int spaces = 1;
Path myPath = Paths.get("tmp", "dir1", "dir2", "dir3", "file.txt");
for (Path subPath : myPath) {
  System.out.format("%" + spaces + "s%s%n", "", subPath);
  spaces += 2; }
```

When you run this example, a (simplistic) tree is printed. Thanks to the variable `spaces` (which is increased with each iteration by 2), the different subpaths are printed like a directory tree.

```
tmp
   dir1
      dir2
         dir3
            file.txt
```

Normalizing a Path

Normally (no pun intended), when you create a `Path`, you create it in a direct way. However, all three of these return the same logical `Path`:

```
Path p1 = Paths.get("myDirectory");
Path p2 = Paths.get("./myDirectory");          // one dot means
                                               // current directory
Path p3 = Paths.get("anotherDirectory", "..",  // two dots means go up
                    "myDirectory");            // one directory
```

`p1` is probably what you would type if you were coding. `p2` is just plain redundant. `p3` is more interesting. The two directories—`anotherDirectory` and `myDirectory`—are on the same level, but we have to go up one level to get there:

```
/ (root)
    |-- anotherDirectory
    |-- myDirectory
```

You might be wondering why on earth we wouldn't just type `myDirectory` in the first place. And you would if you could. Sometimes, that doesn't work out. Let's look at a real example of why this might be.

```
/ (root)
    |-- Build_Project
        |-- scripts
            |-- buildScript.sh
    |-- My_Project
        |-- source
            |-- MyClass.java
```

If you wanted to compile `MyClass`, you would `cd` to `/My_Project/source` and run `javac MyClass.java`. Once your program gets bigger, it could be thousands of classes and have hundreds of jar files. You don't want to type in all of those just to compile, so someone writes a script to build your program. `buildScript.sh` now finds everything that is needed to compile and runs the `javac` command for you. The problem is that the current directory is now `/Build_Project/scripts`, not `/My_Project/source`. The build script helpfully builds a path for you by doing something like this:

```
String buildProject                         // build scripts to express
    = "/Build_Project/scripts";             // paths in relation to themselves

String upTwoDirectories = "../..";          // remember what .. means?

String myProject = "/My_Project/source";
Path path = Paths.get(buildProject,
           upTwoDirectories, myProject);  // build path from variables
System.out.println("Original: " + path);
System.out.println("Normalized: " + path.normalize());
```

which outputs:

```
Original:/Build_Project/scripts/../../My_Project/source
Normalized:/My_Project/source
```

Whew. The second one is much easier to read. The `normalize()` method knows that a single dot can be ignored. It also knows that any directory followed by two dots can be removed from a path.

Be careful when using this `normalize()`! It just looks at the `String` equivalent of the path and doesn't check the file system to see whether the directories or files actually exist.

Let's practice and see what `normalize` returns for these paths. This time, we aren't providing a directory structure to show that the directories and files don't need to be present on the computer. What do you think the following prints out?

```
System.out.println(Paths.get("/a/./b/./c").normalize());
System.out.println(Paths.get(".classpath").normalize());
System.out.println(Paths.get("/a/b/c/..").normalize());
System.out.println(Paths.get("../a/b/c").normalize());
```

The output is

```
/a/b/c
.classpath
/a/b
../a/b/c
```

The first one removes all the single dots since they just point to the current directory. The second doesn't change anything since the dot is part of a filename and not a directory. The third sees one set of double dots, so it only goes up one directory. The last one is a little tricky. The two dots do say to go up one directory. But since there isn't a directory before it, `Path` can't simplify it.

To review, `normalize()` removes unneeded parts of the `Path`, making it more like you'd normally type it. (That's not where the word "normalize" comes from, but it is a nice way to remember it.)

Resolving a Path

So far, you have an overview of all methods that can be invoked on a single `Path` object, but what if you need to combine two paths? You might want to do this if you have one `Path` representing your home directory and another containing the `Path` within that directory.

```
Path dir = Paths.get("/home/java");
Path file = Paths.get("models/Model.pdf");
Path result = dir.resolve(file);
System.out.println("result = " + result);
```

This produces the absolute path by merging the two paths:

```
result = /home/java/models/Model.pdf
```

`path1.resolve(path2)` should be read as "resolve `path2` within `path1`'s directory." In this example, we resolved the path of the `file` within the directory provided by `dir`.

Keeping this definition in mind, let's look at some more complex examples:

```
Path absolute = Paths.get("/home/java");
Path relative = Paths.get("dir");
Path file = Paths.get("Model.pdf");
System.out.println("1: " + absolute.resolve(relative));
System.out.println("2: " + absolute.resolve(file));
System.out.println("3: " + relative.resolve(file));
System.out.println("4: " + relative.resolve(absolute));  // BAD
System.out.println("5: " + file.resolve(absolute));      // BAD
System.out.println("6: " + file.resolve(relative));      // BAD
```

The output is

```
1:  /home/java/dir
2:  /home/java/Model.pdf
3:  dir/Model.pdf
4:  /home/java
5:  /home/java
6:  Model.pdf/dir
```

The first three do what you'd expect. They add the parameter to resolve to the provided path object. The fourth and fifth ones try to resolve an absolute path within the context of something else. The problem is that an absolute path doesn't depend on other directories. It is absolute. Therefore, `resolve()` just returns that absolute path. The output of the sixth one looks a little bit weird, but Java does the only right thing to do here. For all it knows the `Path` referred to by `Model.pdf` may be a directory and the `Path` referred to by `dir` may be a file!

Just like `normalize()`, keep in mind that `resolve()` will not check that the directory or file actually exists. To review, `resolve()` tells you how to resolve one path within another.

Careful with methods that come in two flavors: one with a **`Path`** *parameter and the*

other with a **`String`** *parameter such as* **`resolve()`**. *The tricky part here is that* **`null`** *is a valid value for both a* **`Path`** *and a* **`String`**. *What will happen if you pass just* **`null`** *as a parameter? Which method will be invoked?*

```
Path path = Paths.get("/usr/bin/zip");
path.resolve(null);
```

The compiler can't decide which method to invoke: the one with the **`Path`** *parameter or the other one with the* **`String`** *parameter. That's why this code won't compile, and if you see such code in an exam question, you'll know what to do.*

The following examples will compile without any problem, because the compiler knows which method to invoke, thanks to the type of the variable **`other`** *and the explicit cast to* **`String`**.

```
Path path = Paths.get("/usr/bin/zip");
Path other = null;
path.resolve(other);
path.resolve ((String) null);
```

Relativizing a Path

Now suppose we want to do the opposite of resolve. We have the absolute path of our home directory and the absolute path of the music file in our home directory. We want to know just the music file directory and name.

```
Path dir = Paths.get("/home/java");
Path music = Paths.get("/home/java/country/Swift.mp3");
Path mp3 = dir.relativize(music);
System.out.println(mp3);
```

The output is

```
country/Swift.mp3.
```

Java recognized that the `/home/java` part is the same and returned a path of just the remainder.

`path1.relativize(path2)` should be read as "give me a path that shows how to get from `path1` to `path2`." In this example, we determined that `music` is a file in a directory

named country within `dir`.

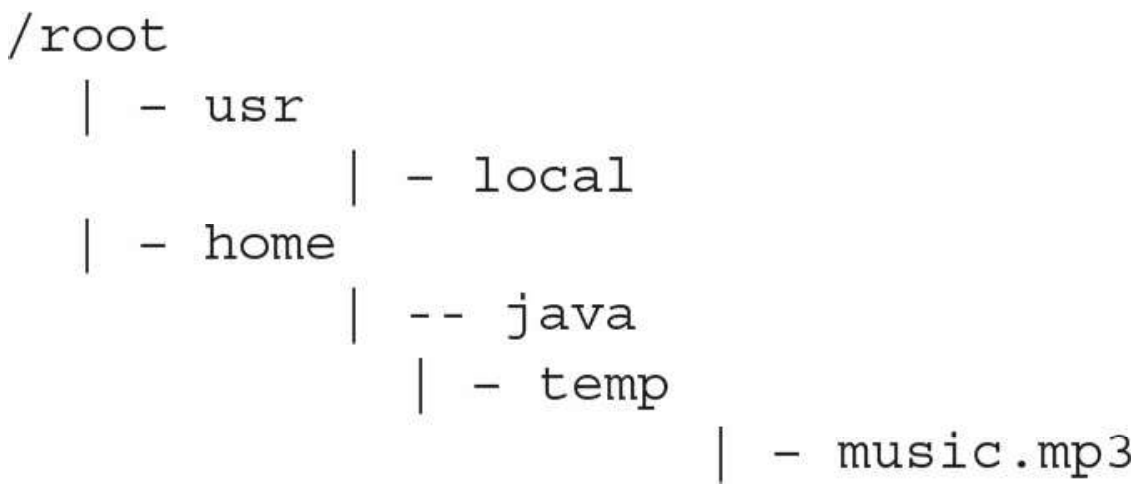
Keeping this definition in mind, let's look at some more complex examples:

```
Path absolute1 = Paths.get("/home/java");
Path absolute2 = Paths.get("/usr/local");
Path absolute3 = Paths.get("/home/java/temp/music.mp3");
Path relative1 = Paths.get("temp");
Path relative2 = Paths.get("temp/music.pdf");
System.out.println("1: " + absolute1.relativize(absolute3));
System.out.println("2: " + absolute3.relativize(absolute1));
System.out.println("3: " + absolute1.relativize(absolute2));
System.out.println("4: " + relative1.relativize(relative2));
System.out.println("5: " + absolute1.relativize(relative1));//BAD
```

The output is

```
1: temp/music.mp3
2: ../..
3: ../../usr/local
4: music.pdf
Exception in thread "main" java.lang.IllegalArgumentException: 'other'
is different type of Path
```

Before you scratch your head, let's look at the logical directory structure here. Keep in mind the directory doesn't actually need to exist; this is just to visualize it.



Now we can trace it through. The first example is straightforward. It tells us how to get to `absolute3` from `absolute1` by going down two directories. The second is similar. We get to `absolute1` from `absolute3` by doing the opposite—going up two directories. Remember from `normalize()` that a double dot means to go up a directory.

The third output statement says that we have to go up two directories and then down two directories to get from `absolute1` to `absolute2`. Java knows this because we provided absolute paths. The worst possible case is to have to go all the way up to the root like we did here.

The fourth output statement is okay. Even though they are both relative paths, there is enough in common for Java to tell what the difference in the path is.

The fifth example throws an exception. Java can't figure out how to make a relative path out of one absolute path and one relative path.

Remember, `relativize()` and `resolve()` are opposites. And just like `resolve()`, `relativize()` does not check that the path actually exists. To review, `relativize()` tells you how to get a relative path between two paths.

CERTIFICATION OBJECTIVE

File and Directory Attributes (OCP Objective 9.2)

9.2 Use Files class to check, read, delete, copy, move, manage metadata of a file or directory.

Metadata is data about data. For a file, you can think of the stuff that's in the file as the data, and the attributes of the file, like the date the file was created, as the metadata; that is, the data about the data that's in the file. When you see "metadata" in this

objective, think "attributes" of files and directories.

Reading and Writing Attributes the Easy Way

In this section, we'll add classes and interfaces from the `java.nio.file.attribute` package to the discussion. Prior to NIO.2, you could read and write just a handful of attributes. Just like we saw when creating files, there is a new way to do this using `Files` instead of `File`. Oracle also took the opportunity to clean up the method signatures a bit. The following example creates a file, changes the last modified date, prints it out, and deletes the file using both the old and new method names. We might do this if we want to make a file look as if it were created in the past. (As you can see, there is a lesson about not relying on file timestamps here!)

```
ZonedDateTime janFirstDateTime =
    ZonedDateTime.of(                                  // create a date
        LocalDate.of(2017, 1, 1),
        LocalTime.of(10, 0), ZoneId.of("US/Pacific"));
Instant januaryFirst = janFirstDateTime.toInstant();

// old way
File file = new File("c:/temp/file");
file.createNewFile();                                  // create the file
file.setLastModified(
    januaryFirst.getEpochSecond()*1000);               // set time
System.out.println(file.lastModified());               // get time
file.delete();                                         // delete the file

// new way
Path path = Paths.get("c:/temp/file2");
Files.createFile(path);                                // create another file
FileTime fileTime =                                    // convert to the new
    FileTime.fromMillis(                               // FileTime object
        januaryFirst.getEpochSecond()*1000);
Files.setLastModifiedTime(path, fileTime);             // set time
System.out.println(Files.getLastModifiedTime(path)); // get time
Files.delete(path);
```

As you can see from the output, the only change in functionality is that the new `Files.getLastModifiedTime()` uses a human-friendly date format.

```
1483293600000
2017-01-01T18:00:00Z
```

The other common type of attribute you can set are file permissions. Both Windows and UNIX have the concept of three types of permissions. Here's what they mean:

- Read You can open the file or list what is in that directory.
- Write You can make a change to the file or add a file to that directory.
- Execute You can run the file if it is a runnable program or go into that directory.

Printing out the file permissions is easy. Note that these permissions are just for the user who is running the program—you! There are other types of permissions as well, but these can't be set in one line.

```
System.out.println(Files.isExecutable(path));
System.out.println(Files.isReadable(path));
System.out.println(Files.isWritable(path));
```

Table 5-6 shows how to get and set these attributes that can be set in one line, both using the older I/O way and the new `Files` class. You may have noticed that setting file permissions isn't in the table. That's more code, so we will talk about it later.

TABLE 5-6 I/O vs. NIO.2 Permissions

Description	I/O Approach	NIO.2 Approach
Get the last modified date/time	`File file = new File("test");` `file.lastModified();`	`Path path = Paths.get("test");` `Files.getLastModifiedTime(path);`
Is read permission set	`File file = new File("test");` `file.canRead();`	`Path path = Paths.get("test");` `Files.isReadable(path);`
Is write permission set	`File file = new File("test");` `file.canWrite();`	`Path path = Paths.get("test");` `Files.isWritable(path);`
Is executable permission set	`File file = new File("test");` `file.canExecute();`	`Path path = Paths.get("test");` `Files.isExecutable(path);`

Set the last modified date/time (Note: `timeInMillis` is an appropriate `long`.)	`File file = new File("test");`
	`file.setLastModified(timeInMillis);`
		`Path path = Paths.get("test");`
		`FileTime fileTime = FileTime.`
		`fromMillis(timeInMillis);`
		`Files.setLastModifiedTime(path,`
		`fileTime);`

Types of Attribute Interfaces

The attributes you set by calling methods on `Files` are the most straightforward ones. Beyond that, Java NIO.2 added attribute interfaces so you could read attributes that might not be on every operating system.

- BasicFileAttributes In the JavaDoc, Oracle says these are "attributes common to many file systems." What they mean is that you can rely on these attributes being

available to you unless you are writing Java code for some funky new operating system. Basic attributes include things like creation date.

- PosixFileAttributes POSIX stands for Portable Operating System Interface. This interface is implemented by both UNIX- and Linux-based operating systems. You can remember this because POSIX ends in "x," as do UNIX and Linux.
- DosFileAttributes DOS stands for Disk Operating System. It is part of all Windows operating systems. Even Windows 8 and 10 have a DOS prompt available.

There are also separate interfaces for setting or updating attributes. While the details aren't in scope for the exam, you should be familiar with the purpose of each one.

- BasicFileAttributeView Used to set the last updated, last accessed, and creation dates.
- PosixFileAttributeView Used to set the groups or permissions on UNIX/Linux systems. There is an easier way to set these permissions though, so you won't be using the attribute view.
- DosFileAttributeView Used to set file permissions on DOS/Windows systems. Again, there is an easier way to set these, so you won't be using the attribute view.
- FileOwnerAttributeView Used to set the primary owner of a file or directory.
- AclFileAttributeView Sets more advanced permissions on a file or directory.

Working with BasicFileAttributes

The `BasicFileAttributes` interface provides methods to get information about a file or directory.

```
BasicFileAttributes basic =  Files.readAttributes(path,  // assume a valid path
     BasicFileAttributes.class);
System.out.println("create: " + basic.creationTime());
System.out.println("access: " + basic.lastAccessTime());
System.out.println("modify: " + basic.lastModifiedTime());
System.out.println("directory: " + basic.isDirectory());
```

The sample output shows that all three date/time values can be different. A file is created once. It can be modified many times. And it can be last accessed for reading after that. The `isDirectory` method is the same as `Files.isDirectory(path)`. It is just an alternative way of getting the same information.

```
create: 2017-03-21T23:14:36Z
access: 2017-09-25T02:01:11Z
modify: 2017-04-12T17:38:51Z
directory: false
```

There are some more attributes on `BasicFileAttributes`, but they aren't on the exam and you aren't likely to need them when coding. Just remember to check the JavaDoc if you need more information about a file.

So far, you've noticed that all the attributes are read only. That's because Java provides a different interface for updating attributes. Let's write code to update the last accessed time:

```
BasicFileAttributes basic = Files.readAttributes(
      path, BasicFileAttributes.class);                 // attributes
FileTime lastUpdated = basic.lastModifiedTime();        // get current
FileTime created = basic.creationTime();                // values
FileTime now = FileTime.fromMillis(System.currentTimeMillis());
BasicFileAttributeView basicView = Files.getFileAttributeView(
      path, BasicFileAttributeView.class);              // "view" this time
basicView.setTimes(lastUpdated, now, created);          // set all three
```

In this example, we demonstrated getting all three times. In practice, when calling `setTimes()`, you should pass null values for any of the times you don't want to change, and only pass `FileTimes` for the times you want to change.

The key takeaways here are that the "`XxxFileAttributes`" classes are read only and the "`XxxFileAttributeView`" classes allow updates.

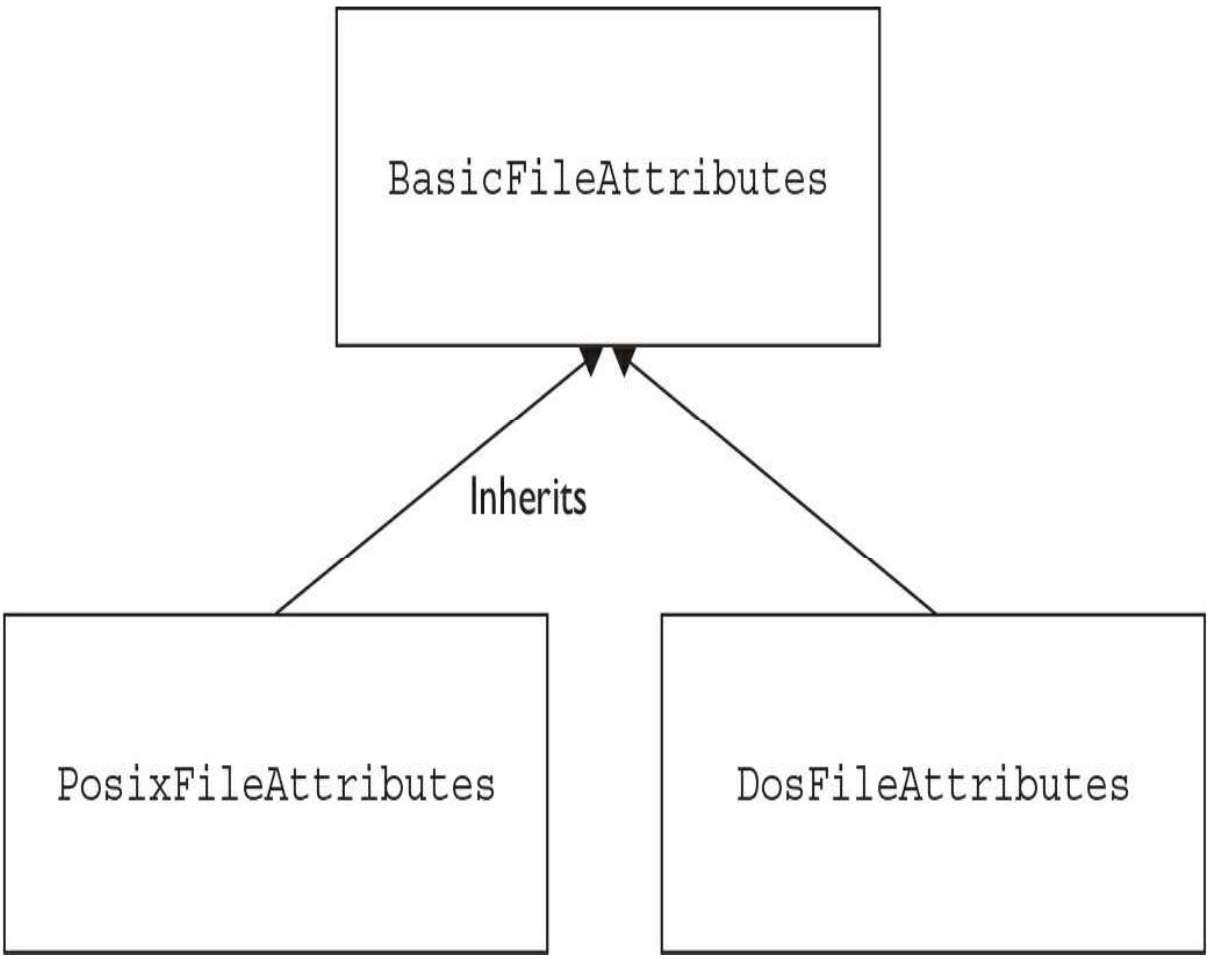
The ***BasicFileAttributes and BasicFileAttributeView*** *interfaces are a bit confusing. They have similar names but different functionality, and you get them in different ways. Try to remember these three things:*

- ***BasicFileAttributeView*** *is singular, but* ***BasicFileAttributes*** *is not.*
- *You get* ***BasicFileAttributeView*** *using* ***Files.getFileAttributeView****, and you get* ***BasicFile Attributes*** *using* ***Files.read***

Attributes.

- You can ONLY update attributes in *BasicFileAttributeView*, not in *BasicFileAttributes*. Remember that the view is for updating.

`PosixFileAttributes` and `DosFileAttributes` inherit from `BasicFileAttributes`. This means you can call Basic methods on a POSIX or DOS subinterface.



Try to use the more general type if you can. For example, if you are only going to use basic attributes, just get `BasicFileAttributes`. This lets your code remain operating system independent. If you are using a mix of basic and POSIX attributes, you can use `PosixFileAttributes` directly rather than calling `readAttributes()` twice to get two different ones.

Working with DosFileAttributes

`DosFileAttributes` adds four more attributes to the basics. We'll look at the most common ones here—hidden files and read-only files. Hidden files typically begin with a dot and don't show up when you type dir to list the contents of a directory. Read-only files are what they sound like—files that can't be updated. (The other two attributes are

"archive" and "system," which you are quite unlikely to ever use.)

```
Path path= Paths.get("C:/test");
Files.createFile(path);                          // create file
Files.setAttribute(path, "dos:hidden", true);    // set attribute
Files.setAttribute(path, "dos:readonly", true);  // another one
DosFileAttributes dos = Files.readAttributes(path,
     DosFileAttributes.class);                   // dos attributes
System.out.println(dos.isHidden());
System.out.println(dos.isReadOnly());
Files.setAttribute(path, "dos:hidden", false);
Files.setAttribute(path, "dos:readonly", false);
dos = Files.readAttributes(path,
     DosFileAttributes.class);                   // get attributes again
System.out.println(dos.isHidden());
System.out.println(dos.isReadOnly());
Files.delete(path);
```

The output is

```
true
true
false
false
```

The first tricky thing in this code is that the `String` "`readonly`" is lowercase even though the method name is mixed case. If you forget and use the `String` "`readOnly`," an `IllegalArgumentException` will be thrown at runtime.

The other tricky thing is that you cannot delete a read-only file. That's why the code calls `setAttribute` a second time with `false` as a parameter, to make it no longer "read only" so the code can clean up after itself. And you can see that we had to call `readAttributes` again to see those updated values.

There is an alternative way to set these attributes so you don't have to worry about the *String* values. However, the exam wants you to know how to use *Files*. It is good to know both ways, though.

```
DosFileAttributeView view = Files.getFileAttributeView(path,
                                DosFileAttributeView.class);
view.setHidden(true);
view.setReadOnly(true);
```

Working with PosixFileAttributes

`PosixFileAttributes` adds two more attributes to the basics—groups and permissions. On UNIX, every file or directory has both an owner and group name.

UNIX permissions are also more elaborate than the basic ones. Each file or directory has nine permissions set in a `String`. A sample is "rwxrw-r--." Breaking this into groups of three, we have "rwx", "rw-," and "r--." These sets of permissions correspond to who gets them. In this example, the "user" (owner) of the file has read, write, and execute permissions. The "group" only has read and write permissions. UNIX calls everyone who is not the owner or in the group "other." "Other" only has read access in this example.

Now let's look at some code to set the permissions and output them in human-readable form:

```
Path path = Paths.get("/tmp/file2");
Files.createFile(path);
PosixFileAttributes posix = Files.readAttributes(path,
PosixFileAttributes.class);                          // get the Posix type
Set<PosixFilePermission> perms =
      PosixFilePermissions.fromString("rw-r--r--");  // UNIX style
Files.setPosixFilePermissions(path, perms);          // set permissions
System.out.println(posix.permissions());             // get permissions
```

The output looks like this:

```
[OWNER_WRITE, GROUP_READ, OTHERS_READ, OWNER_READ]
```

It's not symmetric. We gave Java the permissions in cryptic UNIX format and got them back in plain English. You can also output the group name:

```
System.out.println(posix.group());          // get group
```

which outputs something like this:

```
horse
```

Reviewing Attributes

Let's review the most common attributes information in Table 5-7.

TABLE 5-7 Common Attributes

Type	Read and Write an Attribute

Basic	`// read`
	
	`BasicFileAttributes basic = Files.readAttributes(path,`
	`  BasicFileAttributes.class);`
	`FileTime lastUpdated = basic.lastModifiedTime();`
	`FileTime created = basic.creationTime();`
	`FileTime now = FileTime.fromMillis(System.currentTimeMillis());`
	
	`// write`
	`BasicFileAttributeView basicView =`
	`Files.getFileAttributeView(path,`
	`BasicFileAttributeView.class);`
	`basicView.setTimes(lastUpdated, now, created);`

Posix (UNIX/Linux)	`PosixFileAttributes posix = Files.readAttributes(path,`
	`PosixFileAttributes.class);`
	`Set<PosixFilePermission> perms = PosixFilePermissions.`
	`fromString("rw-r--r--");`
	`Files.setPosixFilePermissions(path, perms);`
	`System.out.println(posix.group());`
	`System.out.println(posix.permissions());`

Dos (Windows)	`DosFileAttributes dos = Files.readAttributes(path,`
	`  DosFileAttributes.class);`
	`System.out.println(dos.isHidden());`
	`System.out.println(dos.isReadOnly());`
	`Files.setAttribute(path, "dos:hidden", false);`
	`Files.setAttribute(path, "dos:readonly", false);`

CERTIFICATION OBJECTIVE

DirectoryStream (OCP Objectives 9.2 and 9.3)

9.2 Use Files class to check, read, delete, copy, move, manage metadata of a file or directory.
9.3 Use Stream API with NIO.2.

Now let's return to more NIO.2 capabilities that you'll find in the `java.nio.file` package… You might need to loop through a directory. Let's say you were asked to list out all the users with a home directory on this computer.

```
/home
  | - users
        | - vafi
        | - eyra

Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream =          // use try-with-resources
     Files.newDirectoryStream(dir)) {        // so we don't have close()
  for (Path path : stream)                   // loop through the stream
     System.out.println(path.getFileName());
}
```

As expected, this outputs

```
vafi
eyra
```

The `DirectoryStream` interface lets you iterate through a directory. But this is just the tip of the iceberg. Let's say we have hundreds of users and each day we want to only report on a few of them. The first day, we only want the home directories of users whose names begin with either the letter *v* or the letter *w*.

```
Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(
  dir, "[vw]*")) {                    // "v" or "w" followed by anything
for (Path path : stream)
      System.out.println(path.getFileName());
}
```

This time, the output is

```
vafi
```

Let's examine the expression `[vw]*`. `[vw]` means either of the characters `v` or `w`. The `*` is a wildcard that means zero or more of any character. Notice this is not a regular expression. (If it were, the syntax would be `[vw].*`—see the dot in there.) `DirectoryStream` uses something called a *glob*. We will see more on globs later in the chapter.

There is one limitation with `DirectoryStream`. It can only look at one directory. One way to remember this is that it works like the `dir` command in DOS or the `ls` command in UNIX. Or you can remember that `DirectoryStream` streams one directory.

FileVisitor

Luckily, there is another class that does, in fact, look at subdirectories. Let's say you want to get rid of all the .class files before zipping up and submitting your assignment. You could go through each directory manually, but that would get tedious really fast. You could write a complicated command in Windows and another in UNIX, but then you'd have two programs that do the same thing. Luckily, you can use Java and only write the code once.

Java provides a `SimpleFileVisitor`. You extend it and override one or more methods. Then you can call `Files.`*`walkFileTree`*, which knows how to recursively look through a directory structure and call methods on a visitor subclass. Let's try our example:

```
/home
  | - src
        | - Test.java
        | - Test.class
        | - dir
              | - AnotherTest.java
              | - AnotherTest.class
```

```
public class RemoveClassFiles
      extends SimpleFileVisitor<Path> {            // need to extend visitor
      public FileVisitResult visitFile(            // called "automatically"
         Path file, BasicFileAttributes attrs)
               throws IOException {
          if ( file.getFileName().toString().endsWith(".class"))
            Files.delete(file);                    // delete the file
          return FileVisitResult.CONTINUE;         // go on to next file
      }
     public static void main(String[] args) throws Exception {
          RemoveClassFiles dirs = new RemoveClassFiles();
          Files.walkFileTree(                      // kick off recursive check
                    Paths.get("/home/src"),        // starting point
                    dirs);                         // the visitor
      }
}
```

This is a simple file visitor. It only implements one method: `visitFile`. This method is called for every file in the directory structure. It checks the extension of the file and deletes it if appropriate. In our case, two .class files are deleted.

There are two parameters to `visitFile()`. The first one is the `Path` object representing the current file. The other is a `BasicFileAttributes` interface. Do you remember what this does? That's right—it lets you find out if the current file is a directory, when it was created, and many other similar pieces of data.

Finally, `visitFile()` returns `FileVisitResult.` *CONTINUE*. This tells `walkFileTree()` that it should keep looking through the directory structure for more files.

Now that we have a feel for the power of this class, let's take a look at all the methods available to us with another example:

```
/home
  | - a.txt
  | - emptyChild
  | - child
        | - b.txt
        | - grandchild
              | - c.txt
```

```
public class PrintDirs extends SimpleFileVisitor<Path> {
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
{
    System.out.println("pre: " + dir);
    return FileVisitResult.CONTINUE; }
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
    System.out.println("file: " + file);
    return FileVisitResult.CONTINUE; }
  public FileVisitResult visitFileFailed(Path file, IOException exc) {
    return FileVisitResult.CONTINUE; }
  public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
    System.out.println("post: " + dir);
    return FileVisitResult.CONTINUE; }
  public static void main(String[] args) throws Exception {
    PrintDirs dirs = new PrintDirs();
    Files.walkFileTree(Paths.get("/home"), dirs); } }
```

You might get the following output:

```
pre: /home
file: /home/a.txt
pre: /home/child
file: /home/child/b.txt
pre: /home/child/grandchild
file: /home/child/grandchild/c.txt
post: /home/child/grandchild
post: /home/child
pre: /home/emptyChild
post: /home/emptyChild
post: /home
```

Note that Java goes down as deep as it can before returning back up the tree. This is called a *depth-first search*. We said "might" because files and directories at the same level can get visited in either order.

You can override as few or as many of the four methods as you'd like. Note that the second half of the methods have IOException as a parameter. This allows those methods to handle problems that came earlier when walking through the tree. Table 5-8 summarizes the methods.

TABLE 5-8 `FileVisitor` Methods

Method	Description	IOException Parameter?
`preVisitDirectory`	Called before drilling down into the directory	No
`visitFile`	Called once for each file (but not for directories)	No
`visitFileFailed`	Called only if there was an error accessing a file, usually a permissions issue	Yes
`postVisitDirectory`	Called when finished with the directory on the way back up	Yes

You actually do have some control, though, through those `FileVisitResult` constants. Suppose we changed the `preVisitDirectory` method to the following:

```
public FileVisitResult preVisitDirectory(
  Path dir, BasicFileAttributes attrs) {
  System.out.println("pre: " + dir);
  String name = dir.getFileName().toString();
  if (name.equals("child"))
    return FileVisitResult.SKIP_SUBTREE;
  return FileVisitResult.CONTINUE;
}
```

Now the output is

```
pre: /home
file: /home/a.txt
pre: /home/child
pre: /home/emptyChild
post: /home/emptyChild
post: /home
```

Since we instructed the program to skip the entire `child` subtree—i.e., we don't see the file: `b.txt`, or the subdirectory: `grandchild`—we also don't see the post visit call.

Now what do you think would happen if we changed `FileVisitResult.SKIP_SUBTREE` to `FileVisitResult.TERMINATE`? The output might be:

```
pre: /home
file: /home/a.txt
pre: /home/child
```

We see that as soon as the "child" directory came up, the program stopped walking the tree. And again, we are using "might" in terms of the output. It's also possible for `emptyChild` to come up first, in which case, the last line of the output would be `/home/emptyChild`.

There's one more result type. What do you think would happen if we changed `FileVisitResult.TERMINATE` to `FileVisitResult.SKIP_SIBLINGS`? The output happens to be the same as the previous example:

```
pre: /home
file: /home/a.txt
pre: /home/child
```

`SKIP_SIBLINGS` is a combination of `SKIP_SUBTREE` and "don't look in any folders at the same level." This means we skip everything under `child` and also skip `emptyChild`.

One more example to make sure you really understand what is going on. What do you think gets output if we use this method?

```
public FileVisitResult preVisitDirectory(Path dir,
  BasicFileAttributes attrs) {
  System.out.println("pre: " + dir);
  String name = dir.getFileName().toString();
  if (name.equals("grandchild"))
    return FileVisitResult.SKIP_SUBTREE;
  if ( name.equals("emptyChild"))
    return FileVisitResult.SKIP_SIBLINGS;
  return FileVisitResult.CONTINUE;
}
```

Assuming `child` is encountered before `emptyChild`, the output is

```
pre: /home
file: /home/a.txt
pre: /home/child
file: /home/child/b.txt
pre: /home/child/grandchild
post: /home/child
pre: /home/emptyChild
post: /home
```

We don't see `file: c.txt` or `post: /home/child/grandchild` because we skip `grandchild` the subtree. We don't see `post: /home/emptyChild` because we skip siblings of `emptyChild`. But wait. Isn't `/home/child` a sibling? It is. But the visitor goes in order. Since `child` was seen before `emptyChild`, it is too late to skip it. Just like when you print a document, it is too late to prevent pages from printing that have already printed. File visitor can only skip subtrees that it has not encountered yet.

PathMatcher

`DirectoryStream` and `FileVisitor` allowed us to go through the files that exist. Things can get complicated fast, though. Imagine you had a requirement to print out the

names of all text files in any subdirectory of "password." You might be wondering why anyone would want to do this. Maybe a teammate foolishly stored passwords for everyone to see and you want to make sure nobody else did that. You could write logic to keep track of the directory structure, but that makes the code harder to read and understand. By the end of this section, you'll know a better way.

Let's start out with a simpler example to see what a `PathMatcher` can do:

```
Path path1 = Paths.get("/home/One.txt");
Path path2 = Paths.get("One.txt");
PathMatcher matcher = FileSystems.getDefault()  // get the PathMatcher
          .getPathMatcher(                      // for the right file system
          "glob:*.txt");                        // wait. What's a glob?
System.out.println(matcher.matches(path1));
System.out.println(matcher.matches(path2));
```

which outputs:

```
false
true
```

We can see that the code checks if a `Path` consists of any characters followed by ".txt." To get a `PathMatcher`, you have to call `FileSystems.getDefault().getPathMatcher` because matching works differently on different operating systems. `PathMatcher`s use a new type that you probably haven't seen before called a glob. Globs are not regular expressions, although they might look similar at first. Let's look at some more examples of globs using a common method so we don't have to keep reading the same "boilerplate" code. (Boilerplate code is the part of the code that is always the same.)

```
public void matches(Path path, String glob) {
  PathMatcher matcher = FileSystems.getDefault().getPathMatcher(glob);
  System.out.println(matcher.matches(path));
}
```

In the world of globs, one asterisk means "match any character except for a directory boundary." Two asterisks means "match any character, including a directory boundary."

```
Path path = Paths.get("/com/java/One.java");
matches(path, "glob:*.java");      // false
matches(path, "glob:**/*.java");   // true
matches(path, "glob:*");           // false
matches(path, "glob:**");          // true
```

Remember that we are using a file system–specific *PathMatcher*. This means slashes and backslashes can be treated differently, depending on what operating system you happen to be running. The previous example does print the same output on both Windows and UNIX because it uses forward slashes.However, if you change just one line of code, the output changes:

```
Path path = Paths.get("com\\java\\One.java");
```

Now Windows still prints:

```
false
true
false
true
```

However, UNIX prints:

```
true
false
true
true
```

Why? Because UNIX doesn't see the backslash as a directory boundary. The lesson here is to use / instead of \\ so your code behaves more predictably across operating systems.

Now let's match files with a four-character extension. A question mark matches any character. A character could be a letter or a number or anything else.

```
Path path1 = Paths.get("One.java");
Path path2 = Paths.get("One.ja^a");
matches(path1, "glob:*.????");      // true
matches(path1, "glob:*.???");       // false
matches(path2, "glob:*.????");      // true
matches(path2, "glob:*.???");       // false
```

Globs also provide a nice way to match multiple patterns. Suppose we want to match anything that begins with the names Kathy or Bert:

```
Path path1 = Paths.get("Bert-book");
Path path2 = Paths.get("Kathy-horse");
matches(path1, "glob:{Bert*,Kathy*}");    // true
matches(path2, "glob:{Bert,Kathy}*");     // true
matches(path1, "glob:{Bert,Kathy}");      // false
```

The first glob shows we can put wildcards inside braces to have multiple glob expressions. The second glob shows that we can put common wildcards outside the braces to share them. The third glob shows that without the wildcard, we will only match the literal strings "Bert" and "Kathy."

You can also use sets of characters like `[a-z]` or `[#$%]` in globs just like in regular expressions. You can also escape special characters with a backslash. Let's put this all together with a tricky example:

```
Path path1 = Paths.get("0*b/test/1");
Path path2 = Paths.get("9\\*b/test/1");
Path path3 = Paths.get("01b/test/1");
Path path4 = Paths.get("0*b/1");
String glob = "glob:[0-9]\\*{A*,b}/**/1";
matches(path1, glob);                    // true
matches(path2, glob);                    // false
matches(path3, glob);                    // false
matches(path4, glob);                    // false
```

Spelling out what the glob does, we have the following:

- [0-9] One single digit. Can also be read as any one character from 0 to 9.
- * The literal character asterisk rather than the asterisk that means to match anything. A single backslash before * escapes it. However, Java won't let you type a single backslash, so you have to escape the backslash itself with another backslash.
- {A*,b} Either a capital *A* followed by anything or the single character *b*.
- /**/ One or more directories with any name.
- 1 The single character 1.

The second path doesn't match because it has the literal backslash followed by the literal asterisk. The glob was looking for the literal asterisk by itself. The third path also doesn't match because there is no literal asterisk. The fourth path doesn't match because there is no directory between "b" and "1" for the ** to match. Luckily, nobody would write such a crazy, meaningless glob. But if you can understand this one, you are all set. Globs tend to be simple expressions like `{*.txt,*.html}` when used for real.

Since globs are just similar enough to regular expressions to be tricky, Table 5-9 reviews the similarities and differences in common expressions. Regular expressions are more powerful, but globs focus on what you are likely to need when matching filenames.

TABLE 5-9 Glob vs. Regular Expression

What to Match	In a Glob	In a Regular Expression
Zero or more of any character, including a directory boundary	`**`	`.*`
Zero or more of any character, not including a directory boundary	`*`	N/A – no special syntax
Exactly one character	`?`	`.`
Any digit	`[0-9]`	`[0-9]`
Begins with cat or dog	`{cat, dog}*`	`(cat|dog).*`

By now, you've probably noticed that we are dealing with `Path` objects, which means they don't actually need to exist on the file system. But we wanted to print out all the text files that actually exist in a subdirectory of password. Luckily, we can combine the power of `PathMatcher`s with what we already know about walking the file tree to accomplish this.

```
public class MyPathMatcher extends SimpleFileVisitor<Path> {
  private PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher(
    "glob:**/password/**.txt");                    // ** means any subdirectory
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
     throws IOException {
    if (matcher.matches(file)) {
      System.out.println(file);
    }
    return FileVisitResult.CONTINUE;
  }
  public static void main(String[] args) throws Exception {
    MyPathMatcher dirs = new MyPathMatcher();
    Files.walkFileTree(Paths.get("/"),   dirs);  // start with root
  }
}
```

The code looks similar, regardless of what you want to do. You just change the glob pattern to what you actually want to match.

WatchService

The last thing you need to know about in NIO.2 is `WatchService`. Suppose you are writing an installer program. You check that the directory you are about to install into is empty. If not, you want to wait until the user manually deletes that directory before continuing. Luckily, you won't have to write this code from scratch, but you should be familiar with the concepts. Here's the directory tree:

```
/dir
   | - directoryToDelete
   | - other
```

Here's the code snippet:

```
Path dir = Paths.get("/dir");                    // get directory containing
                                                 // file/directory we care
                                                 // about
WatchService watcher = FileSystems.getDefault() // file system-specific code
      .newWatchService();                        // create empty WatchService
dir.register(watcher, ENTRY_DELETE);             // needs a static import!
                                                 // start watching for
                                                 // deletions
while (true) {                                   // loop until say to stop
  WatchKey key;
  try {
    key = watcher.take();                        // wait for a deletion
  } catch (InterruptedException x) {
      return;                                    // give up if something goes
                                                 // wrong
  }
  for (WatchEvent<?> event : key.pollEvents()) {
    WatchEvent.Kind<?> kind = event.kind();
    System.out.println(kind.name());             // create/delete/modify
    System.out.println(kind.type());             // always a Path for us
    System.out.println(event.context());         // name of the file
    String name = event.context().toString();
    if (name.equals("directoryToDelete")) {      // only delete right directory
       System.out.format("Directory deleted, now we can proceed");
```

```
            return;                                    // end program, we found what
                                                       // we were waiting for
        }
      }
      key.reset();                                     // keep looking for events
    }
```

Supposing we delete directory "`other`" followed by directory `directoryToDelete`, this outputs:

```
ENTRY_DELETE
interface java.nio.file.Path
other
ENTRY_DELETE
interface java.nio.file.Path
directoryToDelete
Directory deleted, now we can proceed
```

Notice that we had to watch the directory that contains the files or directories we are interested in. This is why we watched `/dir` instead of `/dir/directoryToDelete`. This is also why we had to check the context to make sure the directory we were actually interested in is the one that was deleted.

The basic flow of `WatchService` stays the same, regardless of what you want to do:

1. Create a new `WatchService`.
2. Register it on a `Path` listening to one or more event types.
3. Loop until you are no longer interested in these events.
4. Get a `WatchKey` from the `WatchService`.
5. Call `key.pollEvents` and do something with the events.
6. Call `key.reset` to look for more events.

Let's look at some of these in more detail. You register the `WatchService` on a `Path` using statements like the following:

```
dir1.register(watcher, ENTRY_DELETE);
dir2.register(watcher, ENTRY_DELETE, ENTRY_CREATE);
dir3.register(watcher, ENTRY_DELETE, ENTRY_CREATE, ENTRY_MODIFY);
```

(Note: These `ENTRY_XXX` constants can be found in the `StandardWatchEventKinds` class. Here and in later code, you'll probably want to create static imports for these constants.) You can register one, two, or three of the event types. `ENTRY_DELETE` means you want your program to be informed when a file or directory has been deleted. Similarly, `ENTRY_CREATE` means a new file or directory has been created. `ENTRY_MODIFY` means a file has been edited in the directory. These changes can be made manually by a human or by another program on the computer.

Renaming a file or directory is interesting, as it does not show up as `ENTRY_MODIFY`. From Java's point of view, a rename is equivalent to creating a new file and deleting the original. This means that two events will trigger for a rename—both `ENTRY_CREATE` and `ENTRY_DELETE`. Actually editing a file will show up as `ENTRY_MODIFY`.

To loop through the events, we use `while(true)`. It might seem a little odd to write a loop that never ends. Normally, there is a `break` or `return` statement in the loop so you stop looping once whatever event you were waiting for has occurred. It's also possible you want the program to run until you kill or terminate it at the command line.

Within the loop, you need to get a `WatchKey`. There are two ways to do this. The most common is to call `take()`, which waits until an event is available. It throws an `InterruptedException` if it gets interrupted without finding a key. This allows you to end the program. The other way is to call `poll()`, which returns `null` if an event is not available. You can provide optional timeout parameters to wait up to a specific period of time for an event to show up.

```
watcher.take();                          // wait "forever" for an event
watcher.poll();                          // get event if present right NOW
watcher.poll(10, TimeUnit.SECONDS);      // wait up to 10 seconds for an event
watcher.poll(1, TimeUnit.MINUTES);       // wait up to 1 minute for an event
```

Next, you loop through any events on that key. In the case of rename, you'll get one key with two events—the `EVENT_CREATE` and `EVENT_DELETE`. Remember that you get all the events that happened since the last time you called `poll()` or `take()`. This means you can get multiple seemingly unrelated events out of the same key. They can be from different files but are for the same `WatchService`.

```
for (WatchEvent<?> event : key.pollEvents()) {
```

Finally, you call `key.reset()`. This is very important. If you forget to call reset, the program will work for the first event, but then you will not be notified of any other events.

There are a few limitations you should be aware of with *WatchService*. To begin with, it is slow. You could easily wait five seconds for the event to register. It also isn't 100 percent reliable. You can add code to check whether *kind* == *OVERFLOW*, but that just tells you something went wrong. You don't know what events you lost. In practice, you are unlikely to use *WatchService*.

`WatchService` only watches the files and directories immediately beneath it. What if we want to watch to see if either `p.txt` or `c.txt` is modified?

```
/dir
   | - parent
           | - p.txt
           | - child
                 | - c.txt
```

One way is to register both directories:

```
WatchService watcher =
  FileSystems.getDefault().newWatchService();
Path dir = Paths.get("/dir/parent");
dir.register(watcher, ENTRY_MODIFY);
Path child = Paths.get("dir/parent/child");
child.register(watcher, ENTRY_MODIFY);
```

This works. You can type in all the directories you want to watch. If we had a lot of child directories, this would quickly get to be too much work. Instead, we can have Java do it for us:

```
Path myDir = Paths.get("/dir/parent");
final WatchService watcher =                    // final so visitor can use it
  FileSystems.getDefault().newWatchService();
Files.walkFileTree(myDir, new SimpleFileVisitor<Path>() {
  public FileVisitResult preVisitDirectory(Path dir,
      BasicFileAttributes attrs) throws IOException {
      dir.register(watcher, ENTRY_MODIFY);     // watch each directory
     return FileVisitResult.CONTINUE;
  }
});
```

This code goes through the file tree recursively registering each directory with the watcher. The NIO.2 classes are designed to work together. For example, we could add `PathMatcher` to the previous example to only watch directories that have a specific pattern in their path.

CERTIFICATION OBJECTIVE

Serialization (Objective 8.2)

8.2 Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.

Imagine you want to save the state of one or more objects. If Java didn't have serialization (as the earliest version did not), you'd have to use one of the I/O classes to write out the state of the instance variables of all the objects you want to save. The worst part would be trying to reconstruct new objects that were virtually identical to the objects you were trying to save. You'd need your own protocol for the way in which you wrote and restored the state of each object, or you could end up setting variables with the wrong values. For example, imagine you stored an object that has instance variables for height and weight. At the time you save the state of the object, you could write out the height and weight as two `int`s in a file, but the order in which you write them is crucial. It would be all too easy to re-create the object but mix up the height and weight values—using the saved height as the value for the new object's weight and vice versa.

Serialization lets you simply say "save this object and all of its instance variables." Actually, it is a little more interesting than that because you can add, "...unless I've explicitly marked a variable as `transient`, which means, don't include the transient variable's value as part of the object's serialized state."

Working with ObjectOutputStream and ObjectInputStream

The magic of basic serialization happens with just two methods: one to serialize objects and write them to a stream, and a second to read the stream and deserialize objects.

```
ObjectOutputStream.writeObject()   // serialize and write

ObjectInputStream.readObject()     // read and deserialize
```

The `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes are considered to be *higher*-level classes in the java.io package, and as we learned earlier, that means you'll wrap them around *lower*-level classes, such as `java.io.FileOutputStream` and `java.io.FileInputStream`. Here's a small program that creates a `Cat` object, serializes it, and then deserializes it:

```
import java.io.*;

class Cat implements Serializable { }     // 1

public class SerializeCat {
  public static void main(String[] args) {
    Cat c = new Cat();                    // 2
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(c);                  // 3
      os.close();
    } catch (Exception e) { e.printStackTrace(); }

    try {
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      c = (Cat) ois.readObject();         // 4
      ois.close();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Let's take a look at the key points in this example:

1. We declare that the `Cat` class implements the `Serializable` interface. `Serializable` is a marker interface; it has no methods to implement. (In the next several sections, we'll cover various rules about when you need to declare classes `Serializable`.)
2. We make a new `Cat` object, which as we know is serializable.

3. We serialize the `Cat` object `c` by invoking the `writeObject()` method. It took a fair amount of preparation before we could actually serialize our `Cat`. First, we had to put all of our I/O-related code in a try/catch block. Next, we had to create a `FileOutputStream` to write the object to. Then, we wrapped the `FileOutputStream` in an `ObjectOutputStream`, which is the class that has the magic serialization method that we need. Remember that the invocation of `writeObject()` performs two tasks: it serializes the object, and then it writes the serialized object to a file.
4. We de-serialize the `Cat` object by invoking the `readObject()` method. The `readObject()` method returns an `Object`, so we have to cast the deserialized object back to a `Cat`. Again, we had to go through the typical I/O hoops to set this up.

This is a bare-bones example of serialization in action. Over the next few pages, we'll look at some of the more complex issues that are associated with serialization.

Object Graphs

What does it really mean to save an object? If the instance variables are all primitive types, it's pretty straightforward. But what if the instance variables are themselves references to *objects*? What gets saved? Clearly in Java it wouldn't make any sense to save the actual value of a reference variable, because the value of a Java reference has meaning only within the context of a single instance of a JVM. In other words, if you tried to restore the object in another instance of the JVM, even running on the same computer on which the object was originally serialized, the reference would be useless.

But what about the object that the reference refers to? Look at this class:

```
class Dog {
   private Collar theCollar;
   private int dogSize;
   public Dog(Collar collar, int size) {
     theCollar = collar;
     dogSize = size;
   }
   public Collar getCollar() { return theCollar; }
}
class Collar {
   private int collarSize;
   public Collar(int size) { collarSize = size; }
   public int getCollarSize() { return collarSize; }
}
```

Now make a dog... First, you make a `Collar` for the `Dog`:

```
Collar c = new Collar(3);
```

Then make a new `Dog`, passing it the `Collar`:

```
Dog d = new Dog(c, 8);
```

Now what happens if you save the `Dog`? If the goal is to save and then restore a `Dog`, and the restored `Dog` is an exact duplicate of the `Dog` that was saved, then the `Dog` needs a `Collar` that is an exact duplicate of the `Dog`'s `Collar` at the time the `Dog` was saved. That means both the `Dog` and the `Collar` should be saved.

And what if the `Collar` itself had references to other objects—perhaps a `Color` object? This gets quite complicated very quickly. If it were up to the programmer to know the internal structure of each object the `Dog` referred to, so that the programmer could be sure to save all the state of all those objects…whew. That would be a nightmare with even the simplest of objects.

Fortunately, the Java serialization mechanism takes care of all of this. When you serialize an object, Java serialization takes care of saving that object's entire "object graph." That means a deep copy of everything the saved object needs to be restored. For example, if you

serialize a `Dog` object, the `Collar` will be serialized automatically. And if the `Collar` class contained a reference to another object, *that* object would also be serialized, and so on. And the only object you have to worry about saving and restoring is the `Dog`. The other objects required to fully reconstruct that `Dog` are saved (and restored) automatically through serialization.

Remember, you do have to make a conscious choice to create objects that are serializable by implementing the `Serializable` interface. If we want to save `Dog` objects, for example, we'll have to modify the `Dog` class as follows:

```
class Dog implements Serializable {
   // the rest of the code as before
   // Serializable has no methods to implement
}
```

And now we can save the `Dog` with the following code:

```
import java.io.*;
public class SerializeDog {
  public static void main(String[] args) {
    Collar c = new Collar(3);
    Dog d = new Dog(c, 8);
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(d);
      os.close();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

But when we run this code we get a runtime exception, something like this

```
java.io.NotSerializableException: Collar
```

What did we forget? The `Collar` class must *also* be `Serializable`. If we modify the `Collar` class and make it serializable, then there's no problem:

```
class Collar implements Serializable {
   // same
}
```

Here's the complete listing:

```
import java.io.*;
public class SerializeDog {
  public static void main(String[] args) {
    Collar c = new Collar(3);
    Dog d = new Dog(c, 5);
    System.out.println("before: collar size is "
                       + d.getCollar().getCollarSize());
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(d);
      os.close();
    } catch (Exception e) { e.printStackTrace(); }
    try {
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (Dog) ois.readObject();
      ois.close();
    } catch (Exception e) { e.printStackTrace(); }

    System.out.println("after:  collar size is "
                       + d.getCollar().getCollarSize());
  }
}
class Dog implements Serializable {
   private Collar theCollar;
   private int dogSize;
   public Dog(Collar collar, int size) {
     theCollar = collar;
     dogSize = size;
   }
   public Collar getCollar() { return theCollar; }
}
class Collar implements Serializable {
   private int collarSize;
   public Collar(int size) { collarSize = size; }
   public int getCollarSize() { return collarSize; }
}
```

This produces the output:

```
before: collar size is 3
after:  collar size is 3
```

But what would happen if we didn't have access to the `Collar` class source code? In other words, what if making the `Collar` class serializable was not an option? Are we stuck with a non-serializable `Dog`?

Obviously, we could subclass the `Collar` class, mark the subclass as `Serializable`, and then use the `Collar` subclass instead of the `Collar` class. But that's not always an option either for several potential reasons:

1. The `Collar` class might be final, preventing subclassing.

   OR

2. The `Collar` class might itself refer to other non-serializable objects, and without knowing the internal structure of `Collar`, you aren't able to make all these fixes (assuming you even wanted to *try* to go down that road).

   OR

3. Subclassing is not an option for other reasons related to your design.

So…*then* what do you do if you want to save a Dog?

That's where the `transient` modifier comes in. If you mark the `Dog`'s `Collar` instance variable with `transient`, then serialization will simply skip the `Collar` during serialization:

```
class Dog implements Serializable {
   private transient Collar theCollar;  // add transient
   // the rest of the class as before
}

class Collar {              // no longer Serializable
   // same code
}
```

Now we have a `Serializable Dog`, with a non-`Serializable Collar`, but the `Dog` has marked the `Collar transient`; the output is

```
before: collar size is 3
Exception in thread "main" java.lang.NullPointerException
```

So now what can we do?

Using writeObject and readObject

Consider the problem: we have a `Dog` object we want to save. The `Dog` has a `Collar`, and the `Collar` has state that should also be saved as part of the `Dog`'s state. But…the `Collar` is not `Serializable`, so we must mark it `transient`. That means when the `Dog` is deserialized, it comes back with a null `Collar`. What can we do to somehow make sure that when the `Dog` is deserialized, it gets a new `Collar` that matches the one the `Dog` had when the `Dog` was saved?

Java serialization has a special mechanism just for this—a set of private methods you can implement in your class that, if present, will be invoked automatically during serialization and deserialization. It's almost as if the methods were defined in the `Serializable` interface, except they aren't. They are part of a special callback contract the serialization system offers you that basically says, "If you (the programmer) have a pair of methods matching this exact signature (you'll see them in a moment), these methods will be called during the serialization/deserialization process.

These methods let you step into the middle of serialization and deserialization. So they're perfect for letting you solve the `Dog`/`Collar` problem: when a `Dog` is being saved, you can step into the middle of serialization and say, "By the way, I'd like to add the state of the `Collar`'s variable (an `int`) to the stream when the `Dog` is serialized." You've manually added the state of the `Collar` to the `Dog`'s serialized representation, even though the `Collar` itself is not saved.

Of course, you'll need to restore the `Collar` during deserialization by stepping into the middle and saying, "I'll read that extra `int` I saved to the `Dog` stream, and use it to create a new `Collar`, and then assign that new `Collar` to the `Dog` that's being deserialized." The two special methods you define must have signatures that look *exactly* like this:

```
private void writeObject(ObjectOutputStream os) {
  // your code for saving the Collar variables
}

private void readObject(ObjectInputStream is) {
   // your code to read the Collar state, create a new Collar,
   // and assign it to the Dog
}
```

Yes, we're going to write methods that have the same name as the ones we've been calling! Where do these methods go? Let's change the `Dog` class:

```
class Dog implements Serializable {
  transient private Collar theCollar; // we can't serialize this
  private int dogSize;
  public Dog(Collar collar, int size) {
    theCollar = collar;
    dogSize = size;
  }
  public Collar getCollar() { return theCollar; }
  private void writeObject(ObjectOutputStream os) {
    //  throws IOException {                          // 1
   try {
    os.defaultWriteObject();                          // 2
```

```
    os.writeInt(theCollar.getCollarSize());              // 3
   } catch (Exception e) { e.printStackTrace(); }
  }
  private void readObject(ObjectInputStream is) {
   //   throws IOException, ClassNotFoundException {  // 4
   try {
    is.defaultReadObject();                              // 5
    theCollar = new Collar(is.readInt());                // 6
   } catch (Exception e) { e.printStackTrace(); }
  }
}
```

In our scenario we've agreed that, for whatever real-world reason, we can't serialize a `Collar` object, but we want to serialize a `Dog`. To do this we're going to implement `writeObject()` and `readObject()`. By implementing these two methods you're saying to the compiler: "If anyone invokes `writeObject()` or `readObject()` concerning a `Dog` object, use this code as part of the read and write."

Let's take a look at the preceding code.

1. Like most I/O-related methods `writeObject()` can throw exceptions. You can declare them or handle them, but we recommend handling them.
2. When you invoke `defaultWriteObject()` from within `writeObject()`, you're telling the JVM to do the normal serialization process for this object. When implementing `writeObject()`, you will typically request the normal serialization process *and* do some custom writing and reading, too.
3. In this case, we decided to write an extra `int` (the collar size) to the stream that's creating the serialized `Dog`. You can write extra stuff before and/or after you invoke `defaultWriteObject()`. But…when you read it back in, you have to read the extra stuff in the same order you wrote it.
4. Again, we chose to handle rather than declare the exceptions.
5. When it's time to deserialize, `defaultReadObject()` handles the normal deserialization you'd get if you didn't implement a `readObject()` method.
6. Finally, we build a new `Collar` object for the `Dog` using the collar size that we manually serialized. (We had to invoke `readInt()` *after* we invoked

`defaultReadObject()` or the streamed data would be out of sync!)

Remember, the most common reason to implement `writeObject()` and `readObject()` is when you have to save some part of an object's state manually. If you choose, you can write and read *all* of the state yourself, but that's very rare. So, when you want to do only a *part* of the serialization/deserialization yourself, you *must* invoke the `defaultReadObject()` and `defaultWriteObject()` methods to do the rest.

Which brings up another question—why wouldn't *all* Java classes be serializable? Why isn't class `Object` serializable? There are some things in Java that simply cannot be serialized because they are runtime specific. Things like streams, threads, runtime, etc., and even some GUI classes (which are connected to the underlying OS) cannot be serialized. What is and is not serializable in the Java API is *not* part of the exam, but you'll need to keep them in mind if you're serializing complex objects.

How Inheritance Affects Serialization

Serialization is very cool, but in order to apply it effectively you're going to have to understand how your class's superclasses affect serialization.

If a superclass is ***Serializable****, then, according to normal Java interface rules, all subclasses of that class automatically implement* ***Serializable*** *implicitly. In other words, a subclass of a class marked* ***Serializable*** *passes the IS-A test for* ***Serializable*** *and thus can be saved without having to explicitly mark the subclass as* ***Serializable****. You simply cannot tell whether a class is or is not* ***Serializable*** *unless you can see the class inheritance tree to see whether any other superclasses implement* ***Serializable****. If the class does not explicitly extend any other class and does not implement* ***Serializable****, then you know for certain that the class is not* ***Serializable,*** *because class* ***Object*** *does not implement* ***Serializable****.*

That brings up another key issue with serialization…what happens if a superclass is not marked `Serializable`, but the subclass is? Can the subclass still be serialized even if its superclass does not implement `Serializable`? Imagine this:

```
class Animal { }
class Dog extends Animal implements Serializable {
   // the rest of the Dog code
}
```

Now you have a `Serializable Dog` class with a non-`Serializable` superclass. This works! But there are potentially serious implications. To fully understand those implications, let's step back and look at the difference between an object that comes from deserialization versus an object created using `new`. Remember, when an object is constructed using `new` (as opposed to being deserialized), the following things happen (in this order):

1. All instance variables are assigned default values.
2. The constructor is invoked, which immediately invokes the superclass constructor (or another overloaded constructor, until one of the overloaded constructors invokes the superclass constructor).
3. All superclass constructors complete.
4. Instance variables that are initialized as part of their declaration are assigned their initial value (as opposed to the default values they're given prior to the superclass constructors completing).
5. The constructor completes.

But these things do not happen when an object is deserialized. When an instance of a serializable class is deserialized, the constructor does not run and instance variables are not given their initially assigned values! Think about it—if the constructor were invoked and/or instance variables were assigned the values given in their declarations, the object you're trying to restore would revert back to its original state, rather than coming back reflecting the changes in its state that happened sometime after it was created. For example, imagine you have a class that declares an instance variable and assigns it the `int` value `3` and includes a method that changes the instance variable value to `10`:

```
class Foo implements Serializable {
   int num = 3;
   void changeNum() {  num = 10;   }
}
```

Obviously, if you serialize a `Foo` instance *after* the `changeNum()` method runs, the value of the `num` variable should be `10`. When the `Foo` instance is deserialized, you want the `num` variable to still be `10`! You obviously don't want the initialization (in this case, the assignment of the value `3` to the variable `num`) to happen. Think of constructors and instance variable assignments together as part of one complete object initialization process (and, in fact, they do become one initialization method in the bytecode). The point is, when an object is deserialized we do not want any of the normal initialization to happen. We don't want the constructor to run, and we don't want the explicitly declared values to

be assigned. We want only the values saved as part of the serialized state of the object to be reassigned.

Of course, if you have variables marked `transient`, they will not be restored to their original state (unless you implement `readObject()`), but will instead be given the default value for that data type. In other words, even if you say

```
class Bar implements Serializable {
    transient int x = 42;
}
```

when the `Bar` instance is deserialized, the variable `x` will be set to a value of `0`. Object references marked `transient` will always be reset to `null`, regardless of whether they were initialized at the time of declaration in the class.

So, that's what happens when the object is deserialized, and the class of the serialized object directly extends `Object`, or has only serializable classes in its inheritance tree. It gets a little trickier when the serializable class has one or more non-serializable superclasses.

Getting back to our non-serializable `Animal` class with a serializable `Dog` subclass example:

```
class Animal {
   public String name;
}
class Dog extends Animal implements Serializable {
   // the rest of the Dog code
}
```

Because `Animal` is not serializable, any state maintained in the `Animal` class, even though the state variable is inherited by the `Dog`, isn't going to be restored with the `Dog` when it's deserialized! The reason is, the (unserialized) `Animal` part of the `Dog` is going to be reinitialized, just as it would be if you were making a new `Dog` (as opposed to deserializing one). That means all the things that happen to an object during construction will happen —but only to the `Animal` parts of a `Dog`. In other words, the instance variables from the `Dog`'s class will be serialized and deserialized correctly, but the inherited variables from the non-serializable `Animal` superclass will come back with their default/initially assigned values rather than the values they had at the time of serialization.

If you are a serializable class but your superclass is *not* serializable, then any instance variables you inherit from that superclass will be reset to the values they were given during the original construction of the object. This is because the non-serializable class constructor

will run!

In fact, every constructor above the first non-serializable class constructor will also run, no matter what, because once the first super constructor is invoked (during deserialization), it, of course, invokes its super constructor and so on, up the inheritance tree.

For the exam, you'll need to be able to recognize which variables will and will not be restored with the appropriate values when an object is deserialized, so be sure to study the following code example and the output:

```
import java.io.*;
class SuperNotSerial {
  public static void main(String [] args) {

    Dog d = new Dog(35, "Fido");
    System.out.println("before: " + d.name + " "
                       + d.weight);
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(d);
      os.close();
    } catch (Exception e) { e.printStackTrace(); }
    try {
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (Dog) ois.readObject();
      ois.close();
    } catch (Exception e) { e.printStackTrace(); }

    System.out.println("after:  " + d.name + " "
                       + d.weight);
  }
}
class Dog extends Animal implements Serializable {
  String name;
  Dog(int w, String n) {
    weight = w;          // inherited
    name = n;            // not inherited
  }
}
class Animal {           // not serializable !
  int weight = 42;
}
```

which produces the output:

```
before: Fido 35
after:  Fido 42
```

The key here is that because `Animal` is not serializable, when the `Dog` was deserialized, the `Animal` constructor ran and reset the `Dog`'s inherited weight variable.

If you serialize a collection or an array, every element must be serializable! A single non-serializable element will cause serialization to fail. Note also that although the collection interfaces are not serializable, the concrete collection classes in the Java API are.

Serialization Is Not for Statics

Finally, you might have noticed that we've talked only about instance variables, not static variables. Should static variables be saved as part of the object's state? Isn't the state of a static variable at the time an object was serialized important? Yes and no. It might be important, but it isn't part of the instance's state at all. Remember, you should think of static variables purely as *class* variables. They have nothing to do with individual instances. But serialization applies only to *objects*. And what happens if you deserialize three different `Dog` instances, all of which were serialized at different times and all of which were saved when the value of a static variable in class `Dog` was different? Which instance would "win"? Which instance's static value would be used to replace the one currently in the one and only `Dog` class that's currently loaded? See the problem?

Static variables are *never* saved as part of the object's state…because they do not belong to the object!

As simple as serialization code is to write, versioning problems can occur in the real world. If you save a *Dog* object using one version of the class, but attempt to deserialize it using a newer different version of the class, deserialization might fail. See the Java API for details about versioning issues and solutions.

CERTIFICATION SUMMARY

File I/O Remember that objects of type `File` can represent either files or directories, but that until you call `createNewFile()` or `mkdir()`, you haven't actually created anything on your hard drive. Classes in the `java.io` package are designed to be chained together. You will rarely use a `FileReader` or a `FileWriter` without "wrapping" them with a `BufferedReader` or `BufferedWriter` object, which gives you access to more powerful, higher-level methods. As of Java 5, the `PrintWriter` class has been enhanced with advanced `append()`, `format()`, and `printf()` methods, and when you couple that with new constructors that allow you to create `PrintWriters` directly from a `String` name or a `File` object, you may use `BufferedWriters` a lot less. The `Console` class allows you to read nonechoed input (returned in a `char[?]`) and is instantiated using `System.console()`.

NIO.2 objects of type `Path` can be files or directories and are a replacement of type `File`. `Paths` are created with `Paths.get()`. Utility methods in `Files` allow you to create, delete, move, copy, or check information about a `Path`. In addition, `BasicFileAttributes`, `DosFileAttributes` (Windows), and `PosixFileAttributes` (UNIX/Linux/Mac) allow you to check more advanced information about a `Path`. `BasicFileAttributeView`, `DosFileAttributeView`, and `PosixFileAttributeView` allow you to update advanced `Path` attributes.

Using a `DirectoryStream` allows you to iterate through a directory. Extending `SimpleFileVisitor` lets you walk a directory tree recursively looking at files and/or directories. With a `PathMatcher`, you can search directories for files using regex-esqu expressions called globs.

Finally, registering a `WatchService` provides notifications for new/changed/removed files or directories.

Serialization Serialization lets you save, ship, and restore everything you need to know about a *live* object. And when your object points to other objects, they get saved too. The `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes are used to serialize and deserialize objects. Typically, you wrap them around instances of `FileOutputStream` and `FileInputStream`, respectively.

The key method you invoke to serialize an object is `writeObject()`, and to deserialize an object invoke `readObject()`. In order to serialize an object, it must implement the `Serializable` interface. Mark instance variables `transient` if you don't want their state to be part of the serialization process. You can augment the serialization process for your class by implementing `writeObject()` and `readObject()`. If you do that, an embedded call to `defaultReadObject()` and `defaultWriteObject()` will handle the normal serialization tasks, and you can augment those invocations with manual *reading from* and *writing to* the stream.

If a superclass implements `Serializable` then all of its subclasses do too. If a

superclass doesn't implement `Serializable`, then when a subclass object is deserialized, the unserializable superclass's constructor runs—be careful! Finally, remember that serialization is about instances, so static variables aren't serialized.

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

File I/O (OCP Objectives 8.1 and 8.2)

- The classes you need to understand in `java.io` are `File`, `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`, `PrintWriter`, and `Console`.
- A new `File` object doesn't mean there's a new file on your hard drive.
- `File` objects can represent either a file or a directory.
- The `File` class lets you manage (add, rename, and delete) files and directories.
- The methods `createNewFile()` and `mkdir()` add entries to your file system.
- `FileWriter` and `FileReader` are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- `FileOutputStream` and `FileInputStream` are low-level I/O classes. You can use them to write and read bytes to and from files, but they should usually be wrapped.
- Classes in `java.io` are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)
- It's very common to "wrap" a `BufferedReader` around a `FileReader` or a `BufferedWriter` around a `FileWriter` to get access to higher-level (more convenient) methods.
- `PrintWriter`s can be used to wrap other `Writers`, but as of Java 5, they can be built directly from `File`s or `String`s.
- As of Java 5, `PrintWriter`s have `append()`, `format()`, and `printf()` methods.
- `Console` objects can read nonechoed input and are instantiated using `System.console()`.

Path, Paths, File, and Files (OCP Objectives 9.1 and 9.2)

- NIO.2 was introduced in Java 7.
- `Path` replaces `File` for a representation of a file or directory.

- `Paths.get()` lets you create a `Path` object.
- Static methods in `Files` let you work with `Path` objects.
- A `Path` object doesn't mean the file or directory exists on your hard drive.
- The methods `Files.createFile()` and `Files.createDirectory()` add entries to your file system.
- The `Files` class provides methods to move, copy, and delete `Path` objects.
- `Files.delete()` throws an exception and `Files.deleteIfExists()` returns false if the file does not exist.
- On `Path`, `normalize()` simplifies the path representation.
- On `Path`, `resolve()` and `relativize()` work with the relationship between two path objects.

File Attributes (OCP Objective 9.2)

- The `Files` class provides methods for common attributes, such as whether the file is executable and when it was last modified.
- For less common attributes the classes `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` read the attributes.
- `DosFileAttributes` works on Windows operating systems.
- `PosixFileAttributes` works on UNIX, Linux, and Mac operating systems.
- Attributes that can't be updated via the `Files` class are set using these classes: `BasicFileAttributeView`, `DosFileAttributeView`, `PosixFileAttributeView`, `FileOwnerAttributeView`, and `AclFileAttributeView`.

Directory Trees, Matching, and Watching for Changes (OCP Objective 9.2)

- `DirectoryStream` iterates through immediate children of a directory using glob patterns.
- `FileVisitor` walks recursively through a directory tree.
- You can override one or all of the methods of `SimpleFileVisitor` —`preVisitDirectory`, `visitFile`, `visitFileFailed`, and `postVisitDirectory`.
- You can change the flow of a file visitor by returning one of the `FileVisitResult` constants: CONTINUE, SKIP_SUBTREE, SKIP_SIBLINGS, or TERMINATE.
- `PathMatcher` checks if a path matches a glob pattern.

- Know what the following expressions mean for globs: `*`, `**`, `?`, and `{a,b}`.
- Directories register with `WatchService` to be notified about creation, deletion, and modification of files or immediate subdirectories.
- `PathMatcher` and `WatchService` use `FileSystems`-specific implementations.

Serialization (Objective 8.2)

- The classes you need to understand are all in the `java.io` package; they include `ObjectOutputStream` and `ObjectInputStream`, primarily, and `FileOutputStream` and `FileInputStream` because you will use them to create the low-level streams that the `ObjectXxxStream` classes will use.
- A class must implement `Serializable` before its objects can be serialized.
- The `ObjectOutputStream.writeObject()` method serializes objects, and the `ObjectInputStream.readObject()` method deserializes objects.
- If you mark an instance variable `transient`, it will not be serialized even though the rest of the object's state will be.
- You can supplement a class's automatic serialization process by implementing the `writeObject()` and `readObject()` methods. If you do this, embedding calls to `defaultWriteObject()` and `defaultReadObject()`, respectively, will handle the part of serialization that happens normally.
- If a superclass implements `Serializable`, then its subclasses do automatically.
- If a superclass doesn't implement `Serializable`, then, when a subclass object is deserialized, the superclass constructor will be invoked along with its superconstructor(s).

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

1. Note: The use of "drag-and-drop" questions has come and gone over the years. In case Oracle brings them back into fashion, we threw a couple of them in the book.
   Using the fewest fragments possible (and filling the fewest slots possible), complete the following code so that the class builds a directory named "`dir3`" and creates a file named "`file3`" inside "`dir3`." Note you can use each fragment either zero or one times.

   Code:

```
import java.io._____________

class Maker {
  public static void main(String[] args) {

    _____________  _____________  _____________

    _____________  _____________  _____________

    _____________  _____________  _____________

    _____________  _____________  _____________

    _____________  _____________  _____________

    _____________  _____________  _____________

    _____________  _____________  _____________

} }
```

Fragments:

```
File;       FileDescriptor;     FileWriter;     Directory;
try {       .createNewDir();    File dir        File
{ }         (Exception x)       ("dir3");       file
file        .createNewFile();   = new File      = new File
dir         (dir, "file3");     (dir, file);    .createFile();
} catch     ("dir3", "file3");  .mkdir();       File file
```

2. Given:

```
import java.io.*;

class Directories {
  static String [] dirs = {"dir1", "dir2"};
  public static void main(String [] args) {
    for (String d : dirs) {

      // insert code 1 here

      File file = new File(path, args[0]);

      // insert code 2 here
    }
  }
}
```

and that the invocation

```
java Directories file2.txt
```

is issued from a directory that has two subdirectories, "`dir1`" and "`dir2`," and that "`dir1`" has a file "`file1.txt`" and "`dir2`" has a file "`file2.txt`," and the output is "`false true`," which set(s) of code fragments must be inserted? (Choose all that apply.)

A.
```
String path = d;
    System.out.print(file.exists() + " ");
```

B.
```
String path = d;
    System.out.print(file.isFile() + " ");
```

C.
```
String path = File.separator + d;
    System.out.print(file.exists() + " ");
```

D.
```
String path = File.separator + d;
    System.out.print(file.isFile() + " ");
```

3. Given:

```
import java.io.*;
public class ReadingFor {
 public static void main(String[] args) {
   String s;
   try {
     FileReader fr = new FileReader("myfile.txt");
     BufferedReader br = new BufferedReader(fr);
     while((s = br.readLine()) != null)
       System.out.println(s);
     br.flush();
   } catch (IOException e) { System.out.println("io error"); }
 }
}
```

And given that `myfile.txt` contains the following two lines of data:

```
ab
cd
```

What is the result?

A.
```
ab
```

B.
```
abcd
```

C.
```
ab
cd
```

D.
```
a
b
c
d
```

E. Compilation fails

4. Given:

```
1. import java.io.*;
2. public class Talker {
3.  public static void main(String[] args) {
4.    Console c = System.console();
5.    String u = c.readLine("%s", "username: ");
6.    System.out.println("hello " + u);
7.    String pw;
8.    if(c != null && (pw = c.readPassword("%s", "password: ")) != null)
9.      // check for valid password
10.  }
11. }
```

If line 4 creates a valid `Console` object and if the user enters *fred* as a username and *1234* as a password, what is the result? (Choose all that apply.)

A.
```
username:
    password:
```

B.
```
username: fred
    password:
```

C.
```
username: fred
    password: 1234
```

D. Compilation fails

E. An exception is thrown at runtime

5. Given:

```
3. import java.io.*;
4. class Vehicle { }
5. class Wheels { }
6. class Car extends Vehicle implements Serializable {  }
7. class Ford extends Car { }
8. class Dodge extends Car {
9.  Wheels w = new Wheels();
10. }
```

Instances of which class(es) can be serialized? (Choose all that apply.)

A. `Car`
B. `Ford`
C. `Dodge`
D. `Wheels`
E. `Vehicle`

6. Which of the following creates a `Path` object pointing to `c:/temp/exam`? (Choose all that apply.)

A. `new Path("c:/temp/exam")`
B. `new Path("c:/temp", "exam")`
C. `Files.get("c:/temp/exam")`
D. `Files.get("c:/temp", "exam")`
E. `Paths.get("c:/temp/exam")`
F. `Paths.get("c:/temp", "exam")`

7. Given a directory tree at the root of the C: drive and the fact that no other files exist:

```
dir x - |
.........| - dir y
.........| - file a
```

and these two paths:

```
Path one = Paths.get("c:/x");
Path two = Paths.get("c:/x/y/a");
```

Which of the following statements prints out: `y/a`?

A. `System.out.println(one.relativize(two));`

B. `System.out.println(two.relativize(one));`

C. `System.out.println(one.resolve(two));`

D. `System.out.println(two.resolve(one));`

E. `System.out.println(two.resolve(two));`

F. None of the above

8. Given the following statements:

   I. A nonempty directory can usually be deleted using `Files.delete`

   II. A nonempty directory can usually be moved using `Files.move`

   III. A nonempty directory can usually be copied using `Files.copy`

   Which of the following is true?

   A. I only

   B. II only

   C. III only

   D. I and II only

   E. II and III only

   F. I and III only

   G. I, II, and III

9. Given:

```
new File("c:/temp/test.txt").delete();
```

   How would you write this line of code using Java 7 APIs?

A. `Files.delete(Paths.get("c:/temp/test.txt"));`

B. `Files.deleteIfExists(Paths.get("c:/temp/test.txt"));`

C. `Files.deleteOnExit(Paths.get("c:/temp/test.txt"));`

D. `Paths.get("c:/temp/test.txt").delete();`

E. `Paths.get("c:/temp/test.txt").deleteIfExists();`

F. `Paths.get("c:/temp/test.txt").deleteOnExit();`

10. Given:

```
public void read(Path dir) throws IOException {
  // CODE HERE
  System.out.println(attr.creationTime());
}
```

Which code inserted at `// CODE HERE` will compile and run without error on Windows? (Choose all that apply.)

A. `BasicFileAttributes attr = Files.readAttributes(dir, BasicFileAttributes.class);`

B. `BasicFileAttributes attr = Files.readAttributes(dir, DosFileAttributes.class);`

C.
```
DosFileAttributes attr = Files.readAttributes(dir,
BasicFileAttributes.class);
```

D.
```
DosFileAttributes attr = Files.readAttributes(dir,
DosFileAttributes.class);
```

E.
```
PosixFileAttributes attr = Files.readAttributes(dir,
PosixFileAttributes.class);
```

F.
```
BasicFileAttributes attr = new BasicFileAttributes(dir);
```

G.
```
BasicFileAttributes attr =dir.getBasicFileAttributes();
```

11. Which of the following are true? (Choose all that apply.)

    A. The class `AbstractFileAttributes` applies to all operating systems

    B. The class `BasicFileAttributes` applies to all operating systems

    C. The class `DosFileAttributes` applies to Windows-based operating systems

    D. The class `WindowsFileAttributes` applies to Windows-based operating systems

    E. The class `PosixFileAttributes` applies to all Linux/UNIX-based operating systems

    F. The class `UnixFileAttributes` applies to all Linux/UNIX-based operating systems

12. Given a partial directory tree:

```
dir x - |
.........| - dir y
.........| - file a
```

In what order can the following methods be called if walking the directory tree from `x`? (Choose all that apply.)

I: preVisitDirectory x

II: preVisitDirectory x/y

III: postVisitDirectory x/y

IV: postVisitDirectory x

V: visitFile x/a

A. I, II, III, IV, V
B. I, II, III, V, IV
C. I, V, II, III, IV
D. I, V, II, IV, III
E. V, I, II, III, IV
F. V, I, II, IV, III

13. Given:

```
public class MyFileVisitor extends SimpleFileVisitor<Path> {
  // more code here
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
    throws IOException {
    System.out.println("File " + file);
    if ( file.getFileName().endsWith("Test.java")) {
      // CODE HERE
    }
    return FileVisitResult.CONTINUE;
  }
  // more code here
}
```

Which code inserted at `// CODE HERE` would cause the `FileVisitor` to stop visiting files after it sees the file `Test.java`?

A. `return FileVisitResult.CONTINUE;`

B. `return FileVisitResult.END;`

C. `return FileVisitResult.SKIP_SIBLINGS;`

D. `return FileVisitResult.SKIP_SUBTREE;`

E. `return FileVisitResult.TERMINATE;`

F. `return null;`

14. Assume all the files referenced by these paths exist:

```
Path a = Paths.get("c:/temp/dir/a.txt");
Path b = Paths.get("c:/temp/dir/subdir/b.txt");
```

What is the correct string to pass to `PathMatcher` to match both these files?

A. `"glob:*/*.txt"`

B. `"glob:**.txt"`

C. `"glob:*.txt"`

D. `"glob:/*/*.txt"`

E. `"glob:/**.txt"`

F. `"glob:/*.txt"`

G. None of the above

15. Given a partial directory tree at the root of the drive:

```
dir x - |
........| - file a.txt
........| - dir y
................| - file b.txt
................| - dir y
........................| - file c.txt
```

And the following snippet:

```
Path dir = Paths.get("c:/x");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "**/*.txt")) {
for (Path path : stream) {
  System.out.println(path);
} }
```

What is the result?

A. c:/x/a.txt

B. c:/x/a.txt
   c:/x/y/b.txt
   c:/x/y/z/c.txt

C. Code compiles but does not output anything

D. Does not compile because `DirectoryStream` comes from `FileSystems`, not `Files`

E. Does not compile for another reason

16. Given a partial directory tree:

```
dir x - |
.........| - dir y
.........| -file a
```

and given that a valid `Path` object, `dir`, points to `x`, and given this snippet:

```
WatchKey key = dir.register(watcher, ENTRY_CREATE);
```

If a `WatchService` is set using the given `WatchKey`, what would be the result if a file is added to `dir y`?

A. No notice is given

B. A notice related to `dir x` is issued

C. A notice related to `dir y` is issued

D. Notices for both `dir x` and `dir y` are given

E. An exception is thrown

F. The behavior depends on the underlying operating system

17. Given:

```
import java.io.*;
class Player {
  Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
  CardPlayer() { System.out.print("c"); }
  public static void main(String[] args) {
    CardPlayer c1 = new CardPlayer();
    try {
   FileOutputStream fos = new FileOutputStream("play.txt");
   ObjectOutputStream os = new ObjectOutputStream(fos);
   os.writeObject(c1);
   os.close();
   FileInputStream fis = new FileInputStream("play.txt");
   ObjectInputStream is = new ObjectInputStream(fis);
   CardPlayer c2 = (CardPlayer) is.readObject();
   is.close();
  } catch (Exception x ) { }
 }
}
```

What is the result?

A. `pc`

B. `pcc`

C. `pcp`

D. `pcpc`

E. Compilation fails

F. An exception is thrown at runtime

18. Given:

```
import java.io.*;

class Keyboard { }
public class Computer implements Serializable {
  private Keyboard k = new Keyboard();
  public static void main(String[] args) {
    Computer c = new Computer();
    c.storeIt(c);
  }
  void storeIt(Computer c) {
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(c);
      os.close();
      System.out.println("done");
    } catch (Exception x) {System.out.println("exc"); }
  }
}
```

What is the result? (Choose all that apply.)

A. `exc`

B. `done`

C. Compilation fails

D. Exactly one object is serialized

E. Exactly two objects are serialized

19. Given:

```
import java.io.*;

public class TestSer {
  public static void main(String[] args) {
    SpecialSerial s = new SpecialSerial();
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(s);  os.close();
      System.out.print(++s.z + " ");

      ObjectInputStream is = new ObjectInputStream(
         new FileInputStream("myFile"));
      SpecialSerial s2 = (SpecialSerial)is.readObject();
      is.close();
      System.out.println(s2.y + " " + s2.z);
    } catch (Exception x) {System.out.println("exc"); }
  }
}
class SpecialSerial implements Serializable {
  transient int y = 7;
  static int z = 9;
}
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output is `10 0 9`

C. The output is `10 0 10`

D. The output is `10 7 9`

E. The output is `10 7 10`

F. In order to alter the standard deserialization process, you would implement the `readObject()` method in `SpecialSerial`

G. In order to alter the standard deserialization process, you would implement the `defaultReadObject()` method in `SpecialSerial`

A SELF TEST ANSWERS

1. ☑ Answer:

```
import java.io.File;
class Maker {
  public static void main(String[] args) {
    try {
      File dir = new File("dir3");
      dir.mkdir();
      File file = new File(dir, "file3");
      file.createNewFile();
    } catch (Exception x) { }
} }
```

Notes: The `new File` statements don't make actual files or directories, just objects. You need the `mkdir()` and `createNewFile()` methods to actually create the directory and the file. While drag-and-drop questions are no longer on the exam, it is still good to be able to complete them. (OCP Objective 8.2)

2. ☑ A and B are correct. Because you are invoking the program from the directory whose direct subdirectories are to be searched, you don't start your path with a

`File.separator` character. The `exists()` method tests for either files or directories; the `isFile()` method tests only for files. Since we're looking for a file, both methods work.

☒ C and D are incorrect based on the above. (OCP Objective 8.2)

3. ☑ E is correct. You need to call `flush()` only when you're writing data. Readers don't have `flush()` methods. If not for the call to `flush()`, answer C would be correct.

   ☒ A, B, C, and D are incorrect based on the above. (OCP Objective 8.2)

4. ☑ D is correct. The `readPassword()` method returns a `char[]`. If a `char[]` were used, answer B would be correct.

   ☒ A, B, C, and E are incorrect based on the above. (OCP Objective 8.1)

5. ☑ A and B are correct. `Dodge` instances cannot be serialized because they "have" an instance of `Wheels`, which is not serializable. `Vehicle` instances cannot be serialized even though the subclass `Car` can be.

   ☒ C, D, and E are incorrect based on the above. (Pre-OCPJP 7 only)

6. ☑ E and F are correct since `Paths` must be created using the `Paths.get()` method. This method takes a varargs `String` parameter, so you can pass as many path segments to it as you like.

   ☒ A and B are incorrect because you cannot construct a `Path` directly. C and D are incorrect because the `Files` class works with `Path` objects but does not create them from Strings. (Objective 9.1)

7. ☑ A is correct because it prints the path to get to `two` from `one`.

   ☒ B is incorrect because it prints out `../..`, which is the path to navigate to `one` from `two`. This is the reverse of what we want. C, D, and E are incorrect because it does not make sense to call `resolve` with absolute paths. They might print out `c:/x/c:/x/y/a`, `c:/x/y/a/c:/x`, and `c:/x/y/a/c:/x/y/a`, respectively. F is incorrect because of the above. Note that the directory structure provided is redundant. Neither `relativize()` nor `resolve()` requires either path to actually exist. (OCP Objective 9.1)

8. ☑ E is correct because a directory containing files or subdirectories is copied or moved in its entirety. Directories can only be deleted if they are empty. Trying to delete a nonempty directory will throw a `DirectoryNotEmptyException`. The question says "usually" because copy and move success depends on file permissions. Think about the most common cases when encountering words such as "usually" on the exam.

   ☒ A, B, C, D, F, and G are incorrect because of the above. (OCP Objective 9.2)

9. ☑ B is correct because, like the Java 7 code, it returns `false` if the file does not exist.

   ☒ A is incorrect because this code throws an exception if the file does not exist.C, D, E, and F are incorrect because they do not compile. There is no `deleteOnExit()` method, and file operations such as delete occur using the `Files` class rather than the path object directly. (OCP Objective 9.2)

10. ☑ A, B, and D are correct. Creation time is a basic attribute, which means you can read `BasicFileAttributes` or any of its subclasses to read it. `DosFileAttributes` is one such subclass.

    ☒ C is incorrect because you cannot cast a more general type to a more specific type. E is incorrect because this example specifies it is being run on Windows. Although it would work on UNIX, it throws an `UnsupportedOperationException` on Windows due to requesting the `WindowsFileSystemProvider` to get a POSIX class. F and G are incorrect because those methods do not exist. You must use the `Files` class to get the attributes. (OCP Objective 9.2)

11. ☑ B, C, and E are correct. `BasicFileAttributes` is the general superclass. `DosFileAttributes` subclasses `BasicFileAttributes` for Windows operating systems. `PosixFileAttributes` subclasses `BasicFileAttributes` for UNIX/Linux/Mac operating systems.

    ☒ A, D, and F are incorrect because no such classes exist. (OCP Objective 9.2)

12. ☑ B and C are correct because file visitor does a depth-first search. When files and directories are at the same level of the file tree, they can be visited in either order. Therefore, "`y`" and "`a`" could be reversed. All of the subdirectories and files are visited before `postVisit` is called on the directory.

    ☒ A, D, E, and F are incorrect because of the above. (OCP Objective 9.2)

13. ☑ E is correct because it is the correct constant to end the `FileVisitor`.

    ☒ B is incorrect because `END` is not defined as a result constant. A, C, and D are incorrect. Although they are valid constants, they do not end file visiting. `CONTINUE` proceeds as if nothing special has happened. `SKIP_SUBTREE` skips the subdirectory, which doesn't even make sense for a Java file. `SKIP_SIBLINGS` would skip any files in the same directory. Since we weren't told what the file structure is, we can't assume there weren't other directories or subdirectories. Therefore, we have to choose the most general answer of `TERMINATE`. F is incorrect because file visitor throws a `NullPointerException` if null is returned as the result. (OCP Objective 9.2)

14. ☑ B is correct. `**` matches zero or more characters, including multiple directories.

    ☒ A is incorrect because `*/` only matches one directory. It will match "`temp`" but not "`c:/temp`," let alone "`c:/temp/dir`." C is incorrect because `*.txt` only matches filenames and not directory paths. D, E, and F are incorrect because the paths we

want to match do not begin with a slash. G is incorrect because of the above. (OCP Objective 9.2)

15. ☑ C is correct because `DirectoryStream` only looks at files in the immediate directory. `**/*.txt` means zero or more directories followed by a slash, followed by zero or more characters followed by `.txt`. Since the slash is in there, it is required to match, which makes it mean one or more directories. However, this is impossible because `DirectoryStream` only looks at one directory. If the expression were simply `*.txt`, answer A would be correct.

    ☒ A, B, D, and E are incorrect because of the above. (OCP Objective 9.2).

16. ☑ A is correct because `WatchService` only looks at a single directory. If you want to look at subdirectories, you need to set recursive watch keys. This is usually done using a `FileVisitor`.

    ☒ B, C, D, E, and F are incorrect because of the above. (OCP Objective 9.2).

17. ☑ C is correct. It's okay for a class to implement `Serializable` even if its superclass doesn't. However, when you deserialize such an object, the non-serializable superclass must run its constructor. Remember, constructors don't run on deserialized classes that implement `Serializable`.

    ☒ A, B, D, E, and F are incorrect based on the above. (OCP Objective 8.2)

18. ☑ A is correct. An instance of type `Computer` Has-a `Keyboard`. Because `Keyboard` doesn't implement `Serializable`, any attempt to serialize an instance of `Computer` will cause an exception to be thrown.

    ☒ B, C, D, and E are incorrect based on the above. If `Keyboard` did implement `Serializable`, then two objects would have been serialized. (OCP Objective 8.2)

19. ☑ C and F are correct. C is correct because `static` and `transient` variables are not serialized when an object is serialized. F is a valid statement.

    ☒ A, B, D, and E are incorrect based on the above. G is incorrect because you don't implement the `defaultReadObject()` method; you call it from within the `readObject()` method, along with any custom read operations your class needs. (OCP Objective 8.2)