6

Strings, I/O, Formatting, and Parsing

CERTIFICATION OBJECTIVES

- Using String, StringBuilder, and StringBuffer
- File I/O using the java.io package
- Serialization using the java.io package
- Working with Dates, Numbers, and Currencies
- Using Regular Expressions

✓ Two-Minute Drill

Q&A Self Test

This chapter focuses on the various API-related topics that were added to the exam for Java 5. J2SE comes with an enormous API, and a lot of your work as a Java programmer will revolve around using this API. The exam team chose to focus on APIs for I/O, formatting, and parsing. Each of these topics could fill an entire book. Fortunately, you won't have to become a total I/O or regex 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 the String, I/O, formatting, and parsing objectives on the exam.

CERTIFICATION OBJECTIVE

String, StringBuilder, and StringBuffer (Exam Objective 3.1)

3.1 Discuss the differences between the String, StringBuilder, and StringBuffer classes.

Everything you needed to know about Strings in the SCJP 1.4 exam, you'll need to know for the SCJP 5 exam…plus, Sun added the *StringBuilder* class to the API, to provide faster, non-synchronized StringBuffer capability. The StringBuilder class has exactly the same methods as the old StringBuffer class, but StringBuilder is faster because its methods aren't synchronized. Both classes give you String-like objects that handle some of the String class's shortcomings (like immutability).

The String Class

This section covers the String class, and the key concept to understand is that once a String object is created, it can never be changed—so what is happening when a String object seems to be changing? Let's find out.

Strings Are Immutable Objects

We'll start with a little background information about strings. You may not need this for the test, but a little context will help. Handling "strings" of characters is a fundamental aspect of most programming languages. In Java, each character in a string is a 16-bit Unicode character. Because Unicode characters are 16 bits (not

the skimpy 7 or 8 bits that ASCII provides), a rich, international set of characters is easily represented in Unicode.

In Java, strings are objects. Just like other objects, you can create an instance of a String with the new keyword, as follows:

```
String s = new String();
```

This line of code creates a new object of class String, and assigns it to the reference variable s. So far, String objects seem just like other objects. Now, let's give the String a value:

```
s = "abcdef";
```

As you might expect, the String class has about a zillion constructors, so you can use a more efficient shortcut:

```
String s = new String("abcdef");
```

And just because you'll use strings all the time, you can even say this:

```
String s = "abcdef";
```

There are some subtle differences between these options that we'll discuss later, but what they have in common is that they all create a new String object, with a value of "abcdef", and assign it to a reference variable s. Now let's say that you want a second reference to the String object referred to by s:

```
String s2 = s;   //  refer s2 to the same String as s
```

So far so good. String objects seem to be behaving just like other objects, so what's all the fuss about?…Immutability! (What the heck is immutability?) Once you have assigned a String a value, that value can never change— it's immutable, frozen solid, won't budge, fini, done. (We'll talk about why later, don't let us forget.) The good news is that while the String object is immutable, its reference variable is not, so to continue with our previous example:

```
s = s.concat(" more stuff");  // the concat() method 'appends'
                              // a literal to the end
```

Now wait just a minute, didn't we just say that Strings were immutable? So what's all this "appending to the end of the string" talk? Excellent question: let's look at what really happened…

The VM took the value of String s (which was `"abcdef"`), and tacked `" more stuff"` onto the end, giving us the value `"abcdef more stuff"`. Since Strings are immutable, the VM couldn't stuff this new value into the old String referenced by s, so it created a new String object, gave it the value `"abcdef more stuff"`, and made s refer to it. At this point in our example, we have two String objects: the first one we created, with the value `"abcdef"`, and the second one with the value `"abcdef more stuff"`. Technically there are now three String objects, because the literal argument to concat, `" more stuff"`, is itself a new String object. But we have references only to `"abcdef"` (referenced by s2) and `"abcdef more stuff"` (referenced by s).

What if we didn't have the foresight or luck to create a second reference variable for the `"abcdef"` String before we called `s = s.concat(" more stuff");`? In that case, the original, unchanged String containing `"abcdef"` would still exist in memory, but it would be considered "lost." No code in our program has any way to reference it—it is lost to us. Note, however, that the original `"abcdef"` String didn't change (it can't, remember, it's immutable); only the reference variable s was changed, so that it would refer to a different String. Figure 6-1 shows what happens on the heap when you reassign a reference variable. Note that the dashed line indicates a deleted reference.

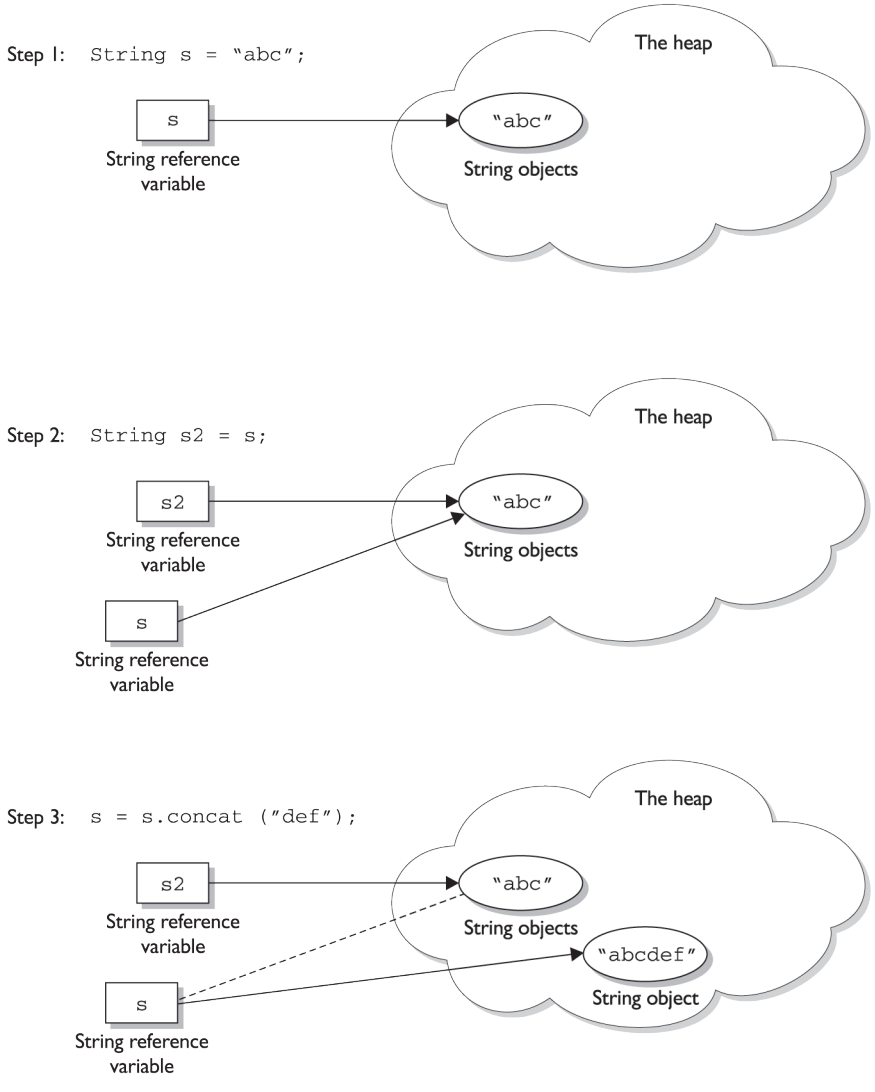
To review our first example:

```
String s = "abcdef";   // create a new String object, with
                       // value "abcdef", refer s to it
String s2 = s;         // create a 2nd reference variable
                       // referring to the same String

// create a new String object, with value "abcdef more stuff",
// refer s to it. (Change s's reference from the old String
// to the new String.) ( Remember s2 is still referring to
// the original "abcdef" String.)

s = s.concat(" more stuff");
```

FIGURE 6-1 String objects and their reference variables

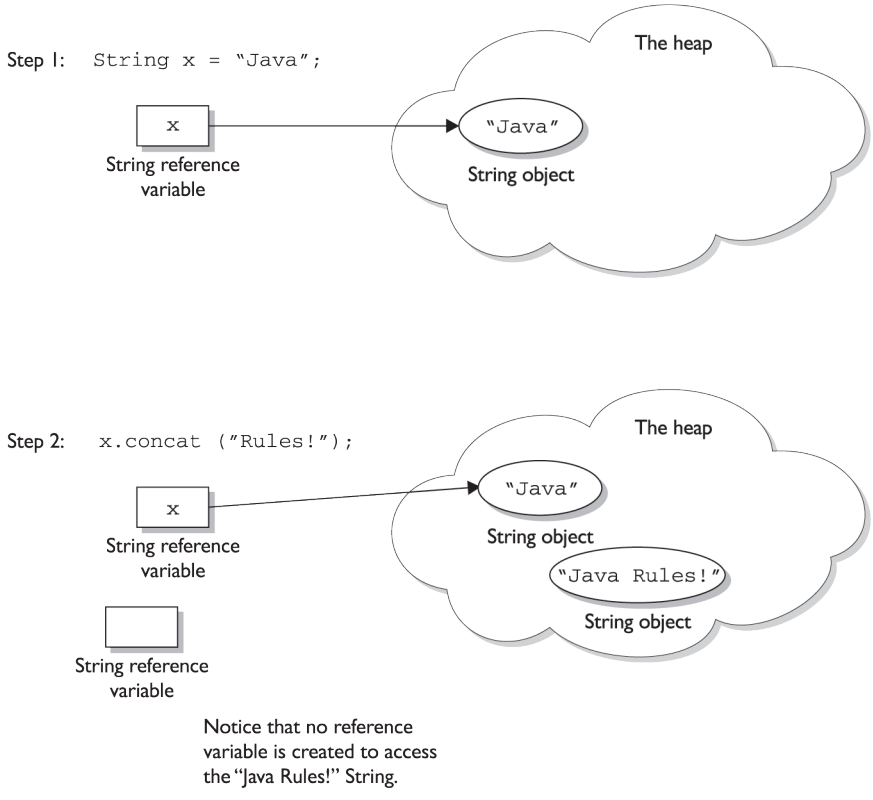


Let's look at another example:

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x);  // the output is "x = Java"
```

The first line is straightforward: create a new String object, give it the value `"Java"`, and refer `x` to it. Next the VM creates a second String object with the value `"Java Rules!"` but nothing refers to it. The second String object is instantly lost; you can't get to it. The reference variable `x` still refers to the original String with the value `"Java"`. Figure 6-2 shows creating a String without assigning a reference to it.

FIGURE 6-2 A String object is abandoned upon creation



Let's expand this current example. We started with

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x);   // the output is: x = Java
```

Now let's add

```
x.toUpperCase();
System.out.println("x = " + x);   // the output is still:
                                  // x = Java
```

(We actually did just create a new String object with the value `"JAVA"`, but it was lost, and `x` still refers to the original, unchanged String `"Java"`.) How about adding

```
x.replace('a', 'X');
System.out.println("x = " + x);   // the output is still:
                                  // x = Java
```

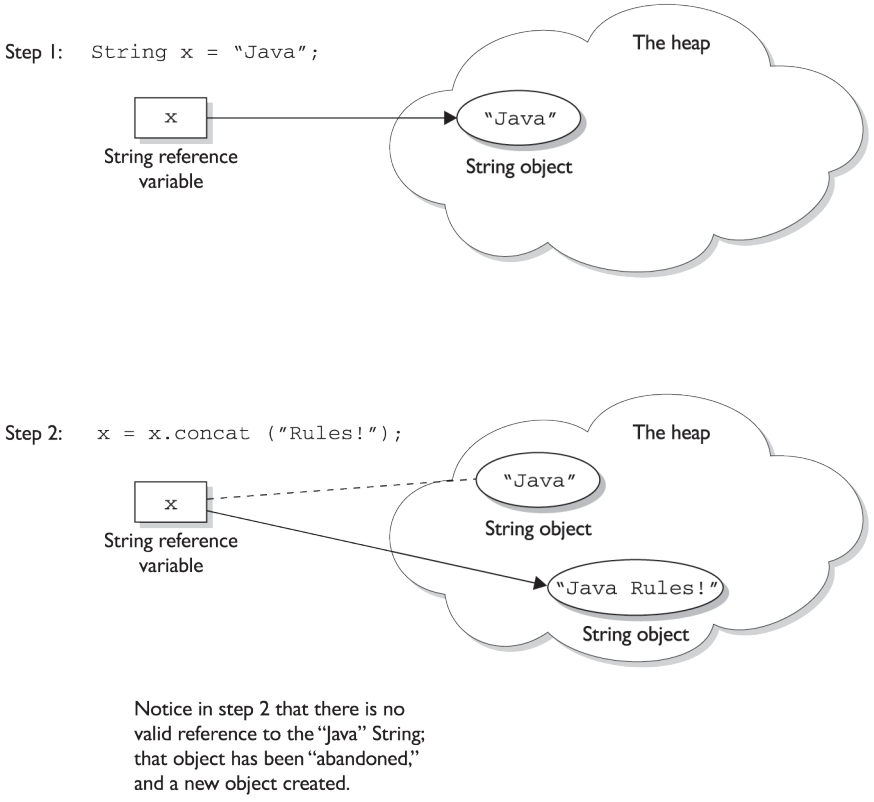
Can you determine what happened? The VM created yet another new String object, with the value `"JXvX"`, (replacing the `a`'s with `X`'s), but once again this new String was lost, leaving x to refer to the original unchanged and unchangeable String object, with the value `"Java"`. In all of these cases we called various String methods to create a new String by altering an existing String, but we never assigned the newly created String to a reference variable.

But we can put a small spin on the previous example:

```
String x = "Java";
x = x.concat(" Rules!");           // Now we're assigning the
                                   // new String to x
System.out.println("x = " + x);    // the output will be:
                                   // x = Java Rules!
```

This time, when the VM runs the second line, a new String object is created with the value of `"Java Rules!"`, and `x` is set to reference it. But wait, there's more—now the original String object, `"Java"`, has been lost, and no one is referring to it. So in both examples we created two String objects and only one reference variable, so one of the two String objects was left out in the cold. See Figure 6-3 for a graphic depiction of this sad story. The dashed line indicates a deleted reference.

FIGURE 6-3 An old String object being abandoned



Let's take this example a little further:

```
String x = "Java";
x = x.concat(" Rules!");
System.out.println("x = " + x);     // the output is:
                                    // x = Java Rules!

x.toLowerCase();                    // no assignment, create a
                                    // new, abandoned String

System.out.println("x = " + x);     // no assignment, the output
                                    // is still: x = Java Rules!
```

```
x = x.toLowerCase();                // create a new String,
                                    // assigned to x
System.out.println("x = " + x);     // the assignment causes the
                                    // output: x = java rules!
```

The preceding discussion contains the keys to understanding Java String immutability. If you really, really get the examples and diagrams, backwards and forwards, you should get 80 percent of the String questions on the exam correct.

We will cover more details about Strings next, but make no mistake—in terms of bang for your buck, what we've already covered is by far the most important part of understanding how String objects work in Java.

We'll finish this section by presenting an example of the kind of devilish String question you might expect to see on the exam. Take the time to work it out on paper (as a hint, try to keep track of how many objects and reference variables there are, and which ones refer to which).

```
String s1 = "spring ";
String s2 = s1 + "summer ";
s1.concat("fall ");
s2.concat(s1);
s1 += "winter ";
System.out.println(s1 + " " + s2);
```

What is the output? For extra credit, how many String objects and how many reference variables were created prior to the `println` statement?

Answer: The result of this code fragment is "spring winter spring summer". There are two reference variables, `s1` and `s2`. There were a total of eight String objects created as follows: "spring", "summer " (lost), "spring summer", "fall" (lost), "spring fall" (lost), "spring summer spring" (lost), "winter" (lost), "spring winter" (at this point "spring" is lost). Only two of the eight String objects are not lost in this process.

Important Facts About Strings and Memory

In this section we'll discuss how Java handles String objects in memory, and some of the reasons behind these behaviors.

One of the key goals of any good programming language is to make efficient use of memory. As applications grow, it's very common for String literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the

universe of String literals for a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.) Now we can start to see why making String objects immutable is such a good idea. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value.

You might say, "Well that's all well and good, but what if someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked `final`. Nobody can override the behaviors of any of the String methods, so you can rest assured that the String objects you are counting on to be immutable will, in fact, be immutable.

Creating New Strings

Earlier we promised to talk more about the subtle differences between the various methods of creating a String. Let's look at a couple of examples of how a String might be created, and let's further assume that no other String objects exist in the pool:

```
String s = "abc";     //  creates one String object and one
                      //  reference variable
```

In this simple case, "abc" will go in the pool and `s` will refer to it.

```
String s = new String("abc");  // creates two objects,
                               // and one reference variable
```

In this case, because we used the `new` keyword, Java will create a new String object in normal (non-pool) memory, and `s` will refer to it. In addition, the literal `"abc"` will be placed in the pool.

Important Methods in the String Class

The following methods are some of the more commonly used methods in the String class, and also the ones that you're most likely to encounter on the exam.

- **charAt()** Returns the character located at the specified index
- **concat()** Appends one String to the end of another ("+" also works)
- **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case
- **length()** Returns the number of characters in a String
- **replace()** Replaces occurrences of a character with a new character
- **substring()** Returns a part of a String
- **toLowerCase()** Returns a String with uppercase characters converted
- **toString()** Returns the value of a String
- **toUpperCase()** Returns a String with lowercase characters converted
- **trim()** Removes whitespace from the ends of a String

Let's look at these methods in more detail.

public char charAt(int index) This method returns the character located at the String's specified index. Remember, String indexes are zero-based—for example,

```
String x = "airplane";
System.out.println( x.charAt(2) );       //  output is 'r'
```

public String concat(String s) This method returns a String with the value of the String passed in to the method appended to the end of the String used to invoke the method—for example,

```
String x = "taxi";
System.out.println( x.concat(" cab") ); // output is "taxi cab"
```

The overloaded + and += operators perform functions similar to the `concat()` method—for example,

```
String x = "library";
System.out.println( x + " card");    // output is "library card"

String x = "Atlantic";
x += " ocean"
System.out.println( x );         // output is "Atlantic ocean"
```

In the preceding "Atlantic ocean" example, notice that the value of x really did change! Remember that the += operator is an assignment operator, so line 2 is really creating a new String, "Atlantic ocean", and assigning it to the x variable. After line 2 executes, the original String x was referring to, "Atlantic", is abandoned.

public boolean equalsIgnoreCase(String s) This method returns a boolean value (true or false) depending on whether the value of the String in the argument is the same as the value of the String used to invoke the method. This method will return true even when characters in the String objects being compared have differing cases—for example,

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT"));   // is "true"
System.out.println( x.equalsIgnoreCase("tixe"));   // is "false"
```

public int length() This method returns the length of the String used to invoke the method—for example,

```
String x = "01234567";
System.out.println( x.length() );     // returns "8"
```

public String replace(char old, char new) This method returns a String whose value is that of the String used to invoke the method, updated so that any occurrence of the char in the first argument is replaced by the char in the second argument—for example,

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') );    // output is
                                              // "oXoXoXoX"
```

public String substring(int begin)
public String substring(int begin, int end) The substring() method is used to return a part (or substring) of the String used to invoke the method. The first argument represents the starting location (zero-based) of the substring. If the call has only one argument, the substring returned will include the characters to the end of the original String. If the call has two arguments, the substring returned will end with the character located in the *n*th position of the original String where *n* is the

exam Watch

Arrays have an attribute (not a method), called `length`***. You may encounter questions in the exam that attempt to use the length() method on an array, or that attempt to use the length attribute on a String. Both cause compiler errors—for example,***

```
String x = "test";
System.out.println( x.length );     // compiler error
```

or

```
String[] x = new String[3];
System.out.println( x.length() );   // compiler error
```

second argument. Unfortunately, the ending argument is not zero-based, so if the second argument is 7, the last character in the returned String will be in the original String's 7 position, which is index 6 (ouch). Let's look at some examples:

```
String x = "0123456789";          // as if by magic, the value
                                  // of each char
                                  // is the same as its index!
System.out.println( x.substring(5) );     // output is  "56789"
System.out.println( x.substring(5, 8));   // output is "567"
```

The first example should be easy: start at index 5 and return the rest of the String. The second example should be read as follows: start at index 5 and return the characters up to and including the 8th position (index 7).

public String toLowerCase() This method returns a String whose value is the String used to invoke the method, but with any uppercase characters converted to lowercase—for example,

```
String x = "A New Moon";
System.out.println( x.toLowerCase() );     // output is
                                           // "a new moon"
```

public String toString() This method returns the value of the String used to invoke the method. What? Why would you need such a seemingly "do nothing" method? All objects in Java must have a `toString()` method, which typically returns a String that in some meaningful way describes the object in question. In the case of a String object, what more meaningful way than the String's value? For the sake of consistency, here's an example:

```
String x = "big surprise";
System.out.println( x.toString() );      // output -
                                         // reader's exercise
```

public String toUpperCase() This method returns a String whose value is the String used to invoke the method, but with any lowercase characters converted to uppercase—for example,

```
String x = "A New Moon";
System.out.println( x.toUpperCase() );     // output is
                                           // "A NEW MOON"
```

public String trim() This method returns a String whose value is the String used to invoke the method, but with any leading or trailing blank spaces removed—for example,

```
String x = "       hi         ";
System.out.println( x + "x" );              // result is
                                            // "     hi    x"
System.out.println( x.trim() + "x");        // result is "hix"
```

The StringBuffer and StringBuilder Classes

The java.lang.StringBuffer and java.lang.StringBuilder classes should be used when you have to make a lot of modifications to strings of characters. As we discussed in the previous section, String objects are immutable, so if you choose to do a lot of manipulations with String objects, you will end up with a lot of abandoned String objects in the String pool. (Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded String pool objects.) On the other hand, objects of type StringBuffer and StringBuilder can be modified over and over again without leaving behind a great effluence of discarded String objects.

on the Job

A common use for StringBuffers and StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and StringBuffer objects are the ideal way to handle a block of data, pass it on, and then reuse the same memory to handle the next block of data.

StringBuffer vs. StringBuilder

The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. (More about thread safety in Chapter 9.) Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So apart from synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa. You might encounter questions on the exam that have to do with using these two classes in the creation of thread-safe applications, and we'll discuss how *that* works in Chapter 9.

Using StringBuilder and StringBuffer

In the previous section, we saw how the exam might test your understanding of String immutability with code fragments like this:

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x);      //  output is "x = abc"
```

Because no new assignment was made, the new String object created with the `concat()` method was abandoned instantly. We also saw examples like this:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x);     // output is "x = abcdef"
```

We got a nice new String out of the deal, but the downside is that the old String "abc" has been lost in the String pool, thus wasting memory. If we were using a StringBuffer instead of a String, the code would look like this:

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb);      // output is "sb = abcdef"
```

All of the StringBuffer methods we will discuss operate on the value of the StringBuffer object invoking the method. So a call to `sb.append("def");` is actually appending "def" to itself (StringBuffer `sb`). In fact, these method calls can be chained to each other—for example,

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );              // output is  "fed---cba"
```

Notice that in each of the previous two examples, there was a single call to new, concordantly in each example we weren't creating any extra objects. Each example needed only a single StringXxx object to execute.

Important Methods in the StringBuffer and StringBuilder Classes

The following method returns a StringXxx object with the argument's value appended to the value of the object that invoked the method.

public synchronized StringBuffer append(String s) As we've seen earlier, this method will update the value of the object that invoked the method, whether or not the return is assigned to a variable. This method will take many different arguments, including boolean, char, double, float, int, long, and others, but the most likely use on the exam will be a String argument—for example,

```
StringBuffer sb = new StringBuffer("set ");
sb.append("point");
System.out.println(sb);       // output is "set point"
StringBuffer sb2 = new StringBuffer("pi = ");
sb2.append(3.14159f);
System.out.println(sb2);      // output is  "pi = 3.14159"
```

public StringBuilder delete(int start, int end) This method returns a StringBuilder object and updates the value of the StringBuilder object that invoked the method call. In both cases, a substring is removed from the original object. The starting index of the substring to be removed is defined by the first argument (which is zero-based), and the ending index of the substring to be removed is defined by the second argument (but it is one-based)! Study the following example carefully:

```
StringBuilder sb = new StringBuilder("0123456789");
System.out.println(sb.delete(4,6));      // output is "01236789"
```

exam Watch

The exam will probably test your knowledge of the difference between String and StringBuffer objects. Because StringBuffer objects are changeable, the following code fragment will behave differently than a similar code fragment that uses String objects:

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println( sb );
```

In this case, the output will be: `"abcdef"`

public StringBuilder insert(int offset, String s) This method returns a StringBuilder object and updates the value of the StringBuilder object that invoked the method call. In both cases, the String passed in to the second argument is inserted into the original StringBuilder starting at the offset location represented by the first argument (the offset is zero-based). Again, other types of data can be passed in through the second argument (`boolean`, `char`, `double`, `float`, `int`, `long`, and so on), but the String argument is the one you're most likely to see:

```
StringBuilder sb = new StringBuilder("01234567");
sb.insert(4, "---");
System.out.println( sb );          //   output is  "0123---4567"
```

public synchronized StringBuffer reverse() This method returns a StringBuffer object and updates the value of the StringBuffer object that invoked the method call. In both cases, the characters in the StringBuffer are reversed, the first character becoming the last, the second becoming the second to the last, and so on:

```
StringBuffer s = new StringBuffer("A man a plan a canal Panama");
sb.reverse();
System.out.println(sb); // output: "amanaP lanac a nalp a nam A"
```

public String toString() This method returns the value of the StringBuffer object that invoked the method call as a String:

```
StringBuffer sb = new StringBuffer("test string");
System.out.println( sb.toString() );  // output is "test string"
```

That's it for StringBuffers and StringBuilders. If you take only one thing away from this section, it's that unlike Strings, StringBuffer objects and StringBuilder objects can be changed.

exam Watch

Many of the exam questions covering this chapter's topics use a tricky (and not very readable) bit of Java syntax known as "chained methods." A statement with chained methods has this general form:

```
result = method1().method2().method3();
```

In theory, any number of methods can be chained in this fashion, although typically you won't see more than three. Here's how to decipher these "handy Java shortcuts" when you encounter them:

1. ***Determine what the leftmost method call will return (let's call it x).***
2. ***Use x as the object invoking the second (from the left) method. If there are only two chained methods, the result of the second method call is the expression's result.***
3. ***If there is a third method, the result of the second method call is used to invoke the third method, whose result is the expression's result—for example,***

```
String x = "abc";
String y = x.concat("def").toUpperCase().replace('C','x');
//chained methods
System.out.println("y = " + y); // result is "y = ABxDEF"
```

Let's look at what happened. The literal `def` ***was concatenated to*** `abc`***, creating a temporary, intermediate String (soon to be lost), with the value*** `abcdef`***. The*** `toUpperCase()` ***method created a new (soon to be lost) temporary String with the value*** `ABCDEF`***. The*** `replace()` ***method created a final String with the value*** `ABxDEF`***, and referred*** `y` ***to it.***

CERTIFICATION OBJECTIVE

File Navigation and I/O (Exam Objective 3.2)

3.2 Given a scenario involving navigating file systems, reading from files, or writing to files, develop the correct solution using the following classes (sometimes in combination), from java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter, and PrintWriter.

I/O has had a strange history with the SCJP certification. It was included in all the versions of the exam up to and including 1.2, then removed from the 1.4 exam, and then re-introduced for Java 5.

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 5 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 class File 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. FileReaders are usually *wrapped* by higher-level objects such as BufferedReaders, 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 FileReaders, BufferedReaders 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. In addition, the BufferedWriter class provides a `newLine()` method that makes it easy 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 very flexible and powerful.

exam Watch

Stream classes are used to read and write bytes, and Readers and Writers are used to read and write characters. Since all of the file I/O on the exam is related to characters, if you see API class names containing the word "Stream", for instance DataOutputStream, then the question is probably about serialization, or something unrelated to the actual I/O objective.

Creating Files Using Class File

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 Java 5 exam focuses 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 that 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.

exam Watch

Remember, the exam creators are trying to jam as much code as they can into a small space, so in the previous example, instead of these three lines code,

```
boolean newFile = false;
...
newFile = file.createNewFile();
System.out.println(newFile);
```

You might see something like the following single line of code, which is a bit harder to read, but accomplishes the same thing:

```
System.out.println(file.createNewFile());
```

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 bytes 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 `size`, and we looped through the `in` array printing out each character we read, 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 been 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 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.

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 very specific purpose (creating high cohesion), 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 pouring 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 we've artificially reduced the API. In terms of studying for exam Objective 3.2, we can imagine that the entire java.io package consisted of the classes listed in exam Objective 3.2, and summarized in Table 6-1, our mini I/O API.

TABLE 6-1 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()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
			*Discussed later

Now let's say that 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 6-1 we can see that BufferedWriter has a `newLine()` method. That sounds a little better than 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, Table 6-1.

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 6-1 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:

1. It can be constructed using a File.
2. 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
```

exam Watch

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 that you use Table 6-1 as a reference, and write 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 Reader or a Writer or a Stream. Specifically, create a FileReader, a FileWriter, a PrintWriter, a FileInputStream, 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 File object, capital D, (even though it's of type File.) We'll call an actual directory on a computer a directory, 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 Reader or Writer will automatically create a file for you 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 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 that we already have a subdirectory called `existingDir` in which resides an existing file `existingDirFile.txt`, which contains several lines of text. When you 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 the 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 also let's 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 that 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, which 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 vary :)

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 serialization, which is covered in the next section).

CERTIFICATION OBJECTIVE

Serialization (Exam Objective 3.3)

3.3 Develop code that serializes and/or de-serializes objects using the following APIs from java.io: DataInputStream, DataOutputStream, FileInputStream, FileOutputStream, ObjectInputStream, ObjectOutputStream, and Serializable.

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 its 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 that 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 set of 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—like 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 os) {
   // 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(); }
  }
}
```

Let's take a look at the preceding code.

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".

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.

exam Watch

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 if 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 vs. 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 `defaultReadObject()`), 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, 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.

exam Watch

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 while the collection interfaces are not serializable, the concrete collection classes in the Java API are.

Serialization Is Not for Statics

Finally, you might notice 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!

exam Watch

What about DataInputStream and DataOutputStream? They're in the objectives! It turns out that while the exam was being created, it was decided that those two classes wouldn't be on the exam after all, but someone forgot to remove them from the objectives! So you get a break. That's one less thing you'll have to worry about. Congratulations, you're closer than you thought.

on the job

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 OBJECTIVE

Dates, Numbers, and Currency (Exam Objective 3.4)

3.4 Use standard J2SE APIs in the java.text package to correctly format or parse dates, numbers and currency values for a specific locale; and, given a scenario, determine the appropriate methods to use if you want to use the default locale or a specific locale. Describe the purpose and use of the java.util.Locale class.

The Java API provides an extensive (perhaps a little *too* extensive) set of classes to help you work with dates, numbers, and currency. The exam will test your knowledge of the basic classes and methods you'll use to work with dates and such. When you've finished this section you should have a solid foundation in tasks such as creating new Date and DateFormat objects, converting Strings to Dates and back again, performing Calendaring functions, printing properly formatted currency values, and doing all of this for locations around the globe. In fact, a large part of why this section was added to the exam was to test whether you can do some basic internationalization (often shortened to "i18n").

Working with Dates, Numbers, and Currencies

If you want to work with dates from around the world (and who doesn't?), you'll need to be familiar with at least four classes from the java.text and java.util packages. In fact, we'll admit it right up front, you might encounter questions on the exam that use classes that aren't specifically mentioned in the Sun objective. Here are the four date related classes you'll need to understand:

- **`java.util.Date`** Most of this class's methods have been deprecated, but you can use this class to bridge between the `Calendar` and `DateFormat` class. An instance of `Date` represents a mutable date and time, to a millisecond.
- **`java.util.Calendar`** This class provides a huge variety of methods that help you convert and manipulate dates and times. For instance, if you want to add a month to a given date, or find out what day of the week January 1, 3000 falls on, the methods in the `Calendar` class will save your bacon.
- **`java.text.DateFormat`** This class is used to format dates not only providing various styles such as "01/01/70" or "January 1, 1970," but also to format dates for numerous locales around the world.
- **`java.text.NumberFormat`** This class is used to format numbers and currencies for locales around the world.
- **`java.util.Locale`** This class is used in conjunction with `DateFormat` and `NumberFormat` to format dates, numbers and currency for specific locales. With the help of the `Locale` class you'll be able to convert a date like "10/08/2005" to "Segunda-feira, 8 de Outubro de 2005" in no time. If you want to manipulate dates without producing formatted output, you can use the Locale class directly with the Calendar class.

Orchestrating Date- and Number-Related Classes

When you work with dates and numbers, you'll often use several classes together. It's important to understand how the classes we described above relate to each other, and when to use which classes in combination. For instance, you'll need to know that if you want to do date formatting for a specific locale, you need to create your Locale object before your DateFormat object, because you'll need your Locale object as an argument to your DateFormat factory method. Table 6-2 provides a quick overview of common date- and number-related use cases and solutions using these classes. Table 6-2 will undoubtedly bring up specific questions about individual classes, and we will dive into specifics for each class next. Once you've gone through the class level discussions, you should find that Table 6-2 provides a good summary.

TABLE 6-2 Common Use Cases When Working with Dates and Numbers

Use Case	Steps
Get the current date and time.	1. Create a Date: `Date d = new Date();` 2. Get its value: `String s = d.toString();`
Get an object that lets you perform date and time calculations in your locale.	1. Create a Calendar: `Calendar c = Calendar.getInstance();` 2. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations in a different locale.	1. Create a Locale:
	`Locale loc = new Locale(language);` or
	`Locale loc = new Locale(language, country);`
	2. Create a Calendar for that locale:
	`Calendar c = Calendar.getInstance(loc);`
	3. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations, and then format it for output in different locales with different date styles.	1. Create a Calendar:
	`Calendar c = Calendar.getInstance();`
	2. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	3. Convert your Calendar to a Date:
	`Date d = c.getTime();`
	4. Create a DateFormat for each Locale:
	`DateFormat df = DateFormat.getDateInstance (style, loc);`
	5. Use the format() method to create formatted dates:
	`String s = df.format(d);`

Get an object that lets you format numbers or currencies across many different locales.	1. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	2. Create a NumberFormat:
	`NumberFormat nf = NumberFormat.getInstance(loc);`
	`-or- NumberFormat nf = NumberFormat. getCurrencyInstance(loc);`
	3. Use the format() method to create formatted output:
	`String s = nf.format(someNumber);`

The Date Class

The Date class has a checkered past. Its API design didn't do a good job of handling internationalization and localization situations. In its current state, most of its methods have been deprecated, and for most purposes you'll want to use the Calendar class instead of the Date class. The Date class is on the exam for several reasons: you might find it used in legacy code, it's really easy if all you want is a quick and dirty way to get the current date and time, it's good when you want a universal time that is not affected by time zones, and finally, you'll use it as a temporary bridge to format a Calendar object using the DateFormat class.

As we mentioned briefly above, an instance of the Date class represents a single date and time. Internally, the date and time is stored as a primitive `long`. Specifically, the `long` holds the number of milliseconds (you know, 1000 of these per second), between the date being represented and January 1, 1970.

Have you ever tried to grasp how big really big numbers are? Let's use the `Date` class to find out how long it took for a trillion milliseconds to pass, starting at January 1, 1970:

```
import java.util.*;
class TestDates {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);  // a trillion!
    System.out.println("1st date " + d1.toString());
  }
}
```

On our JVM, which has a US locale, the output is

```
1st date Sat Sep 08 19:46:40 MDT 2001
```

Okay, for future reference remember that there are a trillion milliseconds for every 31 and 2/3 years.

Although most of Date's methods have been deprecated, it's still acceptable to use the `getTime` and `setTime` methods, although as we'll soon see, it's a bit painful. Let's add an hour to our Date instance, `d1`, from the previous example:

```
import java.util.*;
class TestDates {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);  // a trillion!
```

```
    System.out.println("1st date " + d1.toString());
    d1.setTime(d1.getTime() + 3600000); // 3600000 millis / hour
    System.out.println("new time " + d1.toString());
  }
}
```

which produces (again, on our JVM):

```
1st date Sat Sep 08 19:46:40 MDT 2001
new time Sat Sep 08 20:46:40 MDT 2001
```

Notice that both `setTime()` and `getTime()` used the handy millisecond scale... if you want to manipulate dates using the Date class, that's your only choice. While that wasn't too painful, imagine how much fun it would be to add, say, a year to a given date.

We'll revisit the Date class later on, but for now the only other thing you need to know is that if you want to create an instance of Date to represent "now," you use Date's no-argument constructor:

```
Date now = new Date();
```

(We're guessing that if you call `now.getTime()`, you'll get a number somewhere between one trillion and two trillion.)

The Calendar Class

We've just seen that manipulating dates using the Date class is tricky. The Calendar class is designed to make date manipulation easy! (Well, easier.) While the Calendar class has about a million fields and methods, once you get the hang of a few of them the rest tend to work in a similar fashion.

When you first try to use the `Calendar` class you might notice that it's an abstract class. You can't say

```
Calendar c = new Calendar();   // illegal, Calendar is abstract
```

In order to create a `Calendar` instance, you have to use one of the overloaded `getInstance()` static factory methods:

```
Calendar cal = Calendar.getInstance();
```

When you get a Calendar reference like `cal`, from above, your Calendar reference variable is actually referring to an instance of a concrete subclass of Calendar. You can't know for sure what subclass you'll get (`java.util.GregorianCalendar` is what you'll almost certainly get), but it won't matter to you. You'll be using Calendar's API. (As Java continues to spread around the world, in order to maintain cohesion, you might find additional, locale-specific subclasses of `Calendar`.)

Okay, so now we've got an instance of Calendar, let's go back to our earlier example, and find out what day of the week our trillionth millisecond falls on, and then let's add a month to that date:

```
import java.util.*;
class Dates2 {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);
    System.out.println("1st date " + d1.toString());

    Calendar c = Calendar.getInstance();
    c.setTime(d1);                          // #1

    if(c.SUNDAY == c.getFirstDayOfWeek())   // #2
      System.out.println("Sunday is the first day of the week");
    System.out.println("trillionth milli day of week is "
                       + c.get(c.DAY_OF_WEEK));    // #3

    c.add(Calendar.MONTH, 1);                      // #4
    Date d2 = c.getTime();                         // #5
    System.out.println("new date " + d2.toString() );
  }
}
```

This produces something like

```
1st date Sat Sep 08 19:46:40 MDT 2001
Sunday is the first day of the week
trillionth milli day of week is 7
new date Mon Oct 08 20:46:40 MDT 2001
```

Let's take a look at this program, focusing on the five highlighted lines:

1. We assign the Date `d1` to the Calendar instance `c`.

2. We use Calendar's SUNDAY field to determine whether, for our JVM, SUNDAY is considered to be the first day of the week. (In some locales, MONDAY is the first day of the week.) The Calendar class provides similar fields for days of the week, months, the day of the month, the day of the year, and so on.
3. We use the DAY_OF_WEEK field to find out the day of the week that the trillionth millisecond falls on.
4. So far we've used setter and getter methods that should be intuitive to figure out. Now we're going to use Calendar's add() method. This very powerful method let's you add or subtract units of time appropriate for whichever Calendar field you specify. For instance:

```
c.add(Calendar.HOUR, -4);     // subtract 4 hours from c's value
c.add(Calendar.YEAR, 2);           // add 2 years to c's value
c.add(Calendar.DAY_OF_WEEK, -2);  // subtract two days from
                                   // c's  value
```

5. Convert c's value back to an instance of Date.

The other Calendar method you should know for the exam is the roll() method. The roll() method acts like the add() method, except that when a part of a Date gets incremented or decremented, larger parts of the Date will not get incremented or decremented. Hmmm…for instance:

```
// assume c is October 8, 2001
c.roll(Calendar.MONTH, 9);      // notice the year in the output
Date d4 = c.getTime();
System.out.println("new date " + d4.toString() );
```

The output would be something like this

```
new date Fri Jul 08 19:46:40 MDT 2001
```

Notice that the year did not change, even though we added 9 months to an October date. In a similar fashion, invoking roll() with HOUR won't change the date, the month or the year.

For the exam, you won't have to memorize the Calendar class's fields. If you need them to help answer a question, they will be provided as part of the question.

The DateFormat Class

Having learned how to create dates and manipulate them, let's find out how to format them. So that we're all on the same page, here's an example of how a date can be formatted in different ways:

```
import java.text.*;
import java.util.*;
class Dates3 {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);

    DateFormat[] dfa = new DateFormat[6];
    dfa[0] = DateFormat.getInstance();
    dfa[1] = DateFormat.getDateInstance();
    dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
    dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
    dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
    dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);

    for(DateFormat df : dfa)
      System.out.println(df.format(d1));
  }
}
```

which on our JVM produces

```
9/8/01 7:46 PM
Sep 8, 2001
9/8/01
Sep 8, 2001
September 8, 2001
Saturday, September 8, 2001
```

Examining this code we see a couple of things right away. First off, it looks like `DateFormat` is another abstract class, so we can't use `new` to create instances of `DateFormat`. In this case we used two factory methods, `getInstance()` and `getDateInstance()`. Notice that `getDateInstance()` is overloaded; when we discuss locales, we'll look at the other version of `getDateInstance()` that you'll need to understand for the exam.

Next, we used static fields from the DateFormat class to customize our various instances of DateFormat. Each of these static fields represents a formatting *style*. In

this case it looks like the no-arg version of `getDateInstance()` gives us the same style as the `MEDIUM` version of the method, but that's not a hard and fast rule. (More on this when we discuss locales.) Finally, we used the `format()` method to create Strings representing the properly formatted versions of the Date we're working with.

The last method you should be familiar with is the `parse()` method. The `parse()` method takes a String formatted in the style of the DateFormat instance being used, and converts the String into a Date object. As you might imagine, this is a risky operation because the `parse()` method could easily receive a badly formatted String. Because of this, `parse()` can throw a `ParseException`. The following code creates a `Date` instance, uses `DateFormat.format()` to convert it into a String, and then uses `DateFormat.parse()` to change it back into a Date:

```
Date d1 = new Date(1000000000000L);
System.out.println("d1 = " + d1.toString());

DateFormat df = DateFormat.getDateInstance(
                                    DateFormat.SHORT);
String s = df.format(d1);
System.out.println(s);

try {
  Date d2 = df.parse(s);
  System.out.println("parsed = " + d2.toString());
} catch (ParseException pe) {
  System.out.println("parse exc"); }
```

which on our JVM produces

```
d1 = Sat Sep 08 19:46:40 MDT 2001
9/8/01
parsed = Sat Sep 08 00:00:00 MDT 2001
```

Notice that because we were using a `SHORT` style, we lost some precision when we converted the Date to a String. This loss of precision showed up when we converted back to a Date object, and it went from being 7:46 to midnight.

on the job

The API for DateFormat.parse() explains that by default, the `parse()` method is lenient when parsing dates. Our experience is that `parse()` isn't very lenient about the formatting of Strings it will successfully parse into dates; take care when you use this method!

The Locale Class

Earlier we said that a big part of why this objective exists is to test your ability to do some basic internationalization tasks. Your wait is over; the `Locale` class is your ticket to worldwide domination. Both the DateFormat class and the NumberFormat class (which we'll cover next) can use an instance of Locale to customize formatted output to be specific to a locale. You might ask how Java defines a locale? The API says a locale is "a specific geographical, political, or cultural region." The two Locale constructors you'll need to understand for the exam are

```
Locale(String language)
Locale(String language, String country)
```

The language argument represents an ISO 639 Language Code, so for instance if you want to format your dates or numbers in Walloon (the language sometimes used in southern Belgium), you'd use "`wa`" as your language string. There are over 500 ISO Language codes, including one for Klingon ("`tlh`"), although unfortunately Java doesn't yet support the Klingon locale. We thought about telling you that you'd have to memorize all these codes for the exam…but we didn't want to cause any heart attacks. So rest assured, you won't have to memorize any ISO Language codes or ISO Country codes (of which there are about 240) for the exam.

Let's get back to how you might use these codes. If you want to represent basic Italian in your application, all you need is the language code. If, on the other hand, you want to represent the Italian used in Switzerland, you'd want to indicate that the country is Switzerland (yes, the country code for Switzerland is "`CH`"), but that the language is Italian:

```
Locale locPT = new Locale("it");         // Italian
Locale locBR = new Locale("it", "CH");  // Switzerland
```

Using these two locales on a date could give us output like this:

```
sabato 1 ottobre 2005
sabato, 1. ottobre 2005
```

Now let's put this all together in some code that creates a Calendar object, sets its date, then converts it to a Date. After that we'll take that Date object and print it out using locales from around the world:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);                  // December 14, 2010
                                      // (month is 0-based
Date d2 = c.getTime();

Locale locIT = new Locale("it", "IT");  // Italy
Locale locPT = new Locale("pt");        // Portugal
Locale locBR = new Locale("pt", "BR");  // Brazil
Locale locIN = new Locale("hi", "IN");  // India
Locale locJA = new Locale("ja");        // Japan

DateFormat dfUS = DateFormat.getInstance();
System.out.println("US       " + dfUS.format(d2));

DateFormat dfUSfull = DateFormat.getDateInstance(
                                      DateFormat.FULL);
System.out.println("US full  " + dfUSfull.format(d2));

DateFormat dfIT = DateFormat.getDateInstance(
                                      DateFormat.FULL, locIT);
System.out.println("Italy    " + dfIT.format(d2));

DateFormat dfPT = DateFormat.getDateInstance(
                                      DateFormat.FULL, locPT);
System.out.println("Portugal " + dfPT.format(d2));

DateFormat dfBR = DateFormat.getDateInstance(
                                      DateFormat.FULL, locBR);
System.out.println("Brazil   " + dfBR.format(d2));

DateFormat dfIN = DateFormat.getDateInstance(
                                      DateFormat.FULL, locIN);
System.out.println("India    " + dfIN.format(d2));

DateFormat dfJA = DateFormat.getDateInstance(
                                      DateFormat.FULL, locJA);
System.out.println("Japan    " + dfJA.format(d2));
```

This, on our JVM, produces

```
US       12/14/10 3:32 PM
US full  Sunday, December 14, 2010
Italy    domenica 14 dicembre 2010
```

```
Portugal Domingo, 14 de Dezembro de 2010
Brazil   Domingo, 14 de Dezembro de 2010
India    ??????, ?? ??????, ????
Japan    2010?12?14?
```

Oops! Our machine isn't configured to support locales for India or Japan, but you can see how a single Date object can be formatted to work for many locales.

exam Watch

Remember that both DateFormat and NumberFormat objects can have their locales set only at the time of instantiation. Watch for code that attempts to change the locale of an existing instance—no such methods exist!

There are a couple more methods in Locale (`getDisplayCountry()` and `getDisplayLanguage()`) that you'll have to know for the exam. These methods let you create Strings that represent a given locale's country and language in terms of both the default locale and any other locale:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);
Date d2 = c.getTime();

Locale locBR = new Locale("pt", "BR");  // Brazil
Locale locDK = new Locale("da", "DK");  // Denmark
Locale locIT = new Locale("it", "IT");  // Italy

System.out.println("def " + locBR.getDisplayCountry());
System.out.println("loc " + locBR.getDisplayCountry(locBR));

System.out.println("def " + locDK.getDisplayLanguage());
System.out.println("loc " + locDK.getDisplayLanguage(locDK));
System.out.println("D>I " + locDK.getDisplayLanguage(locIT));
```

This, on our JVM, produces

```
def Brazil
loc Brasil
def Danish
loc dansk
D>I danese
```

Given that our JVM's locale (the default for us) is `US`, the default for the country Brazil is "`Brazil`", and the default for the Danish language is "`Danish`". In Brazil, the country is called "`Brasil`", and in Denmark the language is called "`dansk`". Finally, just for fun, we discovered that in Italy, the Danish language is called "`danese`".

The NumberFormat Class

We'll wrap up this objective by discussing the NumberFormat class. Like the DateFormat class, NumberFormat is abstract, so you'll typically use some version of either `getInstance()` or `getCurrencyInstance()` to create a NumberFormat object. Not surprisingly, you use this class to format numbers or currency values:

```
float f1 = 123.4567f;
Locale locFR = new Locale("fr");          // France
NumberFormat[] nfa = new NumberFormat[4];

nfa[0] = NumberFormat.getInstance();
nfa[1] = NumberFormat.getInstance(locFR);
nfa[2] = NumberFormat.getCurrencyInstance();
nfa[3] = NumberFormat.getCurrencyInstance(locFR);

for(NumberFormat nf : nfa)
  System.out.println(nf.format(f1));
```

This, on our JVM, produces

```
123.457
123,457
$123.46
123,46 ?
```

Don't be worried if, like us, you're not set up to display the symbols for francs, pounds, rupees, yen, baht, or drachmas. You won't be expected to

know the symbols used for currency: if you need one, it will be specified in the question. You might encounter methods other than the format method on the exam. Here's a little code that uses `getMaximumFractionDigits()`, `setMaximumFractionDigits()`, `parse()`, and `setParseIntegerOnly()`:

```
float f1 = 123.45678f;
NumberFormat nf = NumberFormat.getInstance();
System.out.print(nf.getMaximumFractionDigits() + " ");
System.out.print(nf.format(f1) + "  ");
nf.setMaximumFractionDigits(5);
System.out.println(nf.format(f1) + "  ");

try {
  System.out.println(nf.parse("1234.567"));
  nf.setParseIntegerOnly(true);
  System.out.println(nf.parse("1234.567"));
} catch (ParseException pe) {
  System.out.println("parse exc");
}
```

This, on our JVM, produces

```
3  123.457  123.45678
1234.567
1234
```

Notice that in this case, the initial number of fractional digits for the default NumberFormat is three: and that the `format()` method rounds `f1`'s value, it doesn't truncate it. After changing `nf`'s fractional digits, the entire value of `f1` is displayed. Next, notice that the `parse()` method must run in a try/catch block and that the `setParseIntegerOnly()` method takes a `boolean` and in this case, causes subsequent calls to `parse()` to return only the integer part of Strings formatted as floating-point numbers.

As we've seen, several of the classes covered in this objective are `abstract`. In addition, for all of these classes, key functionality for every instance is established at the time of creation. Table 6-3 summarizes the constructors or methods used to create instances of all the classes we've discussed in this section.

TABLE 6-3 Instance Creation for Key java.text and java.util Classes

Class	Key Instance Creation Options
util.Date	new Date(); new Date(long millisecondsSince010170);
util.Calendar	Calendar.getInstance(); Calendar.getInstance(Locale);

util.Locale	Locale.getDefault();
	new Locale(String language);
	new Locale(String language, String country);

text.DateFormat	DateFormat.getInstance(); DateFormat.getDateInstance(); DateFormat.getDateInstance(style); DateFormat.getDateInstance(style, Locale);
text.NumberFormat	NumberFormat.getInstance() NumberFormat.getInstance(Locale) NumberFormat.getNumberInstance() NumberFormat.getNumberInstance(Locale) NumberFormat.getCurrencyInstance() NumberFormat.getCurrencyInstance(Locale)

CERTIFICATION OBJECTIVE

Parsing, Tokenizing, and Formatting (Exam Objective 3.5)

*3.5 Write code that uses standard J2SE APIs in the java.util and java.util.regex packages to format or parse strings or streams. For strings, write code that uses the Pattern and Matcher classes and the String.split method. Recognize and use regular expression patterns for matching (limited to: .(dot), *(star), +(plus), ?, \d, \s, \w, [ ], ( )). The use of *, + , and ? will be limited to greedy quantifiers, and the parenthesis operator will only be used as a grouping mechanism, not for capturing content during matching. For streams, write code using the Formatter and Scanner classes and the PrintWriter.format/printf methods. Recognize and use formatting parameters (limited to: %b, %c, %d, %f, %s) in format Strings.*

We're going to start with yet another disclaimer: This small section isn't going to morph you from regex newbie to regex guru. In this section we'll cover three basic ideas:

- **Finding stuff** You've got big heaps of text to look through. Maybe you're doing some screen scraping, maybe you're reading from a file. In any case, you need easy ways to find textual needles in textual haystacks. We'll use the java.regex.Pattern, java.regex.Matcher, and java.util.Scanner classes to help us find stuff.
- **Tokenizing stuff** You've got a delimited file that you want to get useful data out of. You want to transform a piece of a text file that looks like: "1500.00,343.77,123.4" into some individual float variables. We'll show you the basics of using the `String.split()` method and the java.util.Scanner class, to tokenize your data.
- **Formatting stuff** You've got a report to create and you need to take a float variable with a value of 32500.000f and transform it into a String with a value of "$32,500.00". We'll introduce you to the java.util.Formatter class and to the `printf()` and `format()` methods.

A Search Tutorial

Whether you're looking for stuff or tokenizing stuff, a lot of the concepts are the same, so let's start with some basics. No matter what language you're using, sooner or later you'll probably be faced with the need to search through large amounts of textual data, looking for some specific stuff.

Regular expressions (regex for short) are a kind of language within a language, designed to help programmers with these searching tasks. Every language that provides regex capabilities uses one or more regex *engines*. regex engines search through textual data using instructions that are coded into *expressions*. A regex expression is like a very short program or script. When you invoke a regex engine, you'll pass it the chunk of textual data you want it to process (in Java this is usually a String or a stream), and you pass it the expression you want it to use to search through the data.

It's fair to think of regex as a language, and we will refer to it that way throughout this section. The regex language is used to create expressions, and as we work through this section, whenever we talk about expressions or expression syntax, we're talking about syntax for the regex "language." Oh, one more disclaimer…we know that you regex mavens out there can come up with better expressions than what

we're about to present. Keep in mind that for the most part we're creating these expressions using only a portion of the total regex instruction set, thanks.

Simple Searches

For our first example, we'd like to search through the following *source* String

```
abaaaba
```

for all occurrences (or *matches*) of the *expression*

```
ab
```

In all of these discussions we'll assume that our data sources use zero-based indexes, so if we apply an index to our source string we get

```
source: abaaaba
index:  0123456
```

We can see that we have two occurrences of the expression `ab`: one starting at position 0 and the second starting at position 4. If we sent the previous source data and expression to a regex engine, it would reply by telling us that it found matches at positions 0 and 4:

```
import java.util.regex.*;
class RegexSmall {
  public static void main(String [] args) {
    Pattern p = Pattern.compile("ab");      // the expression
    Matcher m = p.matcher("abaaaba");       // the source
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + " ");
    }
  }
}
```

This produces

```
0 4
```

We're not going to explain this code right now. In a few pages we're going to show you a lot more regex code, but first we want to go over some more regex syntax. Once you understand a little more regex, the code samples will make a lot more sense. Here's a more complicated example of a source and an expression:

```
source: abababa
index:  0123456
expression: aba
```

How many occurrences do we get in this case? Well, there is clearly an occurrence starting at position 0, and another starting at position 4. But how about starting at position 2? In general in the world of regex, the `aba` string that starts at position 2 will not be considered a valid occurrence. The first general regex search rule is

> In general, a regex search runs from left to right, and once a source's character has been used in a match, it cannot be reused.

So in our previous example, the first match used positions 0, 1, and 2 to match the expression. (Another common term for this is that the first three characters of the source were *consumed*.) Because the character in position 2 was consumed in the first match, it couldn't be used again. So the engine moved on, and didn't find another occurrence of `aba` until it reached position 4. This is the typical way that a regex matching engine works. However, in a few pages, we'll look at an exception to the first rule we stated above.

So we've matched a couple of exact strings, but what would we do if we wanted to find something a little more dynamic? For instance, what if we wanted to find all of the occurrences of hex numbers or phone numbers or ZIP codes?

Searches Using Metacharacters

As luck would have it, regex has a powerful mechanism for dealing with the cases we described above. At the heart of this mechanism is the idea of a *metacharacter*. As an easy example, let's say that we want to search through some source data looking for all occurrences of numeric digits. In regex, the following expression is used to look for numeric digits:

```
\d
```

If we change the previous program to apply the expression `\d` to the following source string

```
source: a12c3e456f
index:  0123456789
```

regex will tell us that it found digits at positions 1, 2, 4, 6, 7, and 8. (If you want to try this at home, you'll need to "escape" the `compile` method's "`\d`" argument by making it "`\\d`", more on this a little later.)

Regex provides a rich set of metacharacters that you can find described in the API documentation for java.util.regex.Pattern. We won't discuss them all here, but we will describe the ones you'll need for the exam:

`\d` A digit
`\s` A whitespace character
`\w` A word character (letters, digits, or "_" (underscore))

So for example, given

```
source: "a 1 56 _Z"
index:   012345678
pattern: \w
```

regex will return positions: 0, 2, 4, 5, 7, and 8. The only characters in this source that don't match the definition of a word character are the whitespaces. (Note: In this example we enclosed the source data in quotes to clearly indicate that there was no whitespace at either end.)

You can also specify sets of characters to search for using square brackets and ranges of characters to search for using square brackets and a dash:

`[abc]` Searches only for a's, b's or c's
`[a-f]` Searches only for a, b, c, d, e, or f characters

In addition, you can search across several ranges at once. The following expression is looking for occurrences of the letters `a - f` or `A - F`, it's NOT looking for an `fA` combination:

`[a-fA-F]` Searches for the first six letters of the alphabet, both cases.

So for instance

```
source: "cafeBABE"
index:   01234567
pattern: [a-cA-C]
```

returns positions: 0, 1, 4, 5, 6

on the job

In addition to the capabilities described for the exam, you can also apply the following attributes to sets and ranges within square brackets: "^" to negate the characters specified, nested brackets to create a union of sets, and "&&" to specify the intersection of sets. While these constructs are not on the exam, they are quite useful, and good examples can be found in the API for the java.util.regex.Pattern class.

Searches Using Quantifiers

Let's say that we want to create a regex pattern to search for hexadecimal literals. As a first step, let's solve the problem for one-digit hexadecimal numbers:

```
0[xX][0-9a-fA-F]
```

The preceding expression could be stated: "Find a set of characters in which the first character is a "0", the second character is either an "x" or an "X", and the third character is either a digit from "0" to "9", a letter from "a" to "f" or an uppercase letter from "A" to "F" ". Using the preceding expression, and the following data,

```
source: "12 0x 0x12 0Xf 0xg"
index:   012345678901234567
```

regex would return 6 and 11. (Note: `0x` and `0xg` are not valid hex numbers.)

As a second step, let's think about an easier problem. What if we just wanted regex to find occurrences of integers? Integers can be one or more digits long, so it would be great if we could say "one or more" in an expression. There is a set of regex constructs called quantifiers that let us specify concepts such as "one or more." In fact, the quantifier that represents "one or more" is the "+" character. We'll see the others shortly.

The other issue this raises is that when we're searching for something whose length is variable, getting only a starting position as a return value has limited value. So, in addition to returning starting positions, another bit of information that a regex engine can return is the entire match or *group* that it finds. We're going to change the way we talk about what regex returns by specifying each return on its own line, remembering that now for each return we're going to get back the starting position AND then the group:

```
source: "1 a12 234b"
pattern: \d+
```

You can read this expression as saying: "Find one or more digits in a row." This expression produces the regex output

```
0 1
3 12
6 234
```

You can read this as "At position 0 there's an integer with a value of 1, then at position 3 there's an integer with a value of 12, then at position 6 there's an integer with a value of 234." Returning now to our hexadecimal problem, the last thing we need to know is how to specify the use of a quantifier for only part of an expression. In this case we must have exactly one occurrence of `0x` or `0X` but we can have from one to many occurrences of the hex "digits" that follow. The following expression adds parentheses to limit the "+" quantifier to only the hex digits:

```
0[xX]([0-9a-fA-F])+
```

The parentheses and "+" augment the previous find-the-hex expression by saying in effect: "Once we've found our 0x or 0X, you can find from one to many occurrences of hex digits." Notice that we put the "+" quantifier at the end of the expression. It's useful to think of quantifiers as always quantifying the part of the expression that precedes them.

The other two quantifiers we're going to look at are

* Zero or more occurrences
? Zero or one occurrence

Let's say you have a text file containing a comma-delimited list of all the file names in a directory that contains several very important projects. (BTW, this isn't how we'd arrange our directories :) You want to create a list of all the files whose names start with `proj1`. You might discover .txt files, .java files, .pdf files, who knows? What kind of regex expression could we create to find these various `proj1` files? First let's take a look at what a part of this text might look like:

```
..."proj3.txt,proj1sched.pdf,proj1,proj2,proj1.java"...
```

To solve this problem we're going to use the regex ^ (carat) operator, which we mentioned earlier. The regex ^ operator isn't on the exam, but it will help us create a fairly clean solution to our problem. The ^ is the negation symbol in regex. For instance, if you want to find anything but a's, b's, or c's in a file you could say

```
[^abc]
```

So, armed with the ^ operator and the * (zero or more) quantifier we can create the following:

```
proj1([^,])*
```

If we apply this expression to just the portion of the text file we listed above, regex returns

```
10 proj1sched.pdf
25 proj1
37 proj1.java
```

The key part of this expression is the "give me zero or more characters that aren't a comma."

The last quantifier example we'll look at is the ? (zero or one) quantifier. Let's say that our job this time is to search a text file and find anything that might be a local, 7-digit phone number. We're going to say, arbitrarily, that if we find either seven digits in a row, or three digits followed by a dash or a space followed by 4 digits, that we have a candidate. Here are examples of "valid" phone numbers:

```
1234567
123 4567
123-4567
```

The key to creating this expression is to see that we need "zero or one instance of either a space or a dash" in the middle of our digits:

```
\d\d\d([-\s])?\d\d\d\d
```

The Predefined Dot

In addition to the \s, \d, and \w metacharacters that we discussed, you also have to understand the "." (dot) metacharacter. When you see this character in a regex expression, it means "any character can serve here." For instance, the following source and pattern

```
source: "ac abc a c"
pattern: a.c
```

will produce the output

```
3 abc
7 a c
```

The "." was able to match both the "b" and the " " in the source data.

Greedy Quantifiers

When you use the *, +, and ? quantifiers, you can fine tune them a bit to produce behavior that's known as "greedy," "reluctant," or "possessive." Although you need to understand only the greedy quantifier for the exam, we're also going to discuss the reluctant quantifier to serve as a basis for comparison. First the syntax:

```
? is greedy, ?? is reluctant, for zero or once
* is greedy, *? is reluctant, for zero or more
+ is greedy, +? is reluctant, for one or more
```

What happens when we have the following source and pattern?

```
source:  yyxxxyxx
pattern:  .*xx
```

First off, we're doing something a bit different here by looking for characters that prefix the static (xx) portion of the expression. We think we're saying something

like: "Find sets of characters that ends with xx". Before we tell what happens, we at least want you to consider that there are two plausible results…can you find them? Remember we said earlier that in general, regex engines worked from left to right, and consumed characters as they went. So, working from left to right, we might predict that the engine would search the first 4 characters (0-3), find xx starting in position 2, and have its first match. Then it would proceed and find the second xx starting in position 6. This would lead us to a result like this:

```
0 yyxx
4 xyxx
```

A plausible second argument is that since we asked for a set of characters that ends with xx we might get a result like this:

```
0 yyxxxyxx
```

The way to think about this is to consider the name *greedy*. In order for the second answer to be correct, the regex engine would have to look (greedily) at the *entire* source data before it could determine that there was an xx at the end. So in fact, the second result is the correct result because in the original example we used the greedy quantifier *. The result that finds two different sets can be generated by using the reluctant quantifier *?. Let's review:

```
source:  yyxxxyxx
pattern:  .*xx
```

is using the greedy quantifier * and produces

```
0 yyxxxyxx
```

If we change the pattern to

```
source:  yyxxxyxx
pattern:  .*?xx
```

we're now using the reluctant qualifier *?, and we get the following:

```
0 yyxx
4 xyxx
```

The greedy quantifier does in fact read the entire source data, and then it works backwards (from the right) until it finds the rightmost match. At that point, it includes everything from earlier in the source data up to and including the data that is part of the rightmost match.

on the job

There are a lot more aspects to regex quantifiers than we've discussed here, but we've covered more than enough for the exam. Sun has several tutorials that will help you learn more about quantfiers, and turn you into the go-to person at your job.

When Metacharacters and Strings Collide

So far we've been talking about regex from a theoretical perspective. Before we can put regex to work we have to discuss one more gotcha. When it's time to implement regex in our code, it will be quite common that our source data and/or our expressions will be stored in Strings. The problem is that metacharacters and Strings don't mix too well. For instance. let's say we just want to do a simple regex pattern that looks for digits. We might try something like

```
String pattern = "\d";    // compiler error!
```

This line of code won't compile! The compiler sees the `\` and thinks, "Ok, here comes an escape sequence, maybe it'll be a new line!" But no, next comes the `d` and the compiler says "I've never heard of the `\d` escape sequence." The way to satisfy the compiler is to add another backslash in front of the `\d`

```
String pattern = "\\d";    // a compilable metacharacter
```

The first backslash tells the compiler that whatever comes next should be taken literally, not as an escape sequence. How about the dot (.) metacharacter? If we want a dot in our expression to be used as a metacharacter, then no problem, but what if we're reading some source data that happens to use dots as delimiters? Here's another way to look at our options:

```
String p = ".";   // regex sees this as the "." metacharacter
String p = "\.";  // the compiler sees this as an illegal
                  // Java escape sequence
```

```
String p = "\\."; // the compiler is happy, and regex sees a
                  // dot, not a metacharacter
```

A similar problem can occur when you hand metacharacters to a Java program via command-line arguments. If we want to pass the `\d` metacharacter into our Java program, our JVM does the right thing if we say

```
% java DoRegex "\d"
```

But your JVM might not. If you have problems running the following examples, you might try adding a backslash (i.e. `\\d`) to your command-line metacharacters. Don't worry, you won't see any command-line metacharacters on the exam!

The Java language defines several escape sequences, including

`\n` = linefeed (which you might see on the exam)
`\b` = backspace
`\t` = tab

And others, which you can find in the Java Language Specification. Other than perhaps seeing a `\n` inside a String, you won't have to worry about Java's escape sequences on the exam.

At this point we've learned enough of the regex language to start using it in our Java programs. We'll start by looking at using regex expressions to find stuff, and then we'll move to the closely related topic of tokenizing stuff.

Locating Data via Pattern Matching

Once you know a little regex, using the java.util.regex.Pattern (Pattern) and java.util.regex.Matcher (Matcher) classes is pretty straightforward. The Pattern class is used to hold a representation of a regex expression, so that it can be used and reused by instances of the Matcher class. The Matcher class is used to invoke the regex engine with the intention of performing match operations. The following program shows Pattern and Matcher in action, and it's not a bad way for you to do your own regex experiments:

```
import java.util.regex.*;
class Regex {
  public static void main(String [] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    System.out.println("Pattern is " + m.pattern());
    while(b = m.find()) {
      System.out.println(m.start() + " " + m.group());
    }
  }
}
```

This program uses the first command-line argument (args[0]) to represent the regex expression you want to use, and it uses the second argument (args[1]) to represent the source data you want to search. Here's a test run:

```
% java Regex "\d\w" "ab4 56_7ab"
```

Produces the output

```
Pattern is \d\w
4 56
7 7a
```

(Remember, if you want this expression to be represented in a String, you'd use `\\d\\w`). Because you'll often have special characters or whitespace as part of your arguments, you'll probably want to get in the habit of always enclosing your argument in quotes. Let's take a look at this code in more detail. First off, notice that we aren't using `new` to create a Pattern; if you check the API, you'll find no constructors are listed. You'll use the overloaded, static `compile()` method (that takes `String expression`) to create an instance of Pattern. For the exam, all you'll need to know to create a `Matcher`, is to use the `Pattern.matcher()` method (that takes `String sourceData`).

The important method in this program is the `find()` method. This is the method that actually cranks up the regex engine and does some searching. The `find()` method returns `true` if it gets a match, and remembers the start position of the match. If `find()` returns `true`, you can call the `start()` method to get the starting position of the match, and you can call the `group()` method to get the string that represents the actual bit of source data that was matched.

A common reason to use regex is to perform search and replace operations. Although replace operations are not on the exam you should know that the Matcher class provides several methods that perform search and replace operations. See the `appendReplacement()`***,*** `appendTail()`***, and*** `replaceAll()` ***methods in the Matcher API for more details.***

The Matcher class allows you to look at subsets of your source data by using a concept called *regions*. In real life, regions can greatly improve performance, but you won't need to know anything about them for the exam.

Searching Using the Scanner Class Although the java.util.Scanner class is primarily intended for tokenizing data (which we'll cover next), it can also be used to find stuff, just like the Pattern and Matcher classes. While Scanner doesn't provide location information or search and replace functionality, you can use it to apply regex expressions to source data to tell you how many instances of an expression exist in a given piece of source data. The following program uses the first command-line argument as a regex expression, then asks for input using `System.in`. It outputs a message every time a match is found:

```
import java.util.*;
class ScanIn {
  public static void main(String[] args) {
    System.out.print("input: ");
    System.out.flush();
    try {
      Scanner s = new Scanner(System.in);
      String token;
      do {
        token = s.findInLine(args[0]);
        System.out.println("found " + token);
      } while (token != null);
    } catch (Exception e) { System.out.println("scan exc"); }
  }
}
```

The invocation and input

```
java ScanIn "\d\d"
input: 1b2c335f456
```

produce the following:

```
found 33
found 45
found null
```

Tokenizing

Tokenizing is the process of taking big pieces of source data, breaking them into little pieces, and storing the little pieces in variables. Probably the most common tokenizing situation is reading a delimited file in order to get the contents of the file moved into useful places like objects, arrays or collections. We'll look at two classes in the API that provide tokenizing capabilities: String (using the `split()` method) and Scanner, which has many methods that are useful for tokenizing.

Tokens and Delimiters

When we talk about tokenizing, we're talking about data that starts out composed of two things: tokens and delimiters. Tokens are the actual pieces of data, and delimiters are the expressions that *separate* the tokens from each other. When most people think of delimiters, they think of single characters, like commas or backslashes or maybe a single whitespace. These are indeed very common delimiters, but strictly speaking, delimiters can be much more dynamic. In fact, as we hinted at a few sentences ago, delimiters can be anything that qualifies as a regex expression. Let's take a single piece of source data and tokenize it using a couple of different delimiters:

```
source: "ab,cd5b,6x,z4"
```

If we say that our delimiter is a comma, then our four tokens would be

```
ab
cd5b
6x
z4
```

If we use the same source, but declare our delimiter to be `\d`, we get three tokens:

```
ab,cd
b,
x,z
```

In general, when we tokenize source data, the delimiters themselves are discarded, and all that we are left with are the tokens. So in the second example, we defined digits to be delimiters, so the 5, 6, and 4 do not appear in the final tokens.

Tokenizing with String.split()

The String class's `split()` method takes a regex expression as its argument, and returns a String array populated with the tokens produced by the split (or tokenizing) process. This is a handy way to tokenize relatively small pieces of data. The following program uses `args[0]` to hold a source string, and `args[1]` to hold the regex pattern to use as a delimiter:

```
import java.util.*;
class SplitTest {
  public static void main(String[] args) {
    String[] tokens = args[0].split(args[1]);
    System.out.println("count " + tokens.length);
    for(String s : tokens)
      System.out.println(">" + s + "<");
  }
}
```

Everything happens all at once when the `split()` method is invoked. The source string is split into pieces, and the pieces are all loaded into the `tokens` String array. All the code after that is just there to verify what the split operation generated. The following invocation

```
% java SplitTest "ab5 ccc 45 @" "\d"
```

produces

```
count 4
>ab<
> ccc <
><
> @<
```

(Note: Remember that to represent "\" in a string you may need to use the escape sequence "\\". Because of this, and depending on your OS, your second argument might have to be "\\d" or even "\\\\d".)

We put the tokens inside "> <" characters to show whitespace. Notice that every digit was used as a delimiter, and that contiguous digits created an empty token.

One drawback to using the `String.split()` method is that often you'll want to look at tokens as they are produced, and possibly quit a tokenization operation early when you've created the tokens you need. For instance, you might be searching a large file for a phone number. If the phone number occurs early in the file, you'd like to quit the tokenization process as soon as you've got your number. The Scanner class provides a rich API for doing just such on-the-fly tokenization operations.

exam Watch

Because `System.out.println()` ***is so heavily used on the exam, you might see examples of escape sequences tucked in with questions on most any topic, including regex. Remember that if you need to create a String that contains a double quote*** `"` ***or a backslash*** `\` ***you need to add an escape character first:***

```
System.out.println("\" \\");
```

This prints

```
" \
```

So, what if you need to search for periods (.) in your source data? If you just put a period in the regex expression, you get the "any character" behavior. So, what if you try `\.` ***? Now the Java compiler thinks you're trying to create an escape sequence that doesn't exist. The correct syntax is***

```
String s = "ab.cde.fg";
String[] tokens = s.split("\\.");
```

Tokenizing with Scanner

The java.util.Scanner class is the Cadillac of tokenizing. When you need to do some serious tokenizing, look no further than Scanner—this beauty has it all. In addition to the basic tokenizing capabilities provided by `String.split()`, the Scanner class offers the following features:

- Scanners can be constructed using files, streams, or Strings as a source.

- Tokenizing is performed within a loop so that you can exit the process at any point.
- Tokens can be converted to their appropriate primitive types automatically.

Let's look at a program that demonstrates several of Scanner's methods and capabilities. Scanner's default delimiter is whitespace, which this program uses. The program makes two Scanner objects: `s1` is iterated over with the more generic `next()` method, which returns every token as a String, while `s2` is analyzed with several of the specialized `nextXxx()` methods (where `Xxx` is a primitive type):

```
import java.util.Scanner;
class ScanNext {
  public static void main(String [] args) {
    boolean b2, b;
    int i;
    String s, hits = " ";
    Scanner s1 = new Scanner(args[0]);
    Scanner s2 = new Scanner(args[0]);
    while(b = s1.hasNext()) {
      s = s1.next();  hits += "s";
    }
    while(b = s2.hasNext()) {
      if (s2.hasNextInt()) {
        i = s2.nextInt();  hits += "i";
      } else if (s2.hasNextBoolean()) {
        b2 = s2.nextBoolean();  hits += "b";
      } else {
        s2.next();  hits += "s2";
      }
    }
    System.out.println("hits " + hits);
  }
}
```

If this program is invoked with

```
% java ScanNext "1 true 34 hi"
```

it produces

```
hits  ssssibis2
```

Of course we're not doing anything with the tokens once we've got them, but you can see that `s2`'s tokens are converted to their respective primitives. A key point here is that the methods named `hasNextXxx()` test the value of the next token but do not actually get the token, nor do they move to the next token in the source data. The `nextXxx()` methods all perform two functions: they get the next token, and then they move to the next token.

The Scanner class has `nextXxx()` (for instance `nextLong()`) and `hasNextXxx()` (for instance `hasNextDouble()`) methods for every primitive type except `char`. In addition, the Scanner class has a `useDelimiter()` method that allows you to set the delimiter to be any valid regex expression.

Formatting with printf() and format()

What fun would accounts receivable reports be if the decimal points didn't line up? Where would you be if you couldn't put negative numbers inside of parentheses? Burning questions like these caused the exam creation team to include formatting as a part of the Java 5 exam. The `format()` and `printf()` methods were added to java.io.PrintStream in Java 5. These two methods behave exactly the same way, so anything we say about one of these methods applies to both of them. (The rumor is that Sun added `printf()` just to make old C programmers happy.)

Behind the scenes, the `format()` method uses the `java.util.Formatter` class to do the heavy formatting work. You can use the `Formatter` class directly if you choose, but for the exam all you have to know is the basic syntax of the arguments you pass to the `format()` method. The documentation for these formatting arguments can be found in the Formatter API. We're going to take the "nickel tour" of the formatting String syntax, which will be more than enough to allow you do to a lot of basic formatting work, AND ace all the formatting questions on the exam.

Let's start by paraphrasing the API documentation for format strings (for more complete, way-past-what-you-need-for-the-exam coverage, check out the java.util.Formatter API):

```
printf("format string", argument(s));
```

The format string can contain both normal string literal information that isn't associated with any arguments, and argument-specific formatting data. The clue to determining whether you're looking at formatting data, is that formatting data will always start with a percent sign (`%`). Let's look at an example, and don't panic, we'll cover everything that comes after the `%` next:

```
System.out.printf("%2$d  +  %1$d", 123, 456);
```

This produces

```
456 + 123
```

Let's look at what just happened. Inside the double quotes there is a format string, then a +, and then a second format string. Notice that we mixed literals in with the format strings. Now let's dive in a little deeper and look at the construction of format strings:

```
%[arg_index$][flags][width][.precision]conversion char
```

The values within `[ ]` are optional. In other words, the only required elements of a format string are the `%` and a conversion character. In the example above the only optional values we used were for argument indexing. The `2$` represents the second argument, and the `1$` represents the first argument. (Notice that there's no problem switching the order of arguments.) The `d` after the arguments is a conversion character (more or less the type of the argument). Here's a rundown of the format string elements you'll need to know for the exam:

arg_index An integer followed directly by a `$`, this indicates which argument should be printed in this position.

flags While many flags are available, for the exam you'll need to know:

- "-" Left justify this argument
- "+" Include a sign (+ or -) with this argument
- "0" Pad this argument with zeroes
- "," Use locale-specific grouping separators (i.e., the comma in 123,456)
- "(" Enclose negative numbers in parentheses

width This value indicates the minimum number of characters to print. (If you want nice even columns, you'll use this value extensively.)

precision For the exam you'll only need this when formatting a floating-point number, and in the case of floating point numbers, precision indicates the number of digits to print after the decimal point.

conversion The type of argument you'll be formatting. You'll need to know:

- `b` boolean
- `c` char
- `d` integer
- `f` floating point
- `s` string

Let's see some of these formatting strings in action:

```
int i1 = -123;
int i2 = 12345;
System.out.printf(">%1$(7d< \n", i1);
System.out.printf(">%0,7d< \n", i2);
System.out.format(">%+-7d< \n", i2);
System.out.printf(">%2$b + %1$5d< \n", i1, false);
```

This produces:

```
>  (123)<
>012,345<
>+12345 <
>false +  -123<
```

(We added the > and < literals to help show how minimum widths, and zero padding and alignments work.) Finally, it's important to remember that if you have a mismatch between the type specified in your conversion character and your argument, you'll get a runtime exception:

```
System.out.format("%d", 12.3);
```

This produces something like

```
Exception in thread "main" java.util.IllegalFormatConversionEx-
ception: d != java.lang.Double
```

CERTIFICATION SUMMARY

Strings The most important thing to remember about Strings is that String objects are immutable, but references to Strings are not! You can make a new String by using an existing String as a starting point, but if you don't assign a reference variable to the new String it will be lost to your program—you will have no way to access your new String. Review the important methods in the String class.

The StringBuilder class was added in Java 5. It has exactly the same methods as the old StringBuffer class, except StringBuilder's methods aren't thread-safe. Because StringBuilder's methods are not thread safe, they tend to run faster than StringBuffer methods, so choose StringBuilder whenever threading is not an issue. Both StringBuffer and StringBuilder objects can have their value changed over and over without having to create new objects. If you're doing a lot of string manipulation, these objects will be more efficient than immutable String objects, which are, more or less, "use once, remain in memory forever." Remember, these methods ALWAYS change the invoking object's value, even with no explicit assignment.

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. It will be rare that you'll 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 BufferWriters a lot less.

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 serial 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 `readMethod()`. 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.

Dates, Numbers, and Currency Remember that the Sun objective is a bit misleading, and that you'll have to understand the basics of five related classes: java.util.Date, java.util.Calendar, java.util.Locale, java.text.DateFormat, and java.text.NumberFormat. A Date is the number of milliseconds since Jan. 1, 1970, stored in a `long`. Most of Date's methods have been deprecated, so use the Calendar class for your date-manipulation tasks. Remember that in order to create instances of Calendar, DateFormat, and NumberFormat, you have to use static factory methods like `getInstance()`. The Locale class is used with DateFormat and NumberFormat to generate a variety of output styles that are language and/or country specific.

Parsing, Tokenizing, and Formatting To find specific pieces of data in large data sources, Java provides several mechanisms that use the concepts of regular expressions (regex). regex expressions can be used with the java.util.regex package's Pattern and Matcher classes, as well as with java.util.Scanner and with the `String.split()` method. When creating regex patterns, you can use literal characters for matching or you can use metacharacters, that allow you to match on concepts like "find digits" or "find whitespace." regex provides *quantifiers* that allow you to say things like "find one or more of these things in a row." You won't have to understand the Matcher methods that facilitate replacing strings in data.

Tokenizing is splitting delimited data into pieces. Delimiters are usually as simple as a comma, but they can be as complex as any other regex pattern. The java.util.Scanner class provides full tokenizing capabilities using regex, and allows you to tokenize in a loop so that you can stop the tokenizing process at any point. `String.split()` allows full regex patterns for tokenizing, but tokenizing is done in one step, so that large data sources can take a long time to process.

Formatting data for output can be handled by using the Formatter class, or more commonly by using the new PrintStream methods `format()` and `printf()`. Remember `format()` and `printf()` behave identically. To use these methods, you create a format string that is associated with every piece of data you want to format. You need to understand the subset of format string conventions we covered in the chapter, and you need to remember that if your format string specifies a conversion character that doesn't match your data type, an exception will be thrown.

TWO-MINUTE DRILL

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

Using String, StringBuffer, and StringBuilder (Objective 3.1)

- ❑ String objects are immutable, and String reference variables are not.
- ❑ If you create a new String without assigning it, it will be lost to your program.
- ❑ If you redirect a String reference to a new String, the old String can be lost.
- ❑ String methods use zero-based indexes, except for the second argument of `substring()`.
- ❑ The String class is `final`—its methods can't be overridden.
- ❑ When the JVM finds a String literal, it is added to the String literal pool.
- ❑ Strings have a method: `length()`, arrays have an attribute named `length`.
- ❑ The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
- ❑ StringBuilder methods should run faster than StringBuffer methods.
- ❑ All of the following bullets apply to both StringBuffer and StringBuilder:
  - ❑ They are mutable—they can change without creating a new object.
  - ❑ StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
  - ❑ StringBuffer `equals()` is not overridden; it doesn't compare values.
- ❑ Remember that chained methods are evaluated from left to right.
- ❑ String methods to remember: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- ❑ Stringbuffer methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

File I/O (Objective 3.2)

- ❑ The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, and PrintWriter.
- ❑ 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.
- ❑ 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, to get access to higher-level (more convenient) methods.
- ❑ It's very common to "wrap" a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
- ❑ PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
- ❑ Java 5 PrintWriters have new `append()`, `format()`, and `printf()` methods.

Serialization (Objective 3.3)

- ❑ 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 the Serializable interface 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 thought 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 run.
- ❑ DataInputStream and DataOutputStream aren't actually on the exam, in spite of what the Sun objectives say.

Dates, Numbers, and Currency (Objective 3.4)

- The classes you need to understand are java.util.Date, java.util.Calendar, java.text.DateFormat, java.text.NumberFormat, and java.util.Locale.
- Most of the Date class's methods have been deprecated.
- A Date is stored as a `long`, the number of milliseconds since January 1, 1970.
- Date objects are go-betweens the Calendar and Locale classes.
- The Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.
- Create Calendar instances using static factory methods (`getInstance()`).
- The Calendar methods you should understand are `add()`, which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and `roll()`, which works like `add()` but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
- DateFormat instances are created using static factory methods (`getInstance()` and `getDateInstance()`).
- There are several format "styles" available in the DateFormat class.
- DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
- The `DateFormat.format()` method is used to create Strings containing properly formatted dates.
- The Locale class is used in conjunction with DateFormat and NumberFormat.
- Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.
- For the exam you should understand creating Locales using language, or a combination of language and country.

Parsing, Tokenizing, and Formatting (Objective 3.5)

- regex is short for regular expressions, which are the patterns used to search for data within large data sources.
- regex is a sub-language that exists in Java and other languages (such as Perl).
- regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".

- ❑ Study the `\d`, `\s`, `\w`, and . metacharacters
- ❑ regex provides for *quantifiers* which allow you to specify concepts like: "look for one or more digits in a row."
- ❑ Study the `?`, `*`, and `+` greedy quantifiers.
- ❑ Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance `String s = "\\d";`
- ❑ The Pattern and Matcher classes have Java's most powerful regex capabilities.
- ❑ You should understand the Pattern `compile()` method and the Matcher `matcher()`, `pattern()`, `find()`, `start()`, and `group()` methods.
- ❑ You WON'T need to understand Matcher's replacement-oriented methods.
- ❑ You can use java.util.Scanner to do simple regex searches, but it is primarily intended for tokenizing.
- ❑ Tokenizing is the process of splitting delimited data into small pieces.
- ❑ In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- ❑ Tokenizing can be done with the Scanner class, or with `String.split()`.
- ❑ Delimiters are single characters like commas, or complex regex expressions.
- ❑ The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.
- ❑ The Scanner class allows you to tokenize Strings or streams or files.
- ❑ The `String.split()` method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
- ❑ New to Java 5 are two methods used to format data for output. These methods are `format()` and `printf()`. These methods are found in the PrintStream class, an instance of which is the `out` in `System.out`.
- ❑ The `format()` and `printf()` methods have identical functionality.
- ❑ Formatting data with `printf()` (or `format()`) is accomplished using *formatting strings* that are associated with primitive or string arguments.
- ❑ The `format()` method allows you to mix literals in with your format strings.
- ❑ The format string values you should know are
  - ❑ Flags: `-`, `+`, `0`, `","` , and `(`
  - ❑ Conversions: `b`, `c`, `d`, `f`, and `s`
- ❑ If your conversion character doesn't match your argument type, an exception will be thrown.

SELF TEST

1. Given:

```
import java.util.regex.*;
class Regex2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + m.group());
    }
  }
}
```

And the command line:

```
java Regex2 "\d*" ab34ef
```

What is the result?

A. `234`

B. `334`

C. `2334`

D. `0123456`

E. `01234456`

F. `12334567`

G. Compilation fails.

2. 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.

3. Given that `bw` is a reference to a valid `BufferedWriter`

And the snippet:

```
15.  BufferedWriter b1 = new BufferedWriter(new File("f"));
16.  BufferedWriter b2 = new BufferedWriter(new FileWriter("f1"));
17.  BufferedWriter b3 = new BufferedWriter(new PrintWriter("f2"));
18.  BufferedWriter b4 = new BufferedWriter(new BufferedWriter(bw));
```

What is the result?

A. Compilation succeeds.

B. Compilation fails due only to an error on line 15.

C. Compilation fails due only to an error on line 16.

D. Compilation fails due only to an error on line 17.

E. Compilation fails due only to an error on line 18.

F. Compilation fails due to errors on multiple lines.

4. Given:

```
class TKO {
  public static void main(String[] args) {
    String s = "-";
    Integer x = 343;
    long L343 = 343L;
    if(x.equals(L343))  s += ".e1 ";
    if(x.equals(343))  s += ".e2 ";
    Short s1 = (short)((new Short((short)343)) / (new Short((short)49)));
    if(s1 == 7)        s += "=s ";
    if(s1 < new Integer(7+1))  s += "fly ";
    System.out.println(s);
  }
}
```

Which of the following will be included in the output String `s`? (Choose all that apply.)

A. `.e1`

B. `.e2`

C. `=s`

D. `fly`

E. None of the above.

F. Compilation fails.

G. An exception is thrown at runtime.

5. Given:

```
1. import java.text.*;
2. class DateOne {
3.   public static void main(String[] args) {
4.     Date d = new Date(1123631685981L);
5.     DateFormat df = new DateFormat();
6.     System.out.println(df.format(d));
7.   }
8. }
```

And given that 1123631685981L is the number of milliseconds between Jan. 1, 1970, and sometime on Aug. 9, 2005, what is the result? (Note: the time of day in option **A** may vary.)

A. 8/9/05 5:54 PM

B. `1123631685981L`

C. An exception is thrown at runtime.

D. Compilation fails due to a single error in the code.

E. Compilation fails due to multiple errors in the code.

6. 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.

7. Using the fewest **fragments** possible (and filling the fewest slots possible), complete the **code** below 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
```

8. Which are true? (Choose all that apply.)

A. The `DateFormat.getDate()` is used to convert a String to a Date instance.

B. Both `DateFormat` and `NumberFormat` objects can be constructed to be Locale specific.

C. Both Currency and `NumberFormat` objects must be constructed using static methods.

D. If a `NumberFormat` instance's Locale is to be different than the current Locale, it must be specified at creation time.

E. A single instance of `NumberFormat` can be used to create Number objects from Strings and to create formatted numbers from numbers.

9. Which will compile and run without exception? (Choose all that apply.)

A. `System.out.format("%b", 123);`

B. `System.out.format("%c", "x");`

C. `System.out.printf("%d", 123);`

D. `System.out.printf("%f", 123);`

E. `System.out.printf("%d", 123.45);`

F. `System.out.printf("%f", 123.45);`

G. `System.out.format("%s", new Long("123"));`

10. Which about the three java.lang classes String, StringBuilder, and StringBuffer are true? (Choose all that apply.)

A. All three classes have a `length()` method.

B. Objects of type StringBuffer are thread-safe.

C. All three classes have overloaded `append()` methods.

D. The "+" is an overloaded operator for all three classes.

E. According to the API, StringBuffer will be faster than StringBuilder under most implementations.

F. The value of an instance of any of these three types can be modified through various methods in the API.

11. Given that 1119280000000L is roughly the number of milliseconds from Jan. 1, 1970, to June 20, 2005, and that you want to print that date in German, using the `LONG` style such that "`June`" will be displayed as "`Juni`", complete the code using the fragments below. Note: you can use each fragment either zero or one times, and you might not need to fill all of the slots.

Code:

```
import java._____________

import java._____________

class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);

    DateFormat df = _________________________

                    ________________ , ________________ );

    System.out.println(________________
  }
}
```

Fragments:

```
io.*;      new DateFormat(                Locale.LONG
nio.*;     DateFormat.getInstance(        Locale.GERMANY
util.*;    DateFormat.getDateInstance(    DateFormat.LONG
text.*;    util.regex;                    DateFormat.GERMANY
date.*;    df.format(d));                 d.format(df));
```

12. 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 "dir1", 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() + " ");
```

13. Given:

```
class Polish {
  public static void main(String[] args) {
    int x = 4;
    StringBuffer sb = new StringBuffer("..fedcba");
    sb.delete(3,6);
    sb.insert(3, "az");
    if(sb.length() > 6)  x = sb.indexOf("b");
    sb.delete((x-3), (x-2));
    System.out.println(sb);
} }
```

What is the result?

A. `.faza`

B. `.fzba`

C. `..azba`

D. `.fazba`

E. `..fezba`

F. Compilation fails.

G. An exception is thrown at runtime.

14. Given:

```
1. import java.util.*;
2. class Brain {
3.   public static void main(String[] args) {

4.     // insert code block here

5.   }
6. }
```

Which, inserted independently at line 4, compile and produce the output "`123 82`"? (Choose all that apply.)

A.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");
    while(sc.hasNextInt())  System.out.print(sc.nextInt() + " ");
```

B.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").
                              useDelimiter(" ");
    while(sc.hasNextInt())  System.out.print(sc.nextInt() + " ");
```

C.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");
    while(sc.hasNext()) {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      else sc.next(); }
```

D.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").
                      useDelimiter(" ");
    while(sc.hasNext()) {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      else sc.next(); }
```

E.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");
    do {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      } while ( sc.hasNext() );
```

F.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").
                      useDelimiter(" ");
    do {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      } while ( sc.hasNext() );
```

15. 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 override the `readObject()` method in `SpecialSerial`.

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

SELF TEST ANSWERS

1. Given:

```
import java.util.regex.*;
class Regex2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + m.group());
    }
  }
}
```

And the command line:

```
java Regex2 "\d*" ab34ef
```

What is the result?

A. `234`

B. `334`

C. `2334`

D. `0123456`

E. `01234456`

F. `12334567`

G. Compilation fails.

Answer:

☑ **E** is correct. The `\d` is looking for digits. The `*` is a quantifier that looks for `0` to many occurrences of the pattern that precedes it. Because we specified `*`, the `group()` method returns empty Strings until consecutive digits are found, so the only time `group()` returns a value is when it returns 34 when the matcher finds digits starting in position 2. The `start()` method returns the starting position of the previous match because, again, we said find `0` to many occurrences.

☒ **A, B, C, D, E, F,** and **G** are incorrect based on the above.
(Objective 3.5)

2. 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.

Answer:

☑ **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.
(Objective 3.3)

3. Given:

`bw` is a reference to a valid `BufferedWriter`

And the snippet:

```
15.  BufferedWriter b1 = new BufferedWriter(new File("f"));
16.  BufferedWriter b2 = new BufferedWriter(new FileWriter("f1"));
17.  BufferedWriter b3 = new BufferedWriter(new PrintWriter("f2"));
18.  BufferedWriter b4 = new BufferedWriter(new BufferedWriter(bw));
```

What is the result?

A. Compilation succeeds.

B. Compilation fails due only to an error on line 15.

C. Compilation fails due only to an error on line 16.

D. Compilation fails due only to an error on line 17.

E. Compilation fails due only to an error on line 18.

F. Compilation fails due to errors on multiple lines.

Answer:

☑ **B** is correct. `BufferedWriter`s can be constructed only by wrapping a Writer. Lines 16, 17, and 18 are correct because `BufferedWriter`, `FileWriter`, and `PrintWriter` all extend Writer. (Note: `BufferedWriter` is a decorator class. Decorator classes are used extensively in the java.io package to allow you to extend the functionality of other classes.)

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objective 3.2)

4. Given:

```
class TKO {
  public static void main(String[] args) {
    String s = "-";
    Integer x = 343;
    long L343 = 343L;
    if(x.equals(L343))  s += ".e1 ";
    if(x.equals(343))  s += ".e2 ";
    Short s1 = (short)((new Short((short)343)) / (new Short((short)49)));
    if(s1 == 7)        s += "=s ";
    if(s1 < new Integer(7+1))  s += "fly ";
    System.out.println(s);
} }
```

Which of the following will be included in the output String `s`? (Choose all that apply.)

A. `.e1`
B. `.e2`
C. `=s`
D. `fly`
E. None of the above.
F. Compilation fails.
G. An exception is thrown at runtime.

Answer:

☑ **B, C,** and **D** are correct. Remember, that the `equals()` method for the integer wrappers will only return `true` if the two primitive types and the two values are equal. With **C**, it's okay to unbox and use `==`. For **D**, it's okay to create a wrapper object with an expression, and unbox it for comparison with a primitive.

☒ **A, E, F,** and **G** are incorrect based on the above. (Remember that **A** is using the `equals()` method to try to compare two different types.) (Objective 3.1)

5. Given:

```
1. import java.text.*;
2. class DateOne {
3.   public static void main(String[] args) {
4.     Date d = new Date(1123631685981L);
5.     DateFormat df = new DateFormat();
6.     System.out.println(df.format(d));
7.   }
8. }
```

And given that 1123631685981L is the number of milliseconds between Jan. 1, 1970, and sometime on Aug. 9, 2005, what is the result? (Note: the time of day in option **A** may vary.)

A. 8/9/05 5:54 PM
B. `1123631685981L`
C. An exception is thrown at runtime.
D. Compilation fails due to a single error in the code.
E. Compilation fails due to multiple errors in the code.

Answer:

☑ **E** is correct. The Date class is located in the `java.util` package so it needs an `import`, and `DateFormat` objects must be created using a static method such as `DateFormat.getInstance()` or `DateFormat.getDateInstance()`.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 3.4)

6. 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.

Answer:

☑ **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. (Objective 3.3)

7. Using the fewest fragments possible (and filling the fewest slots possible), complete the code below 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
```

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. (Objective 3.2)

8. Which are true? (Choose all that apply.)

A. The `DateFormat.getDate()` is used to convert a String to a Date instance.

B. Both `DateFormat` and `NumberFormat` objects can be constructed to be Locale specific.

C. Both Currency and `NumberFormat` objects must be constructed using static methods.

D. If a `NumberFormat` instance's Locale is to be different than the current Locale, it must be specified at creation time.

E. A single instance of `NumberFormat` can be used to create Number objects from Strings and to create formatted numbers from numbers.

Answer:

☑ **B, C, D,** and **E** are correct.

☒ **A** is incorrect, `DateFormat.parse()` is used to convert a String to a Date. (Objective 3.4)

9. Which will compile and run without exception? (Choose all that apply.)

A. `System.out.format("%b", 123);`

B. `System.out.format("%c", "x");`

C. `System.out.printf("%d", 123);`

D. `System.out.printf("%f", 123);`

E. `System.out.printf("%d", 123.45);`

F. `System.out.printf("%f", 123.45);`

G. `System.out.format("%s", new Long("123"));`

Answer:

☑ **A, C, F,** and **G** are correct. The `%b` (boolean) conversion character returns `true` for any non-`null` or non-boolean argument.

☒ **B** is incorrect, the `%c` (character) conversion character expects a character, not a String. **D** is incorrect, the `%f` (floating-point) conversion character won't automatically promote an integer type. **E** is incorrect, the `%d` (integral) conversion character won't take a floating-point number. (Note: The `format()` and `printf()` methods behave identically.) (Objective 3.5)

10. Which about the three java.lang classes String, StringBuilder, and StringBuffer are true? (Choose all that apply.)

A. All three classes have a `length()` method.

B. Objects of type StringBuffer are thread-safe.

C. All three classes have overloaded `append()` methods.

D. The "+" is an overloaded operator for all three classes.

E. According to the API, StringBuffer will be faster than StringBuilder under most implementations.

F. The value of an instance of any of these three types can be modified through various methods in the API.

Answer:

☑ **A** and **B** are correct.

☒ **C** is incorrect because String does not have an "`append`" method. **D** is incorrect because only String objects can be operated on using the overloaded "+" operator. **E** is backwards, `StringBuilder` is typically faster because it's not thread-safe. **F** is incorrect because String objects are immutable. A String reference can be altered to refer to a different String object, but the objects themselves are immutable. (Objective 3.1)

11. Given that 1119280000000L is roughly the number of milliseconds from Jan. 1, 1970, to June 20, 2005, and that you want to print that date in German, using the `LONG` style such that "`June`" will be displayed as "`Juni`", complete the code using the fragments below. Note: you can use each fragment either zero or one times, and you might not need to fill all of the slots.

Code:

```
import java._____________

import java._____________

class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);

    DateFormat df = _________________________

                    _________________ , ___________________ );

    System.out.println(_________________
  }
}
```

Fragments:

```
io.*;      new DateFormat(                  Locale.LONG
nio.*;     DateFormat.getInstance(          Locale.GERMANY
util.*;    DateFormat.getDateInstance(      DateFormat.LONG
text.*;    util.regex;                      DateFormat.GERMANY
date.*;    df.format(d));                   d.format(df));
```

Answer:

```
import java.util.*;
import java.text.*;
class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);
  DateFormat df = DateFormat.getDateInstance(
                    DateFormat.LONG, Locale.GERMANY);
    System.out.println(df.format(d));
  }
}
```

Notes: Remember that you must build `DateFormat` objects using static methods. Also remember that you must specify a Locale for a `DateFormat` object at the time of instantiation. The `getInstance()` method does not take a Locale. (Objective 3.4)

12. 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 "`dir1`", 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() + " ");
```

Answer:

☑ **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.
(Objective 3.2)

13. Given:

```
class Polish {
  public static void main(String[] args) {
    int x = 4;
    StringBuffer sb = new StringBuffer("..fedcba");
    sb.delete(3,6);
    sb.insert(3, "az");
```

```
        if(sb.length() > 6)  x = sb.indexOf("b");
        sb.delete((x-3), (x-2));
        System.out.println(sb);
      }
    }
```

What is the result?

A. `.faza`

B. `.fzba`

C. `..azba`

D. `.fazba`

E. `..fezba`

F. Compilation fails.

G. An exception is thrown at runtime.

Answer:

☑ **C** is correct. Remember that `StringBuffer` methods use zero-based indexes, and that ending indexes are typically exclusive.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 3.1)

14. Given:

```
1. import java.util.*;
2. class Brain {
3.   public static void main(String[] args) {

4.     // insert code block here

5.   }
6. }
```

Which, inserted independently at line 4, compile and produce the output "123 82"? (Choose all that apply.)

A.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");
    while(sc.hasNextInt())  System.out.print(sc.nextInt() + " ");
```

B.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").
                              useDelimiter(" ");
    while(sc.hasNextInt())  System.out.print(sc.nextInt() + " ");
```

C.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");
    while(sc.hasNext()) {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      else sc.next(); }
```

D.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").
                      useDelimiter(" ");
    while(sc.hasNext()) {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      else sc.next(); }
```

E.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");
    do {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      } while ( sc.hasNext() );
```

F.
```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").
                      useDelimiter(" ");
    do {
      if(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");
      } while ( sc.hasNext() );
```

Answer:

☑ **C** and **D** are correct. Whitespace is the default delimiter, and the while loop advances through the String using `nextInt()` or `next()`.

☒ **A** and **B** are incorrect because the while loop won't progress past the first non-`int`. **E** and **F** are incorrect. The `do` loop will loop endlessly once the first non-`int` is found because `hasNext()` does not advance through data.
(Objective 3.5)

15. 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 override the `readObject()` method in `SpecialSerial`.

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

Answer:

☑ **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 override the `defaultReadObject()` method, you call it from within the overridden `readObject()` method, along with any custom read operations your class needs. (Objective 3.3)