# 6

# Java.lang—The Math Class, Strings, and Wrappers

## CERTIFICATION OBJECTIVES

- Using the java.lang.String Class
- Using the java.lang.Math Class
- Using Wrapper Classes
- Using the `equals()` Method with Strings and Wrappers and Objects

✓ Two-Minute Drill

Q&A Self Test

This chapter focuses on the aspects of the java.lang package that you'll need to understand for the exam. The java.lang package contains many of the most fundamental and often-used classes in the Java API. The exam will test your knowledge of String and StringBuffer basics, including the infamous immutability of String objects, and how the more common String and StringBuffer methods work. You will be tested on many of the basic methods included in the Math class (extremely interesting), and you will need to know all about wrappers—those methods that allow you to encapsulate your favorite primitives into objects, so that you can do object-like stuff with them (like put them in collections). Finally, we'll reveal more than you've ever wanted to know about how the `equals()` method and `==` operator work when dealing with String objects and wrappers.

As always, our focus will be on the knowledge you'll really need to pass the exam. Undoubtedly some very wonderful methods will be overlooked in our tour of java.lang, but we're dedicated to helping you pass this test.

**CERTIFICATION OBJECTIVE**

# Using the String Class (Exam Objective 8.2)

*Describe the significance of the immutability of String objects.*

This section covers the String and StringBuffer classes. The key concepts we'll cover will help you understand that once a String object is created, it can never be changed—so what *is* happening when a String object *seems* to be changing? We'll find out. We'll also cover the differences between the String and StringBuffer classes and when to use which.

## Strings Are Immutable Objects

Let's start with a little background information about strings. Strictly speaking you may not need this information for the test, but a little context will help you learn what you *do* have to know. 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 the reference variable *s* to it. 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? The certification objective states: "describe the significance of the immutability of String objects." Ah-ha! 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 also 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 String 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.

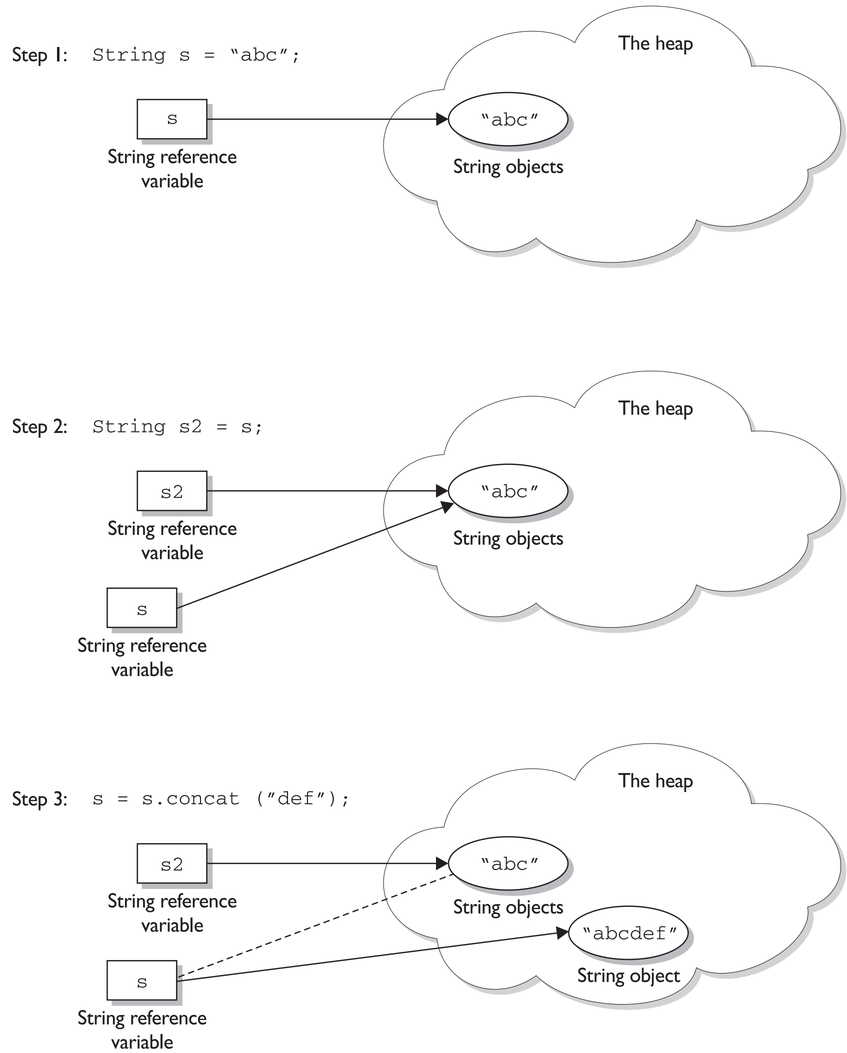
To review our first example:

```
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

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

**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 will be `x = Java`.

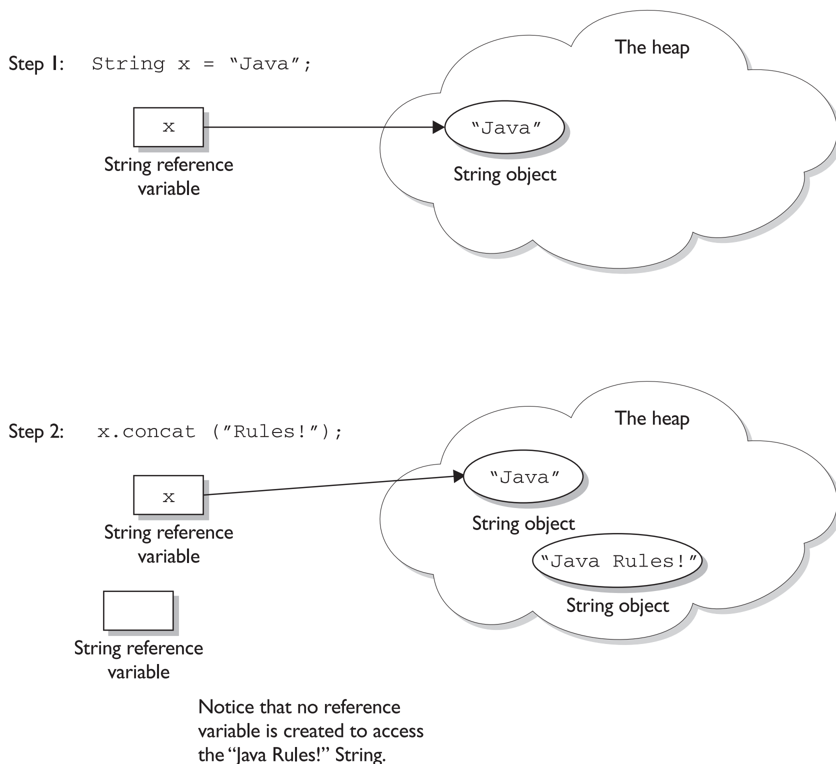
The first line is straightforward: create a new String object, give it the value `"Java"`, and refer *x* to it. What happens next? The VM creates a second String object with the value `"Java Rules!"` but nothing refers to it!!! The second String object is instantly lost; no one can ever get to it. The reference variable *x* still refers to the original String with the value `"Java"`. Figure 6-2 shows creating a String object without assigning to a reference.

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
```

**FIGURE 6-2**

A String object is abandoned upon creation



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 RULES!", 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 x to the new String
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.

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 =
```

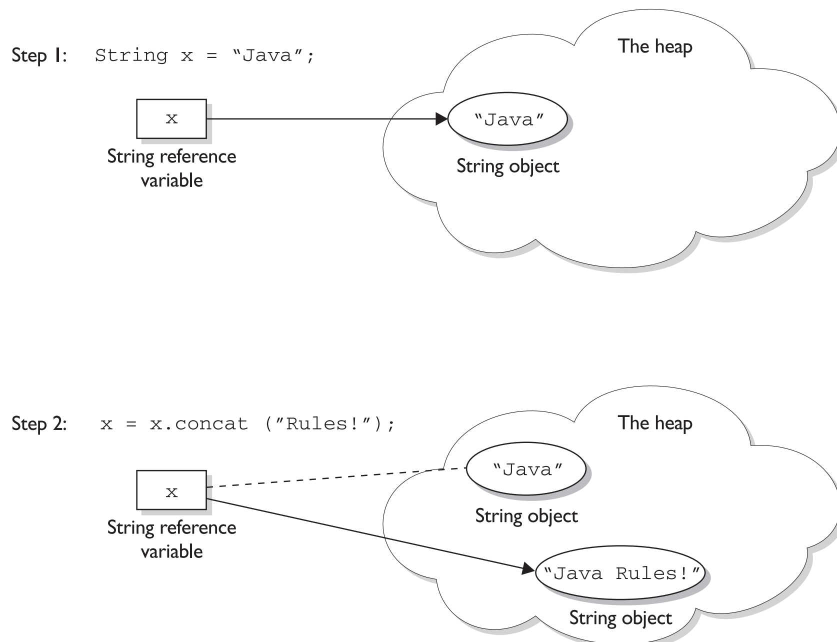
```
x.toLowerCase();                  //    create a new String, assigned to x
System.out.println("x = " + x);      //    the assignment causes the output:
                                     //    x = java rules!
```

The previous 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).

**FIGURE 6-3**

An old String object being abandoned



```
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 that String literals occupy large amounts of a program's memory, and that 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:

```
1 – 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.

```
2 – 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 (nonpool) 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.

**`public char charAt(int index)`**

This method returns the character located at the String's specified index. Remember that 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"
```

```
1. String x = "Atlantic";
2. x += " ocean"
3. 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"));   // returns "true"

System.out.println( x.equalsIgnoreCase("tixe"));   // returns "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"
```

***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() );
```

**`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 *sub*string) 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 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 $8^{th}$ 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 Class

The StringBuffer class 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. On the other hand, objects of type StringBuffer can be modified over and over again without leaving behind a great effluence of discarded String objects.

***A common use for StringBuffers 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.***

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,

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

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"*

## Important Methods in the StringBuffer Class

The following method returns a StringBuffer 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, *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"
```

or

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

```
public synchronized StringBuffer insert(int offset, String s)
```

This method returns a StringBuffer object and updates the value of the StringBuffer object that invoked the method call. In both cases, the String passed in to the second argument is inserted into the original StringBuffer 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*, etc.), but the String argument is the one you're most likely o see:

```
StringBuffer sb = new StringBuffer("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 sb = new StringBuffer("A man a plan a canal Panama");
System.out.println( sb );        // output is "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. If you take only one thing away from this section, it's that *unlike Strings, StringBuffer objects can be changed.*

exam Watch

***Many of the exam questions covering this chapter's topics use a tricky bit of Java syntax known as*** *chained methods.* ***A statement with chained methods has the 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 "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

# Using the Math Class (Exam Objective 8.1)

*Write code using the following methods of the java.lang.Math class: abs, ceil, floor, max, min, random, round, sin, cos, tan, sqrt.*

The java.lang package defines classes that are fundamental to the Java language. For this reason, all classes in the java.lang package are imported automatically, so there is no reason to write an `import` statement for them. The package defines object wrappers for all primitive types. The class names are Boolean, Byte, Character, Double, Float, Integer, Long, Short, and Void as well as Object, the class from which all other Java classes inherit.

The java.lang package also contains the Math class, which is used to perform basic mathematical operations. The Math class defines approximations for the mathematical constants *pi* and *e*. Their signatures are as follows:

```
public final static double Math.PI
public final static double Math.E
```

Because all methods of the Math class are defined as `static`, you don't need to create an instance to use them. In fact, it's not *possible* to create an instance of the Math class because the constructor is `private`. You can't extend the Math class either, because it's marked `final`.

## Methods of the java.lang.Math Class

The methods of the Math class are static and are accessed like any static method—through the class name. For these method calls the general form is

```
result = Math.aStaticMathMethod();
```

The following sections describe the Math methods and include examples of how to use them.

### abs()

The `abs()` method returns the absolute value of the argument—for example,

```
x = Math.abs(99);      // output is 99
x = Math.abs(-99)      // output is 99
```

The method is overloaded to take an *int*, a *long*, a *float*, or a *double* argument. In all but two cases, the returned value is non-negative. The signatures of the `abs()` method are as follows:

```
public static int abs(int a)
public static long abs(long a)
public static float abs(float a)
public static double abs(double a)
```

### ceil()

The `ceil()` method returns the smallest integer, as a *double*, that is *greater than or equal to the argument* and *equal to the nearest integer value*. In other words, the argument is rounded up to the nearest integer equivalent.

Let's look at some examples of this in action, just to make sure you are familiar with the concept. All the following calls to `Math.ceil()` return the *double* value 9.0:

```
Math.ceil(9.0)     // result is 9.0
Math.ceil(8.8)     // rises to 9.0
Math.ceil(8.02)    // still rises to 9.0
```

Negative numbers are similar, but just remember that –9 is greater than –10. All the following calls to `Math.ceil()` return the *double* value -9.0:

```
Math.ceil(-9.0)    // result is -9.0
Math.ceil(-9.4)    // rises to -9.0
Math.ceil(-9.8)    // still rises to -9.0
```

There is only one `ceil()` method and it has the following signature:

```
public static double ceil(double a)
```

### floor()

The `floor()` method returns the largest *double* that is less than or equal to the argument and equal to the nearest integer value. This method is the antithesis of the `ceil()` method.

All the following calls to `Math.floor()` return the *double* value 9.0:

```
Math.floor(9.0)      // result is 9.0
Math.floor(9.4)      // drops to 9.0
Math.floor(9.8)      // still drops to 9.0
```

As before, keep in mind that with negative numbers, –9 is less than –8! All the following calls to `Math.floor()` return the *double* value –9.0:

```
Math.floor(-9.0)     // result is -9.0
Math.floor(-8.8)     // drops to -9.0
Math.floor(-8.1)     // still drops to -9.0
```

The signature of the `floor()` method is as follows:

```
public static double floor(double a)
```

exam watch

***The `floor()` and `ceil()` methods take only doubles. There are no overloaded methods for integral numbers, because the methods would just end up returning the integral numbers they were passed. The whole point of `floor()` and `ceil()` is to convert floating-point numbers (`doubles`), to integers, based on the rules of the methods. It may seem strange (it does to us) that the integer values are returned in a double sized container, but don't let that throw you.***

## max()

The `max()` method takes two numeric arguments and returns the greater of the two—for example,

```
x = Math.max(1024, -5000);     //  output is 1024.
```

This method is overloaded to handle *int*, *long*, *float*, or *double* arguments. If the input parameters are the same, `max()` returns a value equal to the two arguments. The signatures of the `max()` method are as follows:

```
public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)
```

### min()

The `min()` method is the antithesis of the `max()` method; it takes two numeric arguments and returns the lesser of the two—for example,

```
x = Math.min(0.5, 0.0);   //   output is  0.0
```

This method is overloaded to handle *int*, *long*, *float*, or *double* arguments. If the input parameters are the same, `min()` returns a value equal to the two arguments. The signatures of the `min()` method are as follows:

```
public static int min(int a, int b)
public static long min(long a, long b)
public static float min(float a, float b)
public static double min(double a, double b)
```

And for the record, we're pretty impressed with our use of the word "antithesis".

## EXERCISE 6-1

### Using the Math Class

In this exercise we will examine some numbers using the `abs()`, `ceil()`, and `floor()` methods of the Math class. Find the absolute, ceiling, and floor values of the following numbers: 10.5, –10.5, Math.PI, and 0.

- Create a class and a `main()` method to perform the calculations.
- Store these numbers in an array of *double* values.
- Use a *for* loop to go through the array and perform the tests on each of these numbers.
- Try to determine what the results of your program will be before running it.
- An example solution is provided at the end of the chapter.

### random()

The `random()` method returns a random *double* that is greater than or equal to 0.0 and less than 1.0. The `random()` method does not take any parameters—for example,

```
public class RandomTest {
  public static void main(String [] args) {
    for (int x=0; x < 15; x++)
      System.out.print( (int)(Math.random()*10) + " " );
  }
}
```

The `println()` method multiplies the result of the call to `Math.random()` by 10, and then casts the resulting *double* (whose value will be between 0.0 and 9.99999999…), to an integer. Here are some sample results:

```
6 3 3 1 2 0 5 9 3 5 6 6 0 3 5
4 9 3 6 6 8 1 1 3 0 3 2 5 3 4
```

The signature of the `random()` method is as follows:

```
public static double random( )
```

## round()

The `round()` method returns the integer closest to the argument. The algorithm is to add 0.5 to the argument and truncate to the nearest integer equivalent. This method is overloaded to handle a *float* or a *double* argument.

The methods `ceil()`, `floor()`, and `round()` all take floating-point arguments and return integer equivalents (although again, delivered in a *double* variable). If the number after the decimal point is *less than* 0.5, `Math.round()` is equal to `Math.floor()`. If the number after the decimal point is *greater than or equal to* 0.5, `Math.round()` is equal to `Math.ceil()`. Keep in mind that with negative numbers, a number at the .5 mark will round up to the *larger* number—for example,

```
Math.round(-10.5);     // result is -10
```

The signatures of the `round()` method are as follows:

```
public static int round(float a)
public static long round(double a)
```

## sin()

The `sin()` method returns the sine of an angle. The argument is a *double* representing an angle *in radians*. Degrees can be converted to radians by using `Math.toRadians()`—for example,

```
Math.sin(Math.toRadians(90.0)) //  returns 1.0
```

The signature of the `sin()` method is as follows:

```
public static double sin(double a)
```

### cos()

The `cos()` method returns the cosine of an angle. The argument is a *double* representing an angle *in radians*—for example,

```
Math.cos(Math.toRadians(0.0)) // returns 1.0
```

The signature of the `cos()` method is as follows:

```
public static double cos(double a)
```

### tan()

The `tan()` method returns the tangent of an angle. The argument is a *double* representing an angle *in radians*—for example,

```
Math.tan(Math.toRadians(45.0))   // returns 1.0
```

The signature of the `tan()` method is as follows:

```
public static double tan(double a)
```

exam Watch

***Sun does not expect you to be a human calculator. The certification exam will not contain questions that require you to verify the result of calling methods such as `Math.cos(0.623)`. (Although we thought it would be*** *fun* ***to include questions like that…)***

### sqrt()

The `sqrt()` method returns the square root of a *double*—`f`or example,

```
Math.sqrt(9.0) // returns 3.0
```

What if you try to determine the square root of a negative number? After all, the actual mathematical square root function returns a complex number (comprised of real and imaginary parts) when the operand is negative. The Java `Math.sqrt()` method returns `NaN` instead of an object representing a complex number. `NaN` is a bit pattern that denotes "not a number." The signature of the `sqrt()` method is as follows:

```
public static double sqrt(double a)
```

## EXERCISE 6-2

### Rounding Random Numbers

In this exercise we will round a series of random numbers. The program will generate ten random numbers from 0 through 100. Round each one of them, then print the results to the screen. Try to do this with as little code as possible.

1. Create a class and a `main()` method to perform the calculations.
2. Use a *for* loop to go through ten iterations.
3. Each iteration should generate a random number using `Math.random()`. To get a number from 0 through 100 simply multiply the random number by 100. Print this number to the screen. Without rounding it, though, you can't ever get to 100 (the `random()` method always returns something *less* than 1.0).
4. Round the number using the `Math.round()` method. Print the rounded number to the screen.
5. A sample solution is listed at the end of the chapter.

As a bonus, note whether the numbers look random. Is there an equal number of even and odd numbers? Are they grouped more towards the top half of 100 or the bottom half? What happens to the distribution as you generate more random numbers?

### toDegrees()

The `toDegrees()` method takes an argument representing an angle in radians and returns the equivalent angle in degrees—for example,

```
Math.toDegrees(Math.PI * 2.0)   //  returns 360.0
```

The signature of the `toDegrees()` method is as follows:

```
public static double toDegrees(double a)
```

### toRadians()

The `toRadians()` method takes an argument representing an angle in degrees and returns the equivalent angle in radians—for example,

```
Math.toRadians(360.0)   //  returns 6.283185, which is 2 * Math.PI
```

This method is useful for converting an angle in degrees to an argument suitable for use with the trigonometric methods (`cos()`, `sin()`, `tan()`, `acos()`, `asin()`, and `atan()`). For example, to determine the sin of 60 degrees:

```
double d = Math.toRadians(60);
System.out.println("sin 60 = " + Math.sin(d) ); // "sin 60 = 0.866…"
```

The signature of the `toRadians()` method is as follows:

```
public static double toRadians(double a)
```

Table 6-1 summarizes the key static methods of the Math class.

**TABLE 6-1** Important Static Math Class Method Signatures

| Static Math Methods |
|---|
| double ceil ( double a ) |
| double floor ( double a ) |
| double random ( ) |
| double abs ( double a ) |
| float abs ( float a ) |
| int abs ( int a ) |
| long abs ( long a ) |
| double max ( double a, double b ) |
| float max ( float a, float b ) |
| int max ( int a, int b ) |
| long max (long a, long b ) |
| double min ( double a, double b ) |
| float min ( float a, float b )double sqrt ( double a ) |
| int min ( int a, int b ) |
| long min ( long a, long b ) |
| double toDegrees ( double angleInRadians ) |
| double toRadians ( double angleInDegrees ) |
| double tan ( double a ) |

**TABLE 6-1** Important Static Math Class Method Signatures *(continued)*

| Static Math Methods |
|---|
| double sin ( double a ) |
| double cos ( double a ) |
| double sqrt ( double a ) |
| int round ( float a ) |
| long round ( double a ) |

## Miscellaneous Math Class Facts

The following program demonstrates some of the unusual results that can occur when pushing some of the limits of the Math class or performing mathematical functions that are "on the edge" (such as dividing floating-point numbers by 0). These are some of the basic special cases. There are many more, but if you know these you will be in good shape for the exam.

exam Watch

***If you want to live dangerously, or you're running out of study time before the big day, just focus on the examples below with the **.***

```
double d;
float p_i = Float.POSITIVE_INFINITY;    // The floating point classes have
double n_i = Double.NEGATIVE_INFINITY;  // these three special fields.
double notanum = Double.NaN;            // They can be Float or Double

if ( notanum != notanum )               // ** NaN isn't == to anything, not
                                        // even itself!
  System.out.println("NaNs not equal"); // result is  "NaNs not equal"

if ( Double.isNaN(notanum))             // Float and Double have isNan()
                                        // methods to test for NaNs
  System.out.println("got a NaN");      // result is  "got a NaN"

d = Math.sqrt(n_i);                     // square root of negative infinity?
if ( Double.isNaN(d) )
  System.out.println("got sqrt NaN");   // result is "got sqrt NaN"

System.out.println( Math.sqrt(-16d));   // result is "NaN"

System.out.println( 16d / 0.0 );       // ** result is (positive) "Infinity"
System.out.println( 16d / -0.0 );      // ** result is (negative) "-Infinity"

     // divide by 0 only works for floating point numbers
     // divide by 0 with integer numbers results in ArithmeticException

System.out.println("abs(-0) = "+ Math.abs(-0));  // result is "abs(-0) = 0"
```

exam Watch

***The exam will test your knowledge of implicit casting. For the numeric primitives, remember that from narrowest to widest the numeric primitives types are** byte, short, int, long, float, double**. Any numeric primitive can be implicitly cast to any numeric primitive type that is wider than itself. For instance, a** byte **can be implicitly cast to any other numeric primitive, but a** float **can only be implicitly cast to a** double**. Remembering implicit casting, and the method signatures in Table 6-1, will help you answer many of the exam questions.***

CERTIFICATION OBJECTIVE

# Using Wrapper Classes (Exam Objective 8.3)

*Describe the significance of wrapper classes, including making appropriate selections in the wrapper classes to suit specified behavior requirements, stating the result of executing a fragment of code that includes an instance of one of the wrapper classes, and writing code using the following methods of the wrapper classes (e.g., Integer, Double, etc.): doubleValue, floatValue, intValue, longValue, parseXxx, getXxx, toString, toHexString.*

The wrapper classes in the Java API serve two primary purposes:

- To provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value.
- To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

## An Overview of the Wrapper Classes

There is a wrapper class for every primitive in Java. For instance the wrapper class for *int* is Integer, for *float* is Float, and so on. *Remember that the primitive name is*

*simply the lowercase name of the wrapper except for char, which maps to Character, and int, which maps to Integer*. Table 6-2 lists the wrapper classes in the Java API.

## Creating Wrapper Objects

For the exam you need to understand the three most common approaches for creating wrapper objects. Some approaches take a String representation of a primitive as an argument. Those that take a String throw NumberFormatException if the String provided cannot be parsed into the appropriate primitive. For example "two" can't be parsed into "2". Like another class previously discussed in this chapter, wrapper objects are immutable. Once they have been given a value, that value cannot be changed. (Can you guess which other class we're talking about?)

### The Wrapper Constructors

All of the wrapper classes except Character provide two constructors: one that takes a primitive of the type being constructed, and one that takes a String representation of the type being constructed—for example,

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

or

```
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");
```

**TABLE 6-2**

Wrapper Classes and Their Constructor Arguments

| Primitive | Wrapper Class | Constructor Arguments |
|---|---|---|
| boolean | Boolean | boolean or String |
| byte | Byte | byte or String |
| char | Character | char |
| double | Double | double or String |
| float | Float | float or String |
| int | Integer | int or String |
| long | Long | long or String |
| short | Short | short or String |

The Character class provides only one constructor, which takes a *char* as an argument—for example,

```
Character c1 = new Character('c');
```

exam watch

***The constructors for the Boolean wrapper take either a boolean value `true` or `false`, or a case-insensitive String with the value "true" or "false". But a Boolean object can't be used as an expression in a boolean test—for instance,***

```
Boolean b = new Boolean("false");
if (b)   // won't compile, expecting a boolean not a Boolean
```

### The valueOf() Methods

The static `valueOf()` methods provided in most of the wrapper classes give you another approach to creating wrapper objects. Both methods take a String representation of the appropriate type of primitive as their first argument, the second method (when provided) takes an additional argument, `int radix`, which indicates in what base (for example binary, octal, or hexadecimal) the first argument is represented—for example,

```
Integer i2 = Integer.valueOf("101011", 2);  // converts 101011 to 43 and
                                            // assigns the value 43 to the
                                            // Integer object i2
```

or

```
Float f2 = Float.valueOf("3.14f");   // assigns 3.14 to the Float object f2
```

## Using Wrapper Conversion Utilities

As we said earlier, a wrapper's second big function is converting stuff. The following methods are the most commonly used, and are the ones you're most likely to see on the test.

### xxxValue()

When you need to convert the value of a wrapped numeric to a primitive, use one of the many `xxxValue()` methods. All of the methods in this family are no-arg methods. As you can see by referring to Table 6-3, there are 36 `xxxValue()` methods. Each of the six numeric wrapper classes has six methods, so that any numeric wrapper can be converted to any primitive numeric type—for example,

```
Integer i2 = new Integer(42);  //  make a new wrapper object
byte b = i2.byteValue();       //  convert i2's value to a byte
                               //  primitive
short s = i2.shortValue();     //  another of Integer's xxxValue
                               //  methods
double d = i2.doubleValue();   //  yet another of Integer's
                               //  xxxValue methods
```

or

```
Float f2 = new Float(3.14f);   // make a new wrapper object
short s = f2.shortValue();     // convert f2's value to a short
                               // primitive
System.out.println(s);         // result is 3  (truncated, not
                               // rounded)
```

**TABLE 6-3** Common Wrapper Conversion Methods

| **Method**<br>s = static<br>n = NFE exception | **Boolean** | **Byte** | **Character** | **Double** | **Float** | **Integer** | **Long** | **Short** |
|---|---|---|---|---|---|---|---|---|
| byteValue | | x | | x | x | x | x | x |
| doubleValue | | x | | x | x | x | x | x |
| floatValue | | x | | x | x | x | x | x |
| intValue | | x | | x | x | x | x | x |
| longValue | | x | | x | x | x | x | x |
| shortValue | | x | | x | x | x | x | x |
| | | | | | | | | |
| parseXxx *s,n* | | x | | x | x | x | x | x |
| parseXxx *s,n*<br>(with radix) | | x | | | | x | x | x |
| | | | | | | | | |
| valueOf *s,n* | x | x | | x | x | x | x | x |
| valueOf *s,n*<br>(with radix) | | x | | | | x | x | x |

**TABLE 6-3** Common Wrapper Conversion Methods *(continued)*

| Method<br>s = static<br>n = NFE exception | Boolean | Byte | Character | Double | Float | Integer | Long | Short |
|---|---|---|---|---|---|---|---|---|
| toString | x | x | x | x | x | x | x | x |
| toString *s*<br>(primitive) | | x | | x | x | x | x | x |
| toString *s*<br>(primitive, radix) | | | | | | x | x | |
| | | | | | | | | |
| toBinaryString *s* | | | | | | x | x | |
| toHexString *s* | | | | | | x | x | |
| toOctalString *s* | | | | | | x | x | |

In summary, the essential method signatures for Wrapper conversion methods are

- primitive xxxValue( )
- primitive parseXxx(String)
- Wrapper valueOf(String)

## parseXxx() and valueOf()

The six `parseXxx()` methods (one for each numeric wrapper type) are closely related to the `valueOf()` method that exists in all of the numeric wrapper classes (plus Boolean). Both `parseXxx()` and `valueOf()` take a String as an argument, throw a NumberFormatException if the String argument is not properly formed, and can convert String objects from different bases (radix), when the underlying primitive type is any of the four integer types. (See Table 6-3.)

The difference between the two methods is

- `parseXxx()` returns the named primitive.
- `valueOf()` returns a newly created wrapped object of the type that invoked the method.

Some examples of these methods in action:

```
double d4 = Double.parseDouble("3.14");    // convert a String to a primitive
System.out.println("d4 = " + d4);          // result is  "d4 = 3.14"

Double d5 = Double.valueOf("3.14");        // create a Double object
System.out.println(d5 instanceof Double ); // result is "true"
```

The next examples involve using the radix argument, (in this case binary):

```
long L2 = Long.parseLong("101010", 2);    // binary String to a primitive
System.out.println("L2 = " + L2);         // result is "L2 = 42"

Long L3 = Long.valueOf("101010", 2);      // binary String to Long object
System.out.println("L3 value = " + L3);   // result is "L2 value = 42"
```

### toString()

The class Object, the alpha class, the top dog, has a `toString()` method. Since we know that all other Java classes inherit from class Object, we also know (stay with me here) that all other Java classes have a `toString()` method. The idea of the `toString()` method is to allow you to get some *meaningful representation* of a given object. For instance, if you have a Collection of various types of objects, you can loop through the Collection and print out some sort of meaningful representation of each object using the `toString()` method, which is guaranteed to be in every class. We'll talk more about the `toString()` method in the Collections chapter, but for now let's focus on how the `toString()` method relates to the wrapper classes which, as we know, are marked `final`. All of the wrapper classes have a *no-arg*, *nonstatic*, *instance* version of `toString()`. This method returns a String with the value of the primitive wrapped in the object—for instance,

```
Double d = new Double("3.14");
System.out.println("d = " + d.toString() );    //  result is "d = 3.14"
```

All of the numeric wrapper classes provide an overloaded, `static` `toString()` method that takes a primitive numeric of the appropriate type (`Double.toString()` takes a *double*, `Long.toString()` takes a *long*, etc.), and, of course, returns a String with that primitive's value—for example,

```
System.out.println("d = " + Double.toString(3.14);    // result is "d = 3.14"
```

Finally, Integer and Long provide a third `toString()` method. It is `static`, its first argument is the appropriate primitive, and its second argument is a *radix*. The radix argument tells the method to take the first argument (which is radix 10 or base 10 by default), and convert it to the radix provided, then return the result as a String—for instance,

```
System.out.println("hex = " + Long.toString(254,16); // result is "hex = fe"
```

### toXxxString() (Binary, Hexadecimal, Octal)

The Integer and Long wrapper classes let you convert numbers in base 10 to other bases. These conversion methods, `toXxxString()`, take an *int* or *long*, and return a String representation of the converted number, for example,

```
String s3 = Integer.toHexString(254);       // convert 254 to hex
System.out.println("254 in hex = " + s3);   // result is "254 in hex = fe"

String s4 = Long.toOctalString(254);        // convert 254 to octal
System.out.println("254 in octal = "+ s4);  // result is "254 in octal = 376"
```

Studying Table 6-3 is the single best way to prepare for this section of the test. If you can keep the differences between `xxxValue()`, `parseXxx()`, and `valueOf()` straight, you should do well on this part of the exam.

**CERTIFICATION OBJECTIVE**

# Using equals()(Exam Objective 5.2)

*Determine the result of applying the boolean `equals(Object)` method to objects of any combination of the classes java.lang.String, java.lang.Boolean, and java.lang.Object.*

In this chapter we begin our discussion of == and the `equals()` method, and in the Collections chapter we'll dive deeper into these two mysterious comrades. For now, we'll limit our discussion to how == and the `equals()` method relate to String, and the wrapper classes, and an overview of other object classes.

## An Overview of == and the equals() Method

There are three kinds of entities in Java that we might want to compare to determine if they're equivalent: primitive variables, reference variables, and objects. Part of this discussion looks at a critical question: *What exactly does "equivalent" mean?*

### Comparing Variables

Let's start with primitive and reference variables. You always compare primitive variables using ==; the `equals()` method obviously can't be used on primitives.

The == operator returns a *boolean* value: `true` if the variables are equivalent, `false` if they're not. Primitive variables are stored in memory as some absolute number of bits, depending on the type of primitive being handled (*short* is 16 bits, *int* is 32 bits, *long* is 64 bits, etc.). On the other hand, we can't know from one Java implementation to the next how big a reference variable is—it might be 64 bits, it might be 97 bits (probably not!)—but the key thing to remember is that wherever a Java program might run, all of the reference variables running on a single VM will be the same size (in bits) and format. When we use the == operator to compare two reference variables, we're *really* testing to see if the two reference variables *refer to the same object!* So remember that when you compare variables (of either type, primitive or reference), you are really comparing two sets of bit patterns.

Either bit patterns are the same, or they're not. If primitive *a* holds a 5, and primitive *b* holds a 5, then the bits in *a* and *b* are the same and *a* == *b* will be true. If a reference variable *c* refers to object `X017432` and reference variable *d* also refers to object `X017432`, then the bits in *c* and *d* are the same, and *c* == *d* will be true.

When comparing reference variables with the == operator, you can only compare reference variables that refer to objects that are in the same class or class hierarchy. Attempting to use == to compare reference variables for objects in different class hierarchies will result in a compiler error.

exam watch

***Key facts to remember about comparing variables:***

1. ***The rule is the same for reference variables and primitive variables: == returns true if the two bit patterns are identical.***
2. ***Primitive variables must use ==; they cannot use the `equals()` method.***
3. ***For reference variables, == means that both reference variables are referring to the same object.***

## Comparing Objects

We saw what it means to compare reference variables (to see if they refer to the same object), but what does it mean to compare the objects themselves? For an object as simple as a String, it's fairly intuitive to say that if two String objects have the same value (in other words the same characters), we consider them equal. When you want to determine if two objects are *meaningfully equivalent*, use the `equals()` method. Like ==, the `equals()` method returns a *boolean* `true` if the objects are considered equivalent; otherwise, it returns *false*. (Remember, if we want to know whether two String reference variables refer to the same String, we must use ==.) Given the following code sample,

```
String x1 = "abc";
String x2 = "ab";
x2 = x2 + "c";
```

we might want to know, much later on in our code, whether the *contents* of the two different String objects *x1* and *x2* are in fact the same. This is where the `equals()` method comes in:

```
if ( x1 != x2 ) {                            // comparing reference vars
   System.out.println("different objects");
}
if ( x1.equals(x2) ) {                       // comparing values
   System.out.println("same values");
}
```

In the example above we could also have written this:

```
if ( x2.equals(x1) ) {                       // same result
```

In a similar vein, it's a pretty safe bet that when we want to compare two wrapper objects, we're really interested in the primitive *values* that they're wrapping. However, it's important to know that all of the wrapper class' `equals()` methods only return `true` if *both* the primitive values *and* the wrapper's classes are the same.

```
Double d1 = new Double("3.0");
Integer i1 = new Integer(3);              // create a couple of wrappers

if ( d1.equals(i1) )  {                   // are the values equal ?
   System.out.println("wraps are equal");  // no output, different classes
}

Double d2 = d1.valueOf("3.0d");             // create a third wrapper
if (d1.equals(d2) )  {                      // are the Doubles equal ?
   System.out.println("Doubles are equal");  // result is "Doubles are equal"
}
```

## The equals() Method Revealed (or at Least a Little Bit Revealed)

We'll be diving in to the `equals()` method much more deeply in the Collections chapter, but for now let's just cover a few key points. The class Object, the granddaddy of all classes (and from which all classes extend), has an `equals()` method. That means every other Java class (including those in the API or those that you create) inherits an `equals()` method. In `java.lang`, the String and wrapper classes

have overridden the `equals()` method to behave as we just discussed. And remember, the String and wrapper classes are all marked `final`, so you can't override any of their methods, *including* the `equals()` method.

When you create your own classes, you'll have to decide what it means for two distinct objects to be meaningfully equivalent. Your class may have reference variables that collectively represent the value of an instance. If you want to compare instances of a class to one another, it will be up to you to override the `equals()` method to define what it means for two different instances to be *meaningfully* equal.

exam Watch

***Remember the following key points about the `equals()` method:***

1. ***`equals()` is used only to compare objects.***
2. ***`equals()` returns a boolean, `true` or `false`.***
3. ***The StringBuffer class has not overridden `equals()`.***
4. ***The String and wrapper classes are `final` and have overridden `equals()`.***

# CERTIFICATION SUMMARY

## Strings

At the risk of being pedantic, remember that String *objects* are immutable, *references* to Strings are not! You learned that 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 a 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. They're all fairly intuitive except for `substring()`, which needs a little extra brainpower. (And did we mention how annoying—*possibly evil*—it is that the developers of the `substring()` method didn't follow the Java naming convention? It should have been `subString()`!)

StringBuffers are not immutable—you can change them over and over again. The StringBuffer methods are fairly intuitive, but remember that unlike String methods, they *do* modify the StringBuffer object, even if you don't assign the result to anything.

## Math

As the Math class relates to the certification exam, you won't be expected to reproduce complicated mathematical algorithms in your head or know the cosine of an angle. But remember that you *will* need to know how to calculate the result of calling `abs()`, `ceil()`, `floor()`, `max()`, `min()`, and `round()` with

any given values. Know the method signatures in Table 6-1. The exam will test your ability to remember method signatures and follow simple algorithms. Most questions on the Math class are quite simple as long as you've spent the time to commit to memory the Math class methods and their calling signatures. Table 6-1 will really help.

While you're at it, spend some time studying Table 6-1. It's important to know which methods are overridden and which are not. And just in case we're not making ourselves clear, *we really want you to study Table 6-1.*

## Wrappers

Remember that wrappers have two main functions: to wrap primitives so they can be treated like objects, and to provide utility methods for primitives (typically conversions). All the wrapper classes have the same name, capitalized, as their primitive counterparts except for Character and Integer. Remember that Boolean objects can't be used like *boolean* primitives. In terms of return on investment for your studying time, make sure that you know the details of the `xxxValue()` methods, the `parseXxx()` methods, the `valueOf()` methods, and the `toString()` methods. Pay attention to which methods are `static` and which throw NumberFormatException. Study Table 6-3. Copy it by hand, and then place it under your pillow. Frame it and hang it on your wall.

## Equals()

Compare primitives with ==. To determine if two reference variables refer to the *same object, use* ==. To determine if two objects are *meaningfully equivalent, use* `equals()`. When using == to compare reference variables, the compiler will verify that the classes are the same or in the same inheritance hierarchy. Remember that the *StringBuffer class does not override the* `equals()` *method*, which means that there is no built-in method to determine if the contents of one StringBuffer object are the same as the contents of another StringBuffer object.

# TWO-MINUTE DRILL

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

## Using the java.lang.String Class (Exam Objective 8.2)

- ❑ 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 a String literal is encountered by the VM, it is added to the pool.
- ❑ Strings have a method named `length()`, arrays have an attribute named *length*.
- ❑ StringBuffers are mutable—they can change without creating a new object.
- ❑ StringBuffer methods act on the invoking object, but objects can change without an explicit assignment in the statement.
- ❑ StringBuffer `equals()` is not overridden; it doesn't compare values.
- ❑ In all sections, remember that *chained* methods are evaluated from left to right.

## Using the java.lang.Math Class (Exam Objective 8.1)

- ❑ The `abs()` method is overloaded to take an *int*, a *long*, a *float*, or a *double*.
- ❑ The `abs()` method can return a negative if the argument is the minimum *int* or *long* value equal to the value of Integer.MIN_VALUE or Long.MIN_VALUE, respectively.
- ❑ The `max()` method is overloaded to take *int*, *long*, *float*, or *double* arguments.
- ❑ The `min()` method is overloaded to take *int*, *long*, *float*, or *double* arguments.
- ❑ The `random()` method returns a *double* greater than or equal to 0.0 and less than 1.0.

- The `random()` does not take any arguments.
- The methods `ceil()`, `floor()`, and `round()` all return integer equivalent floating-point numbers, `ceil()` and `floor()` return *doubles*, `round()` returns a *float* if it was passed an *int*, or it returns a *double* if it was passed a *long*.
- The `round()` method is overloaded to take a *float* or a *double*.
- The methods `sin()`, `cos()`, and `tan()` take a *double* angle in radians.
- The method `sqrt()` can return `NaN` if the argument is `NaN` or less than zero.
- Floating-point numbers can be divided by 0.0 without error; the result is either positive or negative infinity.
- `NaN` is not equal to anything, not even itself.

### Using Wrappers (Exam Objective 8.3)

- The wrapper classes correlate to the primitive types.
- Wrappers have two main functions:
  - To wrap primitives so that they can be handled like objects
  - To provide utility methods for primitives (usually conversions)
- Other than Character and Integer, wrapper class names are the primitive's name, capitalized.
- Wrapper constructors can take a String or a primitive, except for Character, which can only take a *char*.
- A Boolean object can't be used like a *boolean* primitive.
- The three most important method families are
  - `xxxValue()` Takes no arguments, returns a primitive
  - `parseXxx()` Takes a String, returns a primitive, is static, throws NFE
  - `valueOf()` Takes a String, returns a wrapped object, is static, throws NFE
- Radix refers to bases (typically) other than 10; binary is radix 2, octal = 8, hex = 16.

### Using equals() (Exam Objective 5.2)

- ❑ Use == to compare primitive variables.
- ❑ Use == to determine if two reference variables refer to the *same object*.
- ❑ == compares bit patterns, either primitive bits or reference bits.
- ❑ Use `equals()` to determine if two objects are *meaningfully equivalent*.
- ❑ The String and Wrapper classes override `equals()` to check for values.
- ❑ The StringBuffer class `equals()` is *not* overridden; it uses == under the covers.
- ❑ The compiler will not allow == if the classes are not in the same hierarchy.
- ❑ Wrappers won't pass `equals()` if they are in different classes.

# SELF TEST

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

## Using the java.lang.String Class (Exam Objective 8.2)

**1.** Given the following,

```
1.  public class StringRef {
2.    public static void main(String [] args) {
3.      String s1 = "abc";
4.      String s2 = "def";
5.      String s3 = s2;
6.      s2 = "ghi";
7.      System.out.println(s1 + s2 + s3);
8.    }
9.  }
```

what is the result?

A. abcdefghi

B. abcdefdef

C. abcghidef

D. abcghighi

E. Compilation fails.

F. An exception is thrown at runtime.

**2.** Given the following,

```
11.   String x = "xyz";
12.   x.toUpperCase();
13.   String y = x.replace('Y', 'y');
14.   y = y + "abc";
15.   System.out.println(y);
```

what is the result?

A. abcXyZ

B. abcxyz

C. xyzabc

D. `XyZabc`

E. Compilation fails.

F. An exception is thrown at runtime.

**3.** Given the following,

```
13.  String x = new String("xyz");
14.  y = "abc";
15.  x = x + y;
```

how many String objects have been created?

A. 2

B. 3

C. 4

D. 5

**4.** Given the following,

```
14.    String a = "newspaper";
15.    a = a.substring(5,7);
16.    char b = a.charAt(1);
17.    a = a + b;
18.    System.out.println(a);
```

what is the result?

A. `apa`

B. `app`

C. `apea`

D. `apep`

E. `papp`

F. `papa`

**5.** Given the following,

```
4.    String d = "bookkeeper";
5.    d.substring(1,7);
6.    d = "w" + d;
7.    d.append("woo");
8.    System.out.println(d);
```

what is the result?

A. `wookkeewoo`

B. `wbookkeeper`

C. `wbookkeewoo`

D. `wbookkeeperwoo`

E. Compilation fails.

F. An exception is thrown at runtime.

## Using the java.lang.Math Class (Exam Objective 8.1)

**6.** Given the following,

```
1.   public class Example {
2.     public static void main(String [] args) {
3.       double values[] = {-2.3, -1.0, 0.25, 4};
4.       int cnt = 0;
5.       for (int x=0; x < values.length; x++) {
6.         if (Math.round(values[x] + .5) == Math.ceil(values[x])) {
7.           ++cnt;
8.         }
9.       }
10.      System.out.println("same results " + cnt + " time(s)");
11.    }
12.  }
```

what is the result?

A. same results 0 time(s)

B. same results 2 time(s)

C. same results 4 time(s)

D. Compilation fails.

E. An exception is thrown at runtime.

**7.** Which of the following are valid calls to Math.max? (Choose all that apply.) (Yeah, yeah, we know that on the *real* exam you'd know how many were correct, but we just want you to work a little harder here.)

A. `Math.max(1,4)`

B. `Math.max(2.3, 5)`

C. `Math.max(1, 3, 5, 7)`

D. `Math.max(-1.5, -2.8f)`

**8.** What two statements are true about the result obtained from calling `Math.random()`? (Choose two.)

A. The result is less than 0.0.

B. The result is greater than or equal to 0.0..

C. The result is less than 1.0.

D. The result is greater than 1.0.

E. The result is greater than or equal to 1.0.

F. The result is less than or equal to 1.0.

**9.** Given the following,

```
1.   public class SqrtExample {
2.     public static void main(String [] args) {
3.       double value = -9.0;
4.       System.out.println( Math.sqrt(value));
5.     }
6.   }
```

what is the result?

A. 3.0

B. –3.0

C. NaN

D. Compilation fails.

E. An exception is thrown at runtime.

**10.** Given the following,

```
1.  public class Degrees {
2.    public static void main(String [] args) {
3.      System.out.println( Math.sin(75) );
4.      System.out.println( Math.toDegrees(Math.sin(75) ));
5.      System.out.println( Math.sin(Math.toRadians(75) ));
6.      System.out.println( Math.toRadians(Math.sin(75) ));
7.    }
8.  }
```

at what line will the sine of 75 degrees be output?

A. Line 3
B. Line 4
C. Line 5
D. Line 6
E. Line 3 and either line 4, 5, or 6
F. None of the above

## Using Wrapper Classes (Exam Objective 8.3)

**11.** Given the following,

```
1.  public class WrapTest2 {
2.     public static void main(String [] args) {
3.       Long b = new Long(42);
4.       int x = Integer.valueOf("345");
5.       int x2 = (int) Integer.parseInt("345", 8);
6.       int x3 = Integer.parseInt(42);
7.       int x4 = Integer.parseInt("42");
8.       int x5 = b.intValue();
9.     }
10. }
```

which two lines will cause compiler errors? (Choose two.)

A. Line 3
B. Line 4
C. Line 5
D. Line 6
E. Line 7
F. Line 8

**12.** Given the following,

```
1.   public class NFE {
2.     public static void main(String [] args) {
3.       String s = "42";
4.       try {
5.         s = s.concat(".5");
6.         double d = Double.parseDouble(s);
7.         s = Double.toString(d);
8.         int x = (int) Math.ceil(Double.valueOf(s).doubleValue());
9.         System.out.println(x);
```

```
10.       }
11.       catch (NumberFormatException e) {
12.         System.out.println("bad number");
13.       }
14.     }
15.   }
```

what is the result?

A. 42

B. 42.5

C. 43

D. bad number

E. Compilation fails.

F. An uncaught exception is thrown at runtime.

**13.** Given the following,

```
1.  public class BoolTest {
2.    public static void main(String [] args) {
3.      Boolean b1 = new Boolean("false");
4.      boolean b2;
5.      b2 = b1.booleanValue();
6.      if (!b2) {
7.        b2 = true;
8.        System.out.print("x ");
9.      }
10.     if (b1 & b2) {
11.       System.out.print("y ");
12.     }
13.     System.out.println("z");
14.   }
15. }
```

what is the result?

A. z

B. x z

C. y z

D. x y z

E. Compilation fails.

F. An exception is thrown at runtime.

**14.** Given the following,

```
1.   public class WrapTest3 {
2.     public static void main(String [] args) {
3.       String s = "98.6";
4.       // insert code here
5.     }
6.   }
```

which three lines inserted independently at line 4 will cause compiler errors? (Choose three.)

A. `float f1 = Float.floatValue(s);`

B. `float f2 = Float.valueOf(s);`

C. `float f3 = new Float(3.14f).floatValue();`

D. `float f4 = Float.parseFloat(1.23f);`

E. `float f5 = Float.valueOf(s).floatValue();`

F. `float f6 = (float) Double.parseDouble("3.14");`

**15.** Given the following,

```
11.    try {
12.      Float f1 = new Float("3.0");
13.      int x = f1.intValue();
14.      byte b = f1.byteValue();
15.      double d = f1.doubleValue();
16.      System.out.println(x + b + d);
17.    }
18.    catch (NumberFormatException e) {
19.      System.out.println("bad number");
20.    }
```

what is the result?

A. `9.0`

B. `bad number`

C. Compilation fails on line 13.

D. Compilation fails on line 14.

E. Compilation fails on lines 13 and 14.

F. An uncaught exception is thrown at runtime.

## Using equals() (Exam Objective 5.2)

**16.** Given the following,

```
1.   public class WrapTest {
2.     public static void main(String [] args) {
3.       int result = 0;
4.       short s = 42;
5.       Long x = new Long("42");
6.       Long y = new Long(42);
7.       Short z = new Short("42");
8.       Short x2 = new Short(s);
9.       Integer y2 = new Integer("42");
10.      Integer z2 = new Integer(42);
11.
12.      if (x == y) result = 1;
13.      if (x.equals(y) ) result = result + 10;
14.      if (x.equals(z) ) result = result + 100;
15.      if (x.equals(x2) ) result = result + 1000;
16.      if (x.equals(z2) ) result = result + 10000;
17.
18.      System.out.println("result = " + result);
19.    }
20.  }
```

what is the result?

A. `result = 1`

B. `result = 10`

C. `result = 11`

D. `result = 11010`

E. `result = 11011`

F. `result = 11111`

**17.** Given the following,

```
1.   public class BoolTest {
2.     public static void main(String [] args) {
3.       int result = 0;
4.
5.       Boolean b1 = new Boolean("TRUE");
6.       Boolean b2 = new Boolean("true");
```

```
7.        Boolean b3 = new Boolean("tRuE");
8.        Boolean b4 = new Boolean("false");
9.
10.       if (b1 == b2)  result = 1;
11.       if (b1.equals(b2) ) result = result + 10;
12.       if (b2 == b4)  result = result + 100;
13.       if (b2.equals(b4) ) result = result + 1000;
14.       if (b2.equals(b3) ) result = result + 10000;
15.
16.       System.out.println("result = " + result);
17.     }
18.   }
```

what is the result?

A. 0

B. 1

C. 10

D. 1100

E. 10001

F. 10010

**18.** Given the following,

```
1.   public class ObjComp {
2.     public static void main(String [] args ) {
3.       int result = 0;
4.       ObjComp oc = new ObjComp();
5.       Object o = oc;
6.
7.       if (o == oc)  result = 1;
8.       if (o != oc)  result = result + 10;
9.       if (o.equals(oc) )  result = result + 100;
10.      if (oc.equals(o) )  result = result + 1000;
11.
12.      System.out.println("result = " + result);
13.    }
14.  }
```

what is the result?

A. 1

B. 10

C. `101`

D. `1001`

E. `1101`

**19.** Which two statements are true about wrapper or String classes? (Choose two.)

A. If *x* and *y* refer to instances of different wrapper classes, then the fragment `x.equals(y)` will cause a compiler failure.

B. If *x* and *y* refer to instances of different wrapper classes, then *x* == *y* can sometimes be `true`.

C. If *x* and *y* are String references and if `x.equals(y)` is `true`, then *x* == *y* is `true`.

D. If *x*, *y*, and *z* refer to instances of wrapper classes and `x.equals(y)` is `true`, and `y.equals(z)` is `true`, then `z.equals(x)` will always be `true`.

E. If *x* and *y* are String references and *x* == *y* is `true`, then `y.equals(x)` will be `true`.

# SELF TEST ANSWERS

## Strings (Exam Objective 8.2)

1. ☑ **C.** After line 5 executes, both *s2* and *s3* refer to a String object that contains the value "def". When line 6 executes, a new String object is created with the value "ghi", to which *s2* refers. The reference variable *s3* still refers to the (immutable) String object with the value "def".
   ☒ **A, B, D, E,** and **F** are incorrect based on the logic described above.

2. ☑ **C.** Line 12 creates a new String object with the value "XYZ", but this new object is immediately lost because there is no reference to it. Line 13 creates a new String object referenced by *y*. This new String object has the value "xyz" because there was no "Y" in the String object referred to by *x*. Line 14 creates a new String object, appends "abc" to the value "xyz", and refers *y* to the result.
   ☒ **A, B, D, E,** and **F** are incorrect based on the logic described above.

3. ☑ **C.** Line 13 creates two, one referred to by *x* and the lost String "xyz". Line 14 creates one (for a total of three). Line 15 creates one more (for a total of four), the concatenated String referred to by *x* with a value of "xyzabc".
   ☒ **A, B,** and **D** are incorrect based on the logic described above.

4. ☑ **B.** Both `substring()` and `charAt()` methods are indexed with a zero-base, and `substring()` returns a String of length arg2 – arg1.
   ☒ **A, C, D, E,** and **F** are incorrect based on the logic described above.

5. ☑ **E.** In line 7 the code calls a StringBuffer method, `append()` on a String object.
   ☒ **A, B, C, D,** and **F** are incorrect based on the logic described above.

## Math (Exam Objective 8.1)

6. ☑ **B.** `Math.round()` adds .5 to the argument then performs a `floor()`. Since the code adds an additional .5 before `round()` is called, it's as if we are adding 1 then doing a `floor()`. The values that start out as integer values will in effect be incremented by 1 on the `round()` side but not on the `ceil()` side, and the noninteger values will end up equal.
   ☒ **A, C, D,** and **E** are incorrect based on the logic described above.

**7.** ☑ **A, B,** and **D.** The `max()` method is overloaded to take two arguments of type *int*, *long*, *float*, or *double*.
☒ **C** is incorrect because the `max()` method only takes two arguments.

**8.** ☑ **B** and **C.** The result range for `random()` is 0.0 to < 1.0; 1.0 is not in range.
☒ **A, D, E,** and **F** are incorrect based on the logic above.

**9.** ☑ **C.** The `sqrt()` method returns `NaN` (not a number) when it's argument is less than zero.
☒ **A, B, D,** and **E** are incorrect based on the logic described above.

**10.** ☑ **C.** The Math class' trigonometry methods expect their arguments to be in radians, not degrees. Line 5 can be decoded: "Convert 75 (degrees) into radians, then find the sine of that result."
☒ **A, B, D, E,** and **F** are incorrect based on the logic described above.

## Wrappers (Exam Objective 8.3)

**11.** ☑ **B** and **D. B** is incorrect because the `valueOf()` method returns an Integer object. **D** is incorrect because the `parseInt()` method takes a String.
☒ **A, C, E,** and **F** all represent valid syntax. Line 5 takes the String "345" to be octal number, and converts it to an integer value 229.

**12.** ☑ **C.** All of this code is legal, and line 5 creates a new String with a value of "42.5". Lines 6 and 7 convert the String to a *double* and then back again. Line 8 is fun—`Math.ceil()`'s argument expression is evaluated first. We invoke the `valueOf()` method that returns an anonymous Double object (with a value of 42.5). Then the `doubleValue()` method is called (invoked on the newly created Double object), and returns a *double* primitive (there and back again), with a value of (you guessed it) 42.5. The `ceil()` method converts this to 43.0, which is cast to an *int* and assigned to *x*. We know, we know, but stuff like this is on the exam.
☒ **A, B, D, E,** and **F** are incorrect based on the logic described above.

**13.** ☑ **E.** The compiler fails at line 10 because *b1* is a reference variable to a Boolean wrapper object, not a *boolean* primitive. Logical *boolean* tests can't be made on Boolean objects.
☒ **A, B, C, D,** and **F** are incorrect based on the logic described above.

**14.** ☑ **A, B,** and **D. A** won't compile because the `floatValue()` method is an instance method that takes no arguments. **B** won't compile because the `valueOf()` method returns a wrapper object. **D** won't compile because the `parseFloat()` method takes a String.
☒ **C, E,** and **F** are all legal (if not terribly useful) ways to return a primitive *float*.

**15.** ☑ **A** is correct. The `xxxValue()` methods convert any numeric wrapper object's value to any primitive type. When narrowing is necessary, significant bits are dropped and the results are difficult to calculate.
☒ **B**, **C**, **D**, **E**, and **F** are incorrect based on the logic described above.

## Equals() (Exam Objective 5.2)

**16.** ☑ **B.** Line 12 fails because == compares reference values, not object values. Line 13 succeeds because both String and primitive wrapper constructors resolve to the same value (except for the Character wrapper). Lines 14, 15, and 16 fail because the `equals()` method fails if the object classes being compared are different and not in the same tree hierarchy.
☒ **A**, **C**, **D**, **E**, and **F** are incorrect based on the logic described above.

**17.** ☑ **F.** Line 10 fails because *b1* and *b2* are two different objects. Lines 11 and 14 succeed because the Boolean String constructors are case insensitive. Lines 12 and 13 fail because `true` is not equal to `false`.
☒ **A**, **B**, **C**, **D**, and **E** are incorrect based on the logic described above.

**18.** ☑ **E.** Even though *o* and *oc* are reference variables of different types, they are both referring to the same object. This means that == will resolve to `true` and that the default `equals()` method will also resolve to `true`.
☒ **A**, **B**, **C**, and **D** are incorrect based on the logic described above.

**19.** ☑ **D** and **E. D** describes an example of the `equals()` method behaving transitively. By the way, *x*, *y*, and *z* will all be the same type of wrapper. **E** is true because *x* and *y* are referring to the same String object.
☒ **A** is incorrect—the fragment will compile. **B** is incorrect because *x* == *y* means that the two reference variables are referring to the same object. **C** will only be true if *x* and *y* refer to the same String. It is possible for *x* and *y* to refer to two different String objects with the same value.

# EXERCISE ANSWERS

## Exercise 6-1: Using the Math Class

The following code listing is an example of how you might have written code to complete the exercise:

```
class NumberInterrogation {
   public static void main(String [] argh) {
      double [] num = {10.5, -10.5, Math.PI, 0};
      for(int i=0;i<num.length;++i) {
         System.out.println("abs("+num[i]+")="+Math.abs(num[i]));
         System.out.println("ceil("+num[i]+")="+Math.ceil(num[i]));
         System.out.println("floor("+num[i]+")="+Math.floor(num[i]));
         System.out.println();
      }
   }
}
```

## Exercise 6-2: Rounding Random Numbers

The following code listing is an example of how you might have written code to complete the exercise:

```
class RandomRound {
   public static void main(String [] argh) {
      for(int i=0;i<10;++i) {
         double num = Math.random() * 100;
         System.out.print("The number " + num);
         System.out.println(" rounds to " + Math.round(num));
      }
   }
}
```