JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

3

Operators and Assignments

CERTIFICATION OBJECTIVES

- Java Operators
- Logical Operators
- Passing Variables into Methods

✓ Two-Minute Drill

Q&A Self Test

If you've got variables, you're going to modify them. You'll increment them, add them together, shift their bits, flip their bits, and compare one to another. In this chapter you'll learn how to do all that in Java. We'll end the chapter exploring the effect of passing variables of all types into methods. For an added bonus, you'll learn how to do things that you'll probably never use in the real world, but that will almost certainly be on the exam. After all, what fun would it be if you were tested only on things you already use?

CERTIFICATION OBJECTIVE

Java Operators (Exam Objective 5.1)

Determine the result of applying any operator (including assignment operators and instanceof) to operands of any type, class, scope, or accessibility, or any combination of these.

Java *operators* produce new values from one or more *operands* (just so we're all clear, the *operands* are things on the right or left side of the operator). The result of most operations is either a boolean or numeric value. And because you know by now that *Java is not C++*, you won't be surprised that Java operators can't be overloaded. There is, however, one operator that comes overloaded out of the box: If applied to a String, the + operator concatenates the right-hand operand to the operand on the left.

Stay awake. The operators and assignments portion of the exam is typically the one where exam takers see their lowest scores. We aren't naming names or anything, but even some of the exam *creators* (including one whose last name is a mountain range in California) have been known to get a few of these wrong.

Assignment Operators

Assigning a value to a variable seems straightforward enough; you simply assign the stuff on the right side of the = to the variable on the left. Well, sure, but don't expect to be tested on something like this:

```
x = 6;
```

No, you won't be tested on the *no-brainer* (technical term) assignments. You will, however, be tested on the trickier assignments involving complex expressions and

casting. We'll look at both primitive and reference variable assignments. But before we begin, let's back up and peek inside of a variable. What *is* a variable? How are the *variable* and its *value* related?

Variables are just bit holders, with a designated type. You can have an *int* holder, a *double* holder, a Button holder, and even a `String[]` holder. Within that holder is a bunch of bits representing the value. For primitives, the bits represent a numeric value (although we don't know what that bit pattern looks like for `boolean`, but we don't care). A `byte` with a value of 6, for example, means that the bit pattern in the variable (the *byte* holder) is 00000110, representing the 8 bits.

So the value of a *primitive* variable is clear, but what's inside an *object* holder? If you say

```
Button b = new Button();
```

what's inside the Button holder *b*? Is it the Button object? *No!* A variable referring to an object is just that—a *reference variable*. A reference variable bit holder contains bits representing *a way to get to the object*. We don't know what the format is; the way in which object references are stored is virtual-machine specific (it's a pointer to *something*, we just don't know what that something really is). All we can say for sure is that the variable's value is not the object, but rather a value representing *a specific object on the heap*. Or `null`. If the reference variable has not been assigned a value, or has been explicitly assigned a value of `null`, the variable holds bits representing—you guessed it—`null`. You can read

```
Button b = null;
```

as "The `Button` variable *b* is not referring to any object."

So now that we know a variable is just a little box o' bits, we can get on with the work of changing those bits. We'll look first at assigning values to primitives, and finish with assignments to reference variables.

Primitive Assignments

The equal (=) sign is used for *assigning* a value to a variable, and it's cleverly named the *assignment operator*. There are actually 12 assignment operators, but the other 11 are all combinations of the equal sign and other arithmetic operators, as shown in Table 3-1. These *compound assignment operators* have a couple of special properties we'll look at in this section.

TABLE 3-1 Compound Assignment Operators

=	*=	/=	%=
+=	-=	<<=	>>=
>>>=	&=	^=	|=

You can assign a primitive variable using a literal or the result of an expression. Take a look at the following:

```
int x = 7; // literal assignment
int y = x + 2; // assignment with an expression (including a literal)
int z = x * y; // assignment with an expression
```

The most important point to remember is that a literal integer (such as 7) is always implicitly an `int`. Thinking back to Chapter 1, you'll recall that an int is a 32-bit value. No big deal if you're assigning a value to an `int` or a `long` variable, but what if you're assigning to a *byte* variable? After all, a *byte*-sized holder can't hold as many bits as an *int*-sized holder. Here's where it gets weird. The following is legal,

```
byte b = 27;
```

but only because the compiler *automatically* narrows the literal value to a *byte*. In other words, *the compiler puts in the cast*. The preceding code is identical to the following:

```
byte b = (byte) 27; // Explicitly cast the int literal to a byte
```

It looks as though the compiler gives you a break, and let's you take a shortcut with assignments to integer variables smaller than an *int*. (Everything we're saying about *byte* applies equally to *char* and *short*, both of which are smaller than an *int*.) We're not actually at the weird part yet, by the way.

We know that a literal integer is always an *int*, but more importantly—the result of an expression involving anything *int*-sized or smaller is always an *int*. In other words, add two *bytes* together and you'll get an *int*—even if those two *bytes* are tiny. Multiply an *int* and a *short* and you'll get an *int*. Divide a *short* by a *byte* and you'll get…an *int*. OK, now we're at the weird part. Check this out:

```
byte b = 3; // No problem, 3 fits in a byte
byte c = 8; // No problem, 8 fits in a byte
byte d = b + c; // Should be no problem, sum of the two bytes
                // fits in a byte
```

The last line won't compile! You'll get the following error:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
    byte c = a + b;
                 ^
```

We tried to assign the sum of two *bytes* to a *byte* variable, the result of which (11) was definitely small enough to fit into a *byte*, but the compiler didn't care. It knew the rule about *int*-or-smaller expressions always resulting in an *int*. It would have compiled if we'd done the explicit cast:

```
byte c = (byte) (a + b);
```

Assigning Floating-Point Numbers Floating-point numbers have slightly different assignment behavior than integer types. We've already discussed this in Chapter 1, but we'll do another quick review here while we're on the subject. First, you must know that *every floating-point literal is implicitly a double (64 bits), not a float*. So the literal 2.3, for example, is considered a *double*. If you try to assign a *double* to a *float*, the compiler knows you don't have enough room in a 32-bit float container to hold the precision of a 64-bit *double*, and it lets you know. The following code looks good, but won't compile:

```
float f = 32.3;
```

You can see that 32.3 should fit just fine into a *float*-sized variable, but the compiler won't allow it. In order to assign a floating-point literal to a *float* variable, you must either cast the value or append an *f* to the end of the literal. The following assignments will compile:

```
float f = (float) 32.3;
float g = 32.3f;
float h = 32.3F;
```

Assigning a Literal That Is Too Large for the Variable We'll also get a compiler error if we try to assign a literal value that the compiler knows is too big to fit into the variable.

```
byte a = 128; // byte can only hold up to 127
```

The preceding code gives us this error:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
byte a = 128;
```

We can fix it with a cast:

```
byte a = (byte) 128;
```

But then what's the result? When you narrow a primitive, Java simply truncates the higher-order bits that won't fit. In other words, it loses all the bits to the left of the bits you're narrowing to.

Let's take a look at what happens in the preceding code. There, 128 is the bit pattern `10000000`. It takes a full 8 bits to represent 128. But because the literal 128 is an *int*, we actually get 32 bits, with the 128 living in the right-most (lower-order) 8 bits. So a literal 128 is actually

```
00000000000000000000000010000000
```

Take our word for it; there are 32 bits there.

To narrow the 32 bits representing 128, Java simply lops off the leftmost (higher-order) 24 bits. We're left with just the `10000000`. But remember that a *byte* is signed, with the leftmost bit representing the sign (and not part of the value of the variable). So we end up with a negative number (the 1 that used to represent 128 now represents the negative sign bit). Remember, to find out the value of a negative number using two's complement notation, you flip all of the bits and then add 1. Flipping the 8 zeroes give us: `01111111`, and adding 1 to that gives us `10000000`, or back to 128! And when we apply the sign bit, we end up with -128.

You must use an explicit cast to assign 128 to a *byte*, and the assignment leaves you with the value -128. A cast is nothing more than your way of saying to the compiler, "Trust me. I'm a professional. I take full responsibility for anything weird that happens when those top bits are chopped off."

That brings us to the compound assignment operators. The following will compile,

```
byte b = 3;
b += 7;  // No problem - adds 7 to b (result is 10)
```

and is equivalent to

```
byte b = 3;
b = (byte) (b + 7);  // Won't compile without the
                     // cast, since b + 7 results in an int
```

The compound assignment operator += let's you add to the value of *b*, without putting in an explicit cast.

Assigning One Primitive Variable to Another Primitive Variable

When you assign one primitive variable to another, *the contents of the right-hand variable are copied*, for example,

```
int a = 6;
int b = a;
```

The preceding code can be read as, "Assign the bit pattern for the number 6 to the *int* variable *a*. Then copy the bit pattern in *a*, and place the copy into variable *b*. So, both variables now hold a bit pattern for 6, but the two variables have no other relationship. We used the variable *a* only to copy its contents. At this point, *a* and *b* have identical contents (in other words, identical *values*), but if we change the contents of *a* or *b*, the other variable won't be affected."

Take a look at the following example:

```
class ValueTest {
   public static void main (String [] args) {
      int a = 10;  // Assign a value to a
      System.out.println("a = " + a);
      int b = a;
      b = 30;
      System.out.println("a = " + a + "after change to b");
   }
}
```

The output from this program is

```
%java ValueTest
a = 10
a = 10 after change to b
```

Notice the value of *a* stayed at 10. The key point to remember is that even after you assign *a* to *b*, *a* and *b* are not referring to the same place in memory. The *a* and *b* variables do not share a single value; they have identical *copies*.

Reference Variable Assignments

You can assign a newly created object to an object reference variable as follows:

```
Button b = new Button();
```

The preceding line does three key things:

- Makes a reference variable named *b*, of type Button
- Creates a new Button object on the heap
- Assigns the newly created Button object to the reference variable *b*

You can also assign `null` to an object reference variable, which simply means the variable is not referring to any object:

```
Button c = null;
```

The preceding line creates space for the Button *reference* variable (the bit holder for a reference value), but doesn't create an actual Button *object*.

You can also use a reference variable to refer to any object that is a subclass of the declared reference variable type, as follows:

```
public class Foo {
   public void doFooStuff() {
   }
}
public class Bar extends Foo {
   public void doBarStuff() { }
   }
}
class Test {
   public static void main (String [] args) {
      Foo reallyABar = new Bar();  // Legal because Bar is a subclass of Foo
      Bar reallyAFoo = new Foo(); // Illegal! Foo is not a subclass of Bar
   }
}
```

We'll look at the concept of reference variable assignments in much more detail in Chapter 5, so for now you just need to remember the rule that you can assign a

subclass of the declared type, but not a superclass of the declared type. But think about it…a Bar object is guaranteed to be able to do anything a Foo can do, so anyone with a Foo reference can invoke Foo methods even though the object is actually a Bar.

In the preceding code, we see that Foo has a method `doFooStuff()` that someone with a Foo reference might try to invoke. If the object referenced by the Foo variable is really a Foo, no problem. But it's also no problem if the object is a Bar, since Bar inherited the `doFooStuff()` method. You can't make it work in reverse, however. If a somebody has a Bar reference, they're going to invoke `doBarStuff()`, but if the object being referenced is actually a Foo, it won't know how to respond.

Assigning One Reference Variable to Another

With primitive variables, an assignment of one variable to another means the contents (bit pattern) of one variable are copied into another. Object reference variables work exactly the same way. The contents of a reference variable are a bit pattern, so if you assign reference variable *a* to reference variable *b*, the bit pattern in *a* is copied and the new copy is placed into *b*. If we assign an existing instance of an object to a new reference variable, then two reference variables will hold the same bit pattern—a bit pattern referring to a specific object on the heap. Look at the following code:

```
import java.awt.Dimension;
class ReferenceTest {
   public static void main (String [] args) {
      Dimension a = new Dimension(5,10);
      System.out.println("a.height = " + a.height);
      Dimension b = a;
      b.height = 30;
      System.out.println("a.height = " + a.height +
         "after change to b");
   }
}
```

In the preceding example, a Dimension object *a* is declared and initialized with a width of 5 and a height of 10. Next, Dimension *b* is declared, and assigned the value of *a*. At this point, both variables (*a* and *b)* hold identical values, because the contents of *a* were copied into *b*. There is still only one Dimension object—the one that both *a* and *b* refer to. Finally, the height property is changed using the *b*

reference. Now think for a minute: Is this going to change the height property of *a* as well? Let's see what the output will be:

```
%java ReferenceTest
a.height = 10
a.height = 30 after change to b
```

From this output, we can conclude that both variables refer to the *same* instance of the Dimension object. When we made a change to *b*, the height property was also changed for *a*.

One exception to the way object references are assigned is String. In Java, String objects are given special treatment. For one thing, *String objects are immutable*; you can't change the value of a String object. But it sure *looks* as though you can. Examine the following code:

```
class Strings {
    public static void main(String [] args) {
        String x = "Java";  // Assign a value to x
        String y = x;       // Now y and x refer to the same String object
        System.out.println("y string = " + y);
        x = x + " Bean";    // Now modify the object using the x reference
        System.out.println("y string = " + y);
    }
}
```

You might think String *y* will contain the characters *Java Bean* after the variable *x* is changed, because strings are objects. Let's see what the output is:

```
%java String
y string = Java
y string = Java
```

As you can see, even though *y* is a reference variable to the same object that *x* refers to, when we change *x* it doesn't change *y*! For any other object type, where two references refer to the same object, if either reference is used to modify the object, both references will see the change because there is still only a single object. But with a string, the VM creates a brand new String object every time we use the + operator to concatenate two strings, or any time we make any changes at all to a string. You need to understand what happens when you use a String reference variable to modify a string:

- A new string is created, leaving the original String object untouched.
- The reference used to modify the String (or rather, make a new String by modifying a copy of the original) is then assigned the brand new String object.

So when you say,

```
1. String s = "Fred";
2. String t = s;   // Now t and s refer to the same String object
3. t.toUpperCase();  // Invoke a String method that changes the String
```

you actually haven't changed the original String object created on line 1. When line 2 completes, both *t* and *s* reference the same String object. But when line 3 runs, rather than modifying the object referred to by t (which is the one and only String object up to this point), a brand new String object is created. And then abandoned. Because the new String isn't assigned to a String variable, the newly created String (which holds the string "FRED") is toast. So while two String objects were created in the preceding code, only one is actually referenced, and both *t* and *s* refer to it. The behavior of strings is extremely important in the exam, so we'll cover it in much more detail in Chapter 6.

Comparison Operators

Comparison operators always result in a boolean (`true` or `false`) value. This boolean value is most often used in an *if* test, as follows,

```
int x = 8;
if (x < 9) {
     // do something
}
```

but the resulting value can also be assigned directly to a boolean primitive:

```
class CompareTest {
   public static void main(String [] args) {
      boolean b = 100 > 99;
      System.out.println("The value of b is " + b);
   }
}
```

You have four comparison operators that can be used to compare any combination of integers, floating-point numbers, or characters:

- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

Let's look at some legal comparisons:

```
class GuessAnimal {
   public static void main(String [] args) {
      String animal = "unknown";
      int weight = 700;
      char sex = 'm';
      double colorWaveLength = 1.630;
      if (weight >= 500) animal = "elephant";
      if (colorWaveLength > 1.621) animal = "gray " + animal;
      if (sex <= 'f') animal = "female " + animal;
      System.out.println("The animal is a " + animal);
   }
}
```

In the preceding code, we are using a comparison between characters. It's also legal to compare a character primitive with any number (though it isn't great programming style). Running the preceding class will output the following:

```
%java GuessAnimal
The animal is a gray elephant
```

We mentioned that characters can be used in comparison operators. When comparing a character with a character, or a character with a number, Java will take the ASCII or Unicode value of the character as the numerical value, and compare the numbers.

instanceof Comparison

The `instanceof` operator is used for object reference variables only, and you can use it to check whether an object is of a particular type. By type, we mean class or interface type—in other words, if the object referred to by the variable on the left side of the operator passes the IS-A test for the class or interface type on the right side (Chapter 5 covers IS-A relationships in detail). Look at the following example:

```
public static void main (String [] args) {
   String s = new String("foo");
   if (s instanceof String) {
      System.out.print("s is a String");
   }
}
```

Even if the object being tested is not an actual instantiation of the class type on the right side of the operator, `instanceof` will still return true if the object being compared is assignment compatible with the type on the right. The following example demonstrates testing an object using `instanceof`, to see if it's an instance of one of its superclasses:

```
class A { }
class B extends A { }
 public static void main (String [] args) {
   B b = new B();
   if (b instanceof A) {
      System.out.print("b is an A");
   }
}
```

The preceding code shows that *b* is an *a*. So you can test an object reference against its own class type, or any of its superclasses. This means that *any* object reference will evaluate to true if you use the `instanceof` operator against type Object, as follows,

```
B b = new B();
if (b instanceof Object) {
   System.out.print("b is definitely an Object");
}
```

which prints

```
b is definitely an Object
```

You can use the `instanceof` operator on interface types as well:

```
interface Foo { }
class Bar implements Foo { }
class TestBar {
   public static void main (String [] args) {
      Bar b = new Bar()
      if ( b instanceof Bar) {
         System.out.println("b is a Bar");
      }
      if (b instanceof Foo) {
         System.out.println("b is a Foo");
      }
    }
}
```

Running the TestBar class proves that the Bar object referenced by *b* is both a Bar and a Foo:

```
b is a Bar
b is a Foo
```

exam Watch

Look for instanceof questions that test whether an object is an instance of an interface, when the object's class implements indirectly. An indirect implementation occurs when one of an object's superclasses implements an interface, but the actual class of the instance does not—for example,

```
interface Foo { }
class A implements Foo { }
class B extends A { }
```

Using the definitions above, if we instantiate an A and a B as follows,

```
A a = new A();
B b = new B();
```

the following are true:

```
a instanceof A
a instanceof Foo
b instanceof A
b instanceof B
b instanceof Foo  // Even though class B doesn't implement Foo
directly!
```

An object is said to be of a particular interface type (meaning it will pass the instanceof test) if any of the object's superclasses implement the interface.

In addition, it is legal to test whether a null object (or `null` itself) is an instance of a class. This will always result in false, of course. The following code demonstrates this:

```
class InstanceTest {
   public static void main(String [] args) {
      String a = null;
      boolean b = null instanceof String;
      boolean c = a instanceof String;
      System.out.println(b + " " + c);
   }
}
```

When this code is run, we get the following output:

```
false false
```

So even though variable *a* was defined as a String, the underlying object is null; therefore, `instanceof` returns a value of false when compared to the String class.

exam Watch

Remember that arrays are objects, even if the array is an array of primitives. Look for questions that might look like this:

```
int [] nums = new int[3];
if (nums instanceof Object) { } // result is true
```

An array is always an instance of Object. Any array.

Table 3-2 shows results from several instanceof comparisons. For this table, assume the following:

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

Equality Operators

Equality can be tested with the operators equals and not equals:

- == equals (also known as "equal to")
- != not equals (also known as "not equal to")

Equality operators compare two *things* and return a *boolean* value. Each individual comparison can involve two numbers (including *char*), two *boolean* values, or two

TABLE 3-2 Operands and Results Using `instanceof` Operator

First Operand (Reference Being Tested)	`instanceof` Operand (Type We're Comparing the Reference Against)	Result
null	Any Class or Interface type	false
Foo instance	Foo, Bar, Face, Object	true
Bar instance	Bar, Face, Object	true
Bar instance	Foo	false
Foo []	Foo, Bar, Face	false
Foo []	Object	true
Foo[1]	Foo, Bar, Face, Object	true

object reference variables. You can't compare incompatible types, however. What would it mean to ask if a *boolean* is equal to a *char*? Or if a Button is equal to a String array? (Exactly, nonsense, which is why you can't do it.) There are four different types of *things* that can be tested:

- Numbers
- Characters
- Boolean primitives
- Object reference variables

So what does == actually look at? The value in the variable—in other words, the bit pattern.

Equality for Primitives

Most programmers are familiar with comparing primitive values. The following code shows some equality tests on primitive variables:

```
class ComparePrimitives {
   public static void main(String [] args) {
      System.out.println("character 'a' == 'a'? " + ('a' == 'a'));
      System.out.println("character 'a' == 'b'? " + ('a' == 'b'));
      System.out.println("5 != 6? " + (5 != 6));
      System.out.println("5.0 == 5L? " + (5.0 == 5L));
      System.out.println("true == false? " + (true == false));
   }
}
```

This program produces the following output:

```
%java ComparePrimitives
character 'a' == 'a'? true
character 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true  // Compare a floating point to an int
true == false? false
```

As we can see, if a floating-point number is compared with an integer and the values are the same, the == operator returns *true* as expected.

exam Watch

Don't mistake = for == in a boolean expression. The following is legal:

```
1. boolean b = false;
2. if (b = true) {
3.    System.out.println("b is true");
4.} else {
5.  System.out.println("b is false");
6.}
```

Look carefully! You might be tempted to think the output is "b is false," but look at the boolean test in line 2. The boolean variable b is not being compared to true, it's being set to true, so line 3 executes and we get "b is true." Keeping in mind that the result of any assignment expression is the value of the variable following the assignment, you can see that in line 3, the result of the expression will be true—the value of (b = true). This substitution of = for == works only with boolean variables, since the if test can be done only on boolean expressions. Thus, the following does not compile:

```
7. int x = 1;
8. if (x = 0) { }
```

Because x is an integer (and not a boolean), the result of (x = 0) is 0 (the result of the assignment). Integers cannot be used where a boolean value is expected, so the code in line 8 won't work unless changed from an assignment (=) to an equality test (==) as follows:

```
if (x == 0) { }
```

Equality for Reference Variables

As we saw earlier, two reference variables can refer to the same object, as the following code snippet demonstrates:

```
Button a = new Button("Exit");
Button b = a;
```

After running this code, both variable *a* and variable *b* will refer to the same object (a Button with the label *Exit*). Reference variables can be tested to see if they refer to the same object by using the == operator. Remember, the == operator is looking at the bits in the variable, so for reference variables if the bits in both variables are identical, then both refer to the same object. Look at the following code:

```
import java.awt.Button;
class CompareReference {
   public static void main(String [] args) {
```

```
        Button a = new Button("Exit");
        Button b = new Button("Exit");
        Button c = a;
        System.out.println("Is reference a == b? " + (a == b));
        System.out.println("Is reference a == c? " + (a == c));
    }
}
```

This code creates three reference variables. The first two, *a* and *b*, are separate Button objects that happen to have the same label. The third reference variable, *c*, is initialized to refer to the same object that *a* refers to. When this program runs, the following output is produced:

```
Is reference a == b? false
Is reference a == c? true
```

This shows us that *a* and *c* reference the same instance of a Button. We'll take another look at the implications of testing object references for equality in Chapters 6 and 7, where we cover String comparison and the `equals()` *method* (as opposed to the equals *operator* we're looking at here).

Arithmetic Operators

We're sure you're familiar with the basic arithmetic operators.

- + addition
- – subtraction
- × multiplication
- / division

These can be used in the standard way:

```
class MathTest {
   public static void main (String [] args) {
      int x = 5 * 3;
      int y = x - 4;
      System.out.println("x - 4 is " +  y);  // Prints 11
   }
}
```

(Warning: if you don't know how to use the basic arithmetic operators, your fourth-grade teacher, Mrs. Beasley, should be hunted down and forced to take

the programmer's exam. That's assuming you actually ever *went* to your fourth-grade class.)

One operator you might not be as familiar with (and we won't hold Mrs. Beasley responsible) is the remainder operator, %. The remainder operator divides the left operand by the right operand, and the result is the remainder, as the following code demonstrates:

```
class MathTest {
   public static void main (String [] args) {
      int x = 15;
      int y = x % 4;
      System.out.println("The result of 15 % 4 is the remainder of
15 divided by 4. The remainder is " + y);
   }
}
```

Running class MathTest prints the following:

```
The result of 15 % 4 is the remainder of 15 divided by 4. The remainder is 3
```

You can also use a compound assignment operator (shown in Table 3-1) if the operation is being done to a single variable. The following demonstrates using the %= compound assignment operator:

```
class MathTest {
   public static void main (String [] args) {
      int x = 15;
      x %= 4; // same as x = x % 4;
      System.out.println("The remainder of 15 % 4 is " + x);
   }
}
```

You're expected to know what happens when you divide by zero. With integers, you'll get a runtime exception (ArithmeticException), but with floating-point numbers you won't. Floating-point numbers divided by zero return either positive infinity or negative infinity, depending on whether or not the zero is positive or negative! That's right, some floating-point operators can distinguish between positive and negative zero. Rules to remember are these:

- Dividing an integer by zero will violate an important law of thermodynamics, and cause an ArithmeticException (can't divide by zero).
- Using the remainder operator (%) will result in an ArithmeticException if the right operand is zero (can't divide by zero).

- Dividing a floating-point number by zero will *not* result in an ArithmeticException, and the universe will remain intact.
- Using the remainder operator on floating-point numbers, where the right operand is zero, will *not* result in an ArithmeticException.

String Concatenation Operator

The plus sign can also be used to concatenate two strings together, as we saw earlier (and we'll definitely see again):

```
String animal = "Grey " + "elephant";
```

String concatenation gets interesting when you combine numbers with Strings. Check out the following:

```
String a = "String";
int b = 3;
int c = 7;
System.out.println(a + b + c);
```

Will the + operator act as a plus sign when adding the *int* variables *b* + *c*? Or will the + operator treat 3 and 7 as characters, and concatenate them individually? Will the result be `String10` or `String37`? OK, you've had long enough to think about it. The result is

```
String37
```

The int values were simply treated as characters and glued on to the right side of the string. So we could read the previous code as:

"Start with String *a*, "String", and add the character 3 (the value of *b*) to it, to produce a new string "String3", and then add the character 7 (the value of *c*) to that, to produce a new string "String37", then print it out."

However, if you put parentheses around the two *int* variables, as follows,

```
System.out.println(a + (b + c));
```

you'll get

```
String10
```

Using parentheses causes the (*b* + *c*) to evaluate first, so the + operator functions as the addition operator, given that both operands are *int* values. The key point here

is that the left-hand operand is not a String. If it were, then the + operator would perform String concatenation. The previous code can be read as:

"Add the values of *b* + *c* together, then take the sum and convert it to a String and concatenate it with the String from variable *a*."

The rule to remember is

If either operand is a String, the + operator becomes a String concatenation operator. If both operands are numbers, the + operator is the addition operator.

You'll find that sometimes you might have trouble deciding whether, say, the left hand operator is a String or not. On the exam, don't expect it to always be obvious. (Actually, now that we think about it, don't expect it *ever* to be obvious.) Look at the following code:

```
System.out.println(x.foo() + 7);
```

You can't know how the + operator is being used until you find out what the `foo()` method returns! If `foo()` returns a `String`, then 7 is concatenated to the returned String. But if `foo()` returns a number, then the + operator is used to add 7 to the return value of `foo()`.

exam Watch

If you don't understand how String concatenation works, especially within a print statement, you could actually fail the exam even if you know the rest of the answer to the question! Because so many questions ask, "What is the result?", you need to know not only the result of the code running, but also how that result is printed. Although there will be at least a half-dozen questions directly testing your String knowledge, String concatenation shows up in other questions on virtually every objective, and if you get the concatenation wrong, you'll miss that question regardless of your ability to work out the rest of the code. Experiment! For example, you might see a line such as

```
int b = 2;
int c = 3;
System.out.println("" + b + c);
```

which prints

```
23
```

but if the print statement changes to

```
System.out.println(b + c);
```

then the result becomes

```
5.
```

Increment and Decrement

Java has two operators that will increment or decrement a variable by exactly one. These operators are composed of either two plus signs (++) or two minus signs (--):

- ++ increment (prefix and postfix)
- -- decrement (prefix and postfix)

The operator is placed either before (prefix) or after (postfix) a variable to change the value. Whether the operator comes before or after the operand can change the outcome of an expression. Examine the following:

```
1.  class MathTest {
2.     static int players = 0;
3.     public static void main (String [] args) {
4.        System.out.println("players online: " + players++);
5.        System.out.println("The value of players is " + players);
6.        System.out.println("The value of players is now " + ++players);
7.     }
8.  }
```

Notice that in the fourth line of the program the increment operator is *after* the variable *players*. That means we're using the postfix increment operator, which causes the variable *players* to be incremented by one *but only after the value of players is used in the expression*. When we run this program, it outputs the following:

```
%java MathTest
players online: 0
The value of players is 1
The value of players is now 2
```

Notice that when the variable is written to the screen, at first it says the value is 0. Because we used the *postfix* increment operator, the increment doesn't happen until *after* the *players* variable is used in the print statement. Get it? The *post* in postfix means *after*. The next line, line 5, doesn't increment *players;* it just outputs it to the screen, so the newly incremented value displayed is 1. Line 6 applies the *prefix* operator to players, which means the increment happens *before* the value of the variable is used (*pre* means *before*). So the output is 2.

Expect to see questions mixing the increment and decrement operators with other operators, as in the following example:

```
int x = 2;
int y = 3;
```

```
if ((y == x++) | (x < ++y)) {
   System.out.println("x = " + x + " y = " + y);
 }
```

The preceding code prints

```
x = 3 y = 4
```

You can read the code as

"If 3 is equal to 2 OR 3 < 4…"

The first expression compares *x* and *y*, and the result is *false*, because the increment on *x* doesn't happen until after the == test is made. Next, we increment *x*, so now *x* is 3. Then we check to see if *x* is less than *y*, but *we increment y before comparing it with x*! So the second logical test is (3 < 4). The result is *true*, so the print statement runs.

exam Watch

Look out for questions that use the increment or decrement operators on a final variable. Because final variables can't be changed, the increment and decrement operators can't be used with them, and any attempt to do so will result in a compiler error. The following code won't compile,

```
final int x = 5;
int y = x++;
```

and produces the error

```
Test.java:4: cannot assign a value to final variable x
int y = x++;
            ^
```

You can expect a violation like this to be buried deep in a complex piece of code. If you spot it, you know the code won't compile and you can move on without working through the rest of the code (unless, of course, you're into the sport of Extreme Test-Taking, and you want the running-out-of-time challenge).

As with String concatenation, the increment and decrement operators are used throughout the exam, even on questions that aren't trying to test your knowledge of how those operators work. You might see them in questions on *for* loops, exceptions, even threads. Be ready.

Shift Operators

The following are shift operators:

- >> right shift
- << left shift
- >>> unsigned right shift (also called *zero-filled right shift*)

exam Watch

The more obscure the topic, the more likely it will appear on the exam. Operators such as +, -, *, and / aren't likely to be tested for on the exam because they're so commonly used. Shift operators are rarely used by most programmers; therefore, they will most definitely be on the exam.

The shift operators shift the bits of a number to the right or left, producing a new number. Shift operators are used on integral numbers only (not floating-point numbers). To determine the result of a shift, you have to convert the number into binary. Let's look at an example of a bit shift:

```
8 >> 1;
```

First, we must convert this number to a binary representation:

```
0000 0000 0000 0000 0000 0000 0000 1000
```

An *int* is a 32-bit integer, so all 32 bits must be displayed. If we apply a bit shift of one to the right, using the >> operator, the new bit number is

```
0000 0000 0000 0000 0000 0000 0000 0100
```

Notice how the 1 bit moved over to the right, one place.

We can now convert this back to a decimal number (base 10) to get 4. The following code shows the complete example:

```
class BitShift {
   public static void main(String [] args) {
      int x = 8;
      System.out.println("Before shift x equals " + x);
      x = x >> 1;
      System.out.println("After shift x equals " + x);
   }
}
```

When we compile and run this program we get the following output:

```
%java BitShift
Before shift x equals 8
After shift x equals 4
```

As you can see, the results are exactly what we expected them to be. Shift operations can work on all integer numbers, regardless of the base they're displayed in (octal, decimal, or hexadecimal). The left shift works in exactly the same way, except all bits are shifted in the opposite direction. The following code uses a hexadecimal number to shift:

```
class BitShift {
   public static void main(String [] args) {
      int x = 0x80000000;
      System.out.println("Before shift x equals " + x);
      x = x << 1;
      System.out.println("After shift x equals " + x);
   }
}
```

To understand the preceding example, we'll convert the hexadecimal number to a bit number. Fortunately, it's pretty simple to convert from hexadecimal to bits. Each hex digit converts to a four-bit representation, as we can see here:

```
8    0    0    0    0    0    0    0
1000 0000 0000 0000 0000 0000 0000 0000
```

In the preceding example, the very leftmost bit represents the sign (positive or negative). When the leftmost bit is 1, the number is negative; and when it is 0, the number is positive. Running our program gives us the following:

```
%java BitShift
Before shift x equals -2147483648
After shift x equals 0
```

Shifting the bits one to the left moves the sign bit out where it simply drops off the left edge (it doesn't wrap around or anything like that) leaving us with 0 in the leftmost bit. What about the right side? What gets filled in on the right side as the previous rightmost bits move to the left? With the left shift operator, the right side is always filled with zeroes.

Then what about the left side of a right shift operation? When we shift to the right, what gets filled in on the left as the previous leftmost bit moves to the right? What takes its place? The answer depends on which of the two right shift operators we're using.

When using the right shift operator (>>) to shift the bits of a negative number, the sign bit gets shifted to the right, but the leftmost bits are filled in on the left with whatever the sign bit was. So the bottom line is that *with the right shift operator (>>), a negative number stays negative.* For example, let's use the hex number 0x80000000 again:

```
1000 0000 0000 0000 0000 0000 0000 0000
```

Now we'll shift the bits, using >>, one to the right:

```
1100 0000 0000 0000 0000 0000 0000 0000
```

As we can see, the sign bit is shifted to the right but (and this is important) the leftmost bit is filled with the original sign bit. Let's try some code that shifts it *four* to the right rather than just one:

```
class BitShift {
   public static void main(String [] args) {
      int x = 0x80000000;
      System.out.println("Before shift x equals " + x);
      x = x >> 4;
      System.out.println("After shift x equals " + x);
   }
}
```

In line 5 of this program, the number will be bit shifted four to the right. Running this program gives us the following output:

```
%java BitShift
Before shift x equals -2147483648
After shift x equals -134217728
```

The number now equals the following in bit representation:

```
1111 1000 0000 0000 0000 0000 0000 0000
```

Notice how the four new bits on the left have all been filled in with the original sign bit.

We can use a special shift operator if we don't want to keep the sign bit. This is the unsigned right shift operator >>>. Let's change the code slightly to use this operator:

```
class BitShift {
   public static void main(String [] args) {
      int x = 0x80000000;
      System.out.println("Before shift x equals " + x);
      x >>>= 4; //Assignment operator
      System.out.println("After shift x equals " + x);
   }
}
```

The output for this program is now the following:

```
%java BitShift
Before shift x equals -2147483648
After shift x equals 134217728
```

As we can see, the new number is positive because the negative bit wasn't kept. In bit representation, the old number is

```
1000 0000 0000 0000 0000 0000 0000 0000
```

and the new number is

```
0000 1000 0000 0000 0000 0000 0000 0000
```

Notice how the leftmost bits are filled in with zeroes, even though the original sign bit was a 1. That's why the unsigned right shift operator is often referred to as the "zero filled right shift operator." One important implication of using >>> vs. >> is that, except for a few special cases that we'll discuss next, *the result of an unsigned right shift is always positive, regardless of the original sign bit.*

You also need to know that all operands in a bit shift are promoted to at least an *int*. And what happens if you try to shift by more places than the number of bits in the number being shifted? For example, what happens if you try to shift an *int* by 33? The rule to remember is: the number of bits shifted is always going to be the right operand modulus the total number of bits for that primitive type. So for an *int*, that means you'll shift by the right operand modulus 32, and for a *long*, the right operand modulus 64. For example, if you try to shift an *int* by, say, 34, it looks like this,

```
int x = 2;
int y = x >> 34;
```

but because it's meaningless to shift by 34, since you don't even have that many bits, you actually end up shifting by 34 % 32 (we can use the remainder operator to figure this out), which leaves us with a remainder of 2. So the result is actually

```
int y = x >> 2;
```

Note that when using any of the shift operators, if you shift an *int* by any multiple of 32, or a *long* by any multiple of 64, the original value will not change.

exam Watch

You need to know what the bit shifts are actually doing in practical terms. A right shift operator is actually causing the number being shifted to be** divided **by 2 to the power of the number of bits to shift. For example, shifting** `x >> 4` **is exactly the same as saying** $x / 2^4$**. And** `x >> 8` **is exactly the same as** $x / 2^8$**. With the left shift operator, the result is exactly the same as** multiplying **the number being shifted by 2 to the power of the number of bits to shift. So shifting** `x << 3` **is the same as saying** $x * 2^3$**. One day, you** will **thank us for pointing this out. (We accept checks and chocolate!)

EXERCISE 3-1

Using Shift Operators

1. Try writing a class that takes an integer of 1, shifts the bit 31 to the left, then 31 to the right.
2. What number does this now represent?
3. What is the bit representation of the new number?

Bitwise Operators

The bitwise operators take two individual bit numbers, then use AND/OR to determine the result on a bit-by-bit basis. There are three bitwise operators:

- & AND
- | inclusive OR
- ^ exclusive OR

The & operator compares corresponding bits between two numbers. If both bits are 1, the final bit is also 1. If only one of the bits is 1, the resulting bit is 0.

Once again, for bitwise operations we must convert numbers to bit representations. Table 3-3 displays the truth table for each of these operators. The left side of the table displays the *x* and *y* values, and the right side shows the result of the operator on these two values.

Let's compare two numbers, 10 and 9, with the & operator:

```
1010 & 1001 = 1000
```

Try putting the second operand directly beneath the first, to make it easier to see the result. For the preceding comparison (10 and 9), you can look at it as

```
  1 0 1 0
&
  1 0 0 1
___________
  1 0 0 0
```

As we can see, only the first bit (8) is a 1 in both locations, hence the final number is 1000 in bit representation (or 8 in decimal). Let's see this in some code:

```
class Bitwise {
   public static void main(String [] args) {
      int x = 10 & 9; // 1010 and 1001
      System.out.println("1010 & 1001 = " + x);
   }
}
```

When we run this code, the following output is produced:

```
%java Bitwise
1010 & 1001 = 8
```

The | (OR) operator is different from the & (AND) operator when it compares corresponding bits. Whereas the & operator will set a resulting bit to 1 only if *both*

TABLE 3-3 Calculating Values from a Truth Table

X	Y	& (AND)	| (OR)	^ (XOR)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

operand bits in the same position are 1, the | operator will set the resulting bit to 1 if *either* (of both) of the bits is a 1. So, for the numbers 10 and 9, we get the following,

```
1010 | 1001 = 1011
```

which is easier to see as

```
   1 0 1 0
|
   1 0 0 1
_____________
   1 0 1 1
```

In this case because we have 1s in the 1, 2, and 8 bit slots, those bits are carried in to the result. This expression produces the number 11 (in decimal). Let's look at this in code:

```
class Bitwise {
   public static void main(String [] args) {
      int x = 10 | 9; // 1010 and 1001
      System.out.println("1010 | 1001 = " + x);
   }
}
```

When we run the preceding code, we receive the following:

```
%java Bitwise
1010 | 1001 = 11
```

The ^ (Exclusive OR, also known as XOR) operator compares two bits to see if they are different. If they *are* different, the result is a 1. Look at the numbers 10 and 5 in bit representation:

```
1010 ^ 0101 = 1111
```

As we can see, the result is 15 in decimal form. To see it a little more clearly:

```
    1 0 1 0
^
    0 1 0 1
_____________
    1 1 1 1
```

Now let's look at doing an XOR on 8 and13:

```
1000 ^ 1101 = 0101
```

The result is 5 in decimal form.

```
   1 0 0 0
^
   1 1 0 1
_____________
   0 1 0 1
```

Bitwise Complement Operator

The ~ operator is a flip-the-bits operator. It will change all 1s to 0s and vice versa. Look at the following code:

```
class Bitwise {
   public static void main(String [] args) {
      int x = 5;
      System.out.println("x is initially " + x);
      x = ~x;
      System.out.println("~x is equal to " + x);
   }
}
```

This program is changing every bit into its complement; thus, the output from this program is the following:

```
%java Bitwise
x is initially 5
~x is equal to -6
```

In bit representation, the conversion looks like this,

```
~0000 0000 0000 0000 0000 0000 0000 0101
```

and converts to

```
 1111 1111 1111 1111 1111 1111 1111 1010
```

Conditional Operator

The *conditional operator* is a ternary operator (it has three operands) and is used to evaluate boolean expressions, much like an *if* statement except instead of executing a block of code if the test is *true*, a conditional operator will assign a value to a variable. In other words, the goal of the conditional operator is to decide which of

two values to assign to a variable. A conditional operator is constructed using a ? (question mark) and a : (colon). The parentheses are optional. Its structure is as follows:

someVariable = (boolean expression) ? value to assign if true : value to assign if false

Let's take a look at a conditional operator in code:

```
class Salary {
   public static void main(String [] args) {
      int numOfPets = 3;
      String status = (numOfPets<4)?"Pet limit not exceeded":"too many pets";
      System.out.println("This pet status is " + status);
   }
}
```

You can read the preceding code as:

"Set `numOfPets` equal to 3. Next we're going to assign a `String` to the *status* variable. If *numOfPets* is less than 4, assign "Pet limit not exceeded" to the *status* variable; otherwise, assign "too many pets" to the *status* variable."

A conditional operator starts with a boolean operation, followed by two possible values for the variable to the left of the conditional operator. The first value (the one to the left of the colon) is assigned if the conditional (boolean) test is *true*, and the second value is assigned if the conditional test is *false*. You can even nest conditional operators into one statement.

```
class AssignmentOps {
   public static void main(String [] args) {
      int sizeOfYard = 10;
      int numOfPets = 3;
      String status = (numOfPets<4)?"Pet count OK"
            :(sizeOfYard > 8)? "Pet limit on the edge"
            :"too many pets";
      System.out.println("Pet status is " + status);
   }
}
```

Don't expect many questions using conditional operators, but you need to be able to spot them and respond correctly. Conditional operators are sometimes confused with assertion statements, so be certain you can tell the difference. Chapter 4 covers assertions in detail.

Primitive Casting

Casting lets you convert primitive values from one type to another. We looked at primitive casting earlier in this chapter, in the assignments section, but now we're going to take a deeper look. Object casting is covered in Chapter 5.

Casts can be *implicit* or *explicit*. An *implicit cast* means you don't have to write code for the cast; the conversion happens automatically. Typically, an implicit cast happens when you're doing a *widening* conversion. In other words, putting a smaller thing (say, a *byte*) into a bigger container (like an *int*). Remember those "possible loss of precision" compiler errors we saw in the assignments section? Those happened when you tried to put a larger thing (say, a *long*) into a smaller container (like a *short*). The large-value-into-small-container conversion is referred to as narrowing and requires an *explicit* cast, where you tell the compiler that you're aware of the danger and accept full responsibility. First we'll look at an implicit cast:

```
int a = 100;
long b = a; // Implicit cast, an int value always fits in a long
```

An explicit casts looks like this:

```
float a = 100.001;
int b = (int)a; // Explicit cast, a float can lose info as an int
```

Integer values may be assigned to a *double* variable without explicit casting, because any integer value can fit in a 64-bit `double`. The following line demonstrates this:

```
double d = 100L; // Implicit cast
```

In the preceding statement, a *double* is initialized with a *long* value (as denoted by the *L* after the numeric value). No cast is needed in this case because a *double* can hold every piece of information that a *long* can store. If, however, we want to assign a *double* value to an integer type, we're attempting a narrowing conversion and the compiler knows it:

```
class Casting {
   public static void main(String [] args) {
      int x = 3957.229; // illegal
   }
}
```

If we try to compile the preceding code, the following error is produced:

```
%javac Casting.java
Casting.java:3: Incompatible type for declaration. Explicit cast
needed to convert double to int.
      int x = 3957.229; // illegal
1 error
```

In the preceding code, a floating-point value is being assigned to an integer variable. Because an integer is not capable of storing decimal places, an error occurs. To make this work, we'll cast the floating-point number into an integer:

```
class Casting {
   public static void main(String [] args) {
      int x = (int)3957.229; // legal cast
      System.out.println("int x = " + x);
   }
}
```

When you a cast a floating-point number to an integer type, the value loses all the digits after the decimal. Running the preceding code will produce the following output:

```
%java Casting
int x = 3957
```

We can also cast a larger number type, such as a *long*, into a smaller number type, such as a *byte*. Look at the following:

```
class Casting {
   public static void main(String [] args) {
      long l = 56L;
      byte b = (byte)l;
      System.out.println("The byte is " + b);
   }
}
```

The preceding code will compile and run fine. But what happens if the *long* value is larger than 127 (the largest number a byte can store)? Let's modify the code and find out:

```
class Casting {
   public static void main(String [] args) {
      long l = 130L;
      byte b = (byte)l;
```

```
        System.out.println("The byte is " + b);
    }
}
```

The code compiles fine, and when we run it we get the following:

```
%java Casting
The byte is -126
```

You don't get a runtime error, even when the value being narrowed is too large for the type. The bits to the left of the lower 8 just…go away. As we saw in the assignments section, if the leftmost bit in the byte now happens to be a 1, the 1 is no longer part of the value and instead becomes the sign bit for the new byte.

EXERCISE 3-2

Casting Primitives

Create a `float` number type of any value, and assign it to a short using casting.

1. Declare a float variable: `float f = 234.56F;`
2. Assign the float to a short: `short s = (short)f;`

CERTIFICATION OBJECTIVE

Logical Operators (Exam Objective 5.3)

In an expression involving the operators &, |, && and ||, and variables of known values, state which operands are evaluated and the value of the expression.

There are four logical operators. Two you've seen before; the & and | bitwise operators can be used in boolean expressions. The other two we haven't yet covered, and are known as the *short-circuit* logical operators:

- && short-circuit AND
- || short-circuit OR

Short-Circuit Logical Operators

The && operator is similar to the & operator, except it evaluates only boolean values and can't be used as a bitwise operator. Remember, for an AND expression to be true, both operands must be true—for example,

```
if ((2 < 3) && (3 < 4)) { }
```

The preceding expression evaluates to *true* only because *both* operand one (2 < 3) and operand two (3 < 4) evaluate to *true*.

The short-circuit feature of the && operator is that *it doesn't waste its time on pointless evaluations*. A short-circuit && evaluates the left side of the operation first (operand one), and if operand one resolves to *false*, the && operator doesn't bother looking at the right side of the equation (operand two). The operator already knows that the complete expression can't possibly be true, since one operand has already proven to be *false*.

```
class Logical {
   public static void main(String [] args) {
      boolean b = true && false;
      System.out.println("boolean b = " + b);
   }
}
```

When we run the preceding code, we get

```
C:\Java Projects\BookTest>java Logical
boolean b = false
```

The || operator is similar to the && operator, except that it evaluates the left side first, this time looking for *true*. If the first operand in an OR operation is *true*, the result will be *true*, so the short-circuit || doesn't waste time looking at the right side of the equation. If the first operand is *false*, however, the short-circuit || has to evaluate the second operand to see if the result of the OR operation will be *true* or *false*. Pay close attention to the following example; you'll see quite a few questions like this on the exam:

```
1. class TestOR {
2.    public static void main (String [] args) {
3.     if ((isItSmall(3)) || (isItSmall(7))) {
4.        System.out.println("Result is true");
```

```
 5.     }
 6.     if ((isItSmall(6)) || (isItSmall(9))) {
 7.        System.out.println("Result is true");
 8.     }
 9.  }
10.
11.   public static boolean isItSmall(int i) {
12.      if (i < 5) {
13.         System.out.println("i less than 5");
14.         return true;
15.     } else {
16.      System.out.println("i greater than 5");
17.         return false;
18.      }
19.   }
20. }
```

What is the result?

```
[localhost:~/javatests] kathy% java TestOR
i less than 5
Result is true
i greater than 5
i greater than 5
```

Here's what happened when the `main()` method ran:

1. When we hit line 3, the first operand in the || expression (in other words, the left side of the || operation) is evaluated.
2. The `isItSmall(3)` method is invoked and prints `"i less than 5"`.
3. The `isItSmall(3)` method returns *true*.
4. Because the first operand in the || expression on line 3 is `true`, the || operator doesn't bother evaluating the second operand. So we never see the `"i greater than 5"` that would have printed had the second operand been evaluated (which would have invoked `isItSmall(7)`).
5. Line 6 is now evaluated, beginning with the first operand in the || expression on line 6.
6. The `isItSmall(6)` method is invoked and prints `"i greater than 5"`.
7. The `isItSmall(6)` method returns *false*.

8. Because the first operand in the || expression on line 6 is *false*, the || operator can't skip the second operand; there's still a chance the expression can be *true*, if the second operand evaluates to *true*.
9. The `isItSmall(9)` method is invoked and prints `"i greater than 5"`.
10. The `isItSmall(9)` method returns *false*, so the expression on line 6 is *false*, and thus line 7 never executes.

exam Watch

The || and && operators only work with boolean operands. The exam may try to fool you by using integers with these operators, so be on guard for questions such as,

```
if (5 && 6) { }
```

where it looks as though we're trying to do a bitwise AND on the bits representing the integers 5 and 6, but the code won't even compile.

Logical Operators (not Short-Circuit)

The bitwise operators, & and |, can also be used in logical expressions. But because they aren't the short-circuit operators, *they evaluate both sides of the expression, always!* They're inefficient. For example, even if the first operand (left side) in an & expression is *false*, the second operand will still be evaluated—*even though it's now impossible for the result to be true!* And the | is just as inefficient; if the first operand is *true*, it still plows ahead and evaluates the second operand *even when it* knows *the expression will be true.*

The rule to remember is

The short-circuit operators (&& and ||) can be used only in logical (not bitwise) expressions. The bitwise operators (& and |) can be used in both logical and bitwise expressions, but are rarely used in logical expressions because they're not efficient.

exam Watch

You'll find a lot of questions on the exam that use both the short-circuit and non-short-circuit logical operators. You'll have to know exactly which operands are evaluated and which are not, since the result will vary depending on whether the second operand in the expression is evaluated. The "Self Test" at the end of this chapter includes several logical operator questions similar to those on the exam.

Now that you have a better idea how operators work in Java, the following chart shows some operators in action:

SCENARIO & SOLUTION	
What is the result of (1 & 3)?	1
What is the result of (1 | 3)?	3
What is the result of (1 << 2)?	4
What is the resulting value of (new String("fred") instanceof Object)?	`true`

CERTIFICATION OBJECTIVE

Passing Variables into Methods (Exam Objective 5.4)

Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

Methods can be declared to take primitives and/or object references. You need to know how (or if) the *caller's* variable can be affected by the *called* method. The difference between object reference and primitive variables, when passed into methods, is huge and important. To understand this section, you'll need to be comfortable with the assignments section covered in the first part of this chapter.

Passing Object Reference Variables

When you pass an object variable into a method, you must keep in mind that you're passing the object *reference*, and not the actual *object* itself. Remember that a reference variable holds bits that represent (to the underlying VM) a way to get to a specific object in memory (on the heap). More importantly, you must remember that you aren't even passing the actual reference variable, but rather a *copy* of the reference variable. A copy of a variable means you get a copy of the bits in that variable, so when you pass a reference variable, you're passing a copy of the bits representing how to get to a specific object. In other words, both the caller and the called method will now have identical copies of the reference, and thus both will refer to the same exact (not a copy) object on the heap.

For this example, we'll use the Dimension class from the java.awt package:

```
 1. import java.awt.Dimension;
 2. class ReferenceTest {
 3.  public static void main (String [] args) {
 4.     Dimension d = new Dimension(5,10);
 5.     ReferenceTest rt = new ReferenceTest();
 6.     System.out.println("Before modify() d.height = " + d.height);
 7.     rt.modify(d);
 8.     System.out.println("After modify() d.height = " + d.height);
 9.  }
10.   void modify(Dimension dim) {
11.     dim.height = dim.height + 1;
12.     System.out.println("dim = " + dim.height);
13.  }
14. }
```

When we run this class, we can see that the `modify()` method was indeed able to modify the original (and only) Dimension object created on line 4.

```
C:\Java Projects\Reference>java ReferenceTest
Before modify() d.height = 10
dim = 11
After modify() d.height = 11
```

Notice when the Dimension object on line 4 is passed to the `modify()` method, any changes to the object that occur inside the method are being made to the object whose reference was passed. In the preceding example, reference variables *d* and *dim* both point to the same object.

Does Java Use Pass-By-Value Semantics?

If Java passes objects by passing the reference variable instead, does that mean Java uses *pass-by-reference* for objects? Not exactly, although you'll often hear and read that it does. Java is actually *pass-by-value* for *all* variables running within a single VM. Pass-by-value means pass-by-*variable*-value. And that means, *pass-by-copy-of-the-variable*!

It makes no difference if you're passing primitive or reference variables, you are always passing a copy of the bits in the variable. So for a primitive variable, you're passing a copy of the bits representing the value. For example, if you pass an `int` variable with the value of 3, you're passing a *copy* of the bits representing 3. The called method then gets its own copy of the value, to do with it what it likes.

And if you're passing an object reference variable, you're passing a *copy* of the bits representing the reference to an object. The called method then gets its own copy of the reference variable, to do with it what it likes. But because two identical reference variables refer to the exact same object, if the called method modifies the object (by invoking setter methods, for example), the caller will see that the *object* the caller's original variable refers to has also been changed. In the next section, we'll look at how the picture changes when we're talking about primitives.

The bottom line on pass-by-value: the called method can't change the caller's *variable*, although for object reference variables, the called method *can* change the object the variable referred to. What's the difference between changing the variable and changing the object? For object references, it means the called method can't reassign the caller's original reference variable and make it refer to a different object, or null. For example, in the following code,

```
void bar() {
   Foo f = new Foo();
   doStuff(f);
}

void doStuff(Foo g) {
   g = new Foo();
}
```

reassigning *g* does not reassign *f*! At the end of the `bar()` method, two Foo objects have been created, one referenced by the local variable *f* and one referenced by the local (argument variable) *g*. Because the `doStuff()` method has a *copy* of the reference variable, it has a way to get to the original Foo object, but *the `doStuff()` method does* not *have a way to get to the* `f` *reference variable.* So `doStuff()` can change what *f* refers to, but can't change the actual contents (bit pattern) of *f*.

Passing Primitive Variables

Let's look at what happens when a primitive variable is passed to a method:

```
class ReferenceTest {
   public static void main (String [] args) {
      int a = 1;
      ReferenceTest rt = new ReferenceTest();
      System.out.println("Before modify() a = " + a);
      rt.modify(a);
```

```
        System.out.println("After modify() a = " + a);
    }
    void modify(int number) {
        number = number + 1;
        System.out.println("number = " + number);
    }
}
```

In this simple program, the variable *a* is passed to a method called `modify()`, which increments the variable by 1. The resulting output looks like this:

```
C:\Java Projects\Reference>java ReferenceTest
Before modify() a = 1
number = 2
After modify() a = 1
```

Notice that *a* did not change after it was passed to the method. Remember, it was only a *copy* of *a* that was passed to the method. When a primitive variable is passed to a method, it is *passed by value*, which means pass-by-copy-of-the-bits-in-the-variable.

FROM THE CLASSROOM

The Shadowy World of Variables

Just when you think you've got it all figured out, you see a piece of code with variables not behaving the way you think they should. You might have stumbled into code with a shadowed variable. You can shadow a variable in several ways; we'll look just at the one most likely to trip you up—*hiding an instance variable by shadowing it with a local variable.*

Shadowing involves redeclaring a variable that's already been declared somewhere else. The effect of shadowing is to hide the previously declared variable in such a way that it may *look* as though you're using the hidden variable, but you're actually using the shadowing variable. You might find reasons to shadow a variable intentionally, but typically it happens by accident and causes hard-to-find bugs. On the exam, you can expect to see questions where shadowing plays a role.

FROM THE CLASSROOM

You can shadow an instance variable by declaring a local variable of the same name, either directly or as part of an argument as follows:

```
class Foo {
   static int size = 7;
   static void changeIt(int size) {
      size = size + 200;
      System.out.println("size in changeIt is " + size);
   }
   public static void main (String [] args) {
      Foo f = new Foo();
      System.out.println("size = " + size);
      changeIt(size);
      System.out.println("size after changeIt is " + size);
    }
}
```

The preceding code appears to change the *size* instance variable in the `changeIt()` method, but because `changeIt()` has a parameter named *size*, the local *size* variable is modified while the instance variable *size* is untouched. Running class Foo prints

```
%java Foo
size = 7
size in changeIt is 207
size after changeIt is 7
```

Things become more interesting when the shadowed variable is an object reference, rather than a primitive:

```
class Bar {
   int barNum = 28;
}
class Foo {
   Bar myBar = new Bar();
   void changeIt(Bar myBar) {
      myBar.barNum = 99;
```

FROM THE CLASSROOM

```
        System.out.println("myBar.barNum in changeIt is " + barNum);
        myBar = new Bar();
        myBar.barNum = 420;
        System.out.println("myBar.barNum in changeIt is now " + barNum);
    }
    public static void main (String [] args) {
        Foo f = new Foo();
        System.out.println("f.myBar.barNum is " + f.myBar.barNum);
        changeIt(f.myBar);
        System.out.println("myBar.barNum after changeIt is " + f.myBar.barNum);
     }
}
```

The preceding code prints out this:

```
f.myBar.barNum is 28
myBar.barNum in changeIt is 99
myBar.barNum in changeIt is now 420
f.myBar.barNum after changeIt is 99
```

You can see that the shadowing variable (the local parameter *myBar* in `changeIt()`) can still affect the *myBar* instance variable, because the *myBar* parameter receives a reference to the same Bar object. But when the local *myBar* is reassigned a new Bar object, which we then modify by changing its *barNum* value, Foo's original *myBar* instance variable is untouched.

CERTIFICATION SUMMARY

If you've studied this chapter diligently, and thought of nothing else except this chapter for the last 72 hours, you should have a firm grasp on Java operators. You should understand what equality means when tested with the == operator, and you know how primitives and objects behave when passed to a method. Let's review the highlights of what you've learned in this chapter.

To understand what a bit-shift operation is doing, you need to look at the number being shifted in its binary form. The left shift (<<) shifts all bits to the left, filling the right side with zeroes, and the right shift (>>) shifts all bits right, *filling in the left side with whatever the sign bit was*. The unsigned right shift (>>>) moves all bits to the right, but fills the left side with zeroes, regardless of the original sign bit. Thus, the result of an unsigned right shift is always a positive number, except that when using any of the shift operators, if you shift an *int* by any multiple of 32, or a *long* by any multiple of 64, the original value will not change.

The logical operators (&& and ||) can be used only to evaluate two boolean expressions. The bitwise operators (& and |) can be used on integral numbers to produce a resulting numeric value, or on boolean values to produce a resulting *boolean* value. The difference between && and & is that the && operator won't bother testing the right operand if the left evaluates to *false*, because the result of the && expression can never be *true*. The difference between || and | is that the || operator won't bother testing the right operand if the left evaluates to *true*, because the result is already known to be *true* at that point.

The == operator can be used to compare values of primitives, but it can also be used to determine whether two reference variables refer to the same object.

Although both objects and primitives are passed by value into a method, key differences exist between how they behave once passed. Objects are passed *by a copy of the reference value*, while primitives are passed *by a copy of the variable value*. This means that if an object is modified within a method, other code referring to that object will notice the change. Both the caller and called methods have identical copies of reference variables; therefore, they both refer to the exact same object in memory.

Be prepared for a lot of exam questions involving the topics from this chapter. Even within questions testing your knowledge of another objective, the code will frequently use operators, assignments, object and primitive passing, etc., so be on your toes for this topic, and take the "Self Test" seriously.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 3.

Java Operators (Sun Objective 5.1)

- ❑ The result of performing most operations is either a boolean or a numeric value.
- ❑ Variables are just bit holders with a designated type.
- ❑ A reference variable's bits represent a way to get to an object.
- ❑ An unassigned reference variable's bits represent *null*.
- ❑ There are 12 assignment operators: =, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=.
- ❑ Numeric expressions always result in at least an *int*-sized result—never smaller.
- ❑ Floating-point numbers are implicitly doubles (64 bits).
- ❑ Narrowing a primitive truncates the high-order bits.
- ❑ Two's complement means: flip all the bits, then add 1.
- ❑ Compound assignments (e.g. +=) perform an automatic cast.

Reference Variables

- ❑ When creating a new object, e.g., Button b = new Button();, three things happen:
  - ❑ Make a reference variable named *b*, of type Button
  - ❑ Create a new Button object
  - ❑ Refer the reference variable b to the Button object
- ❑ Reference variables can refer to subclasses of the declared type but not superclasses.

String Objects and References

- ❑ String *objects* are immutable, cannot be changed.

- ❑ When you use a String reference variable to modify a String:
  - ❑ A new string is created (the old string *is immutable*).
  - ❑ The reference variable refers to the new string.

Comparison Operators

- ❑ Comparison operators always result in a boolean value (`true` or `false`).
- ❑ There are four comparison operators: >, >=, <, <=.
- ❑ When comparing characters, Java uses the ASCII or Unicode value of the number as the numerical value.

instanceof Operator

- ❑ `instanceof` is for reference variables only, and checks for whether this object is of a particular type.
- ❑ The `instanceof` operator can be used only to test objects (or *null*) against class types that are in the same class hierarchy.
- ❑ For interfaces, an object is "of a type" if any of its superclasses implement the interface in question.

Equality Operators

- ❑ Four types of things can be tested: numbers, characters, booleans, reference variables.
- ❑ There are two equality operators: == and !=.

Arithmetic Operators

- ❑ There are four primary operators: add, subtract, multiply, and divide.
- ❑ The remainder operator returns the remainder of a division.
- ❑ When floating-point numbers are divided by zero, they return positive or negative infinity, except when the dividend is also zero, in which case you get NaN.
- ❑ When the remainder operator performs a floating-point divide by zero, it will not cause a runtime exception.

- When integers are divided by zero, a runtime ArithmeticException is thrown.
- When the remainder operator performs an integer divide by zero, a runtime ArithmeticException is thrown.

String Concatenation Operator

- If either operand is a String, the + operator concatenates the operands.
- If both operands are numeric, the + operator adds the operands.

Increment/Decrement Operators

- Prefix operator runs before the value is used in the expression.
- Postfix operator runs after the value is used in the expression.
- In any expression, both operands are fully evaluated before the operator is applied.
- Final variables cannot be incremented or decremented.

Shift Operators

- There are three shift operators: >>, <<, >>>; the first two are signed, the last is unsigned.
- Shift operators can only be used on integer types.
- Shift operators can work on all bases of integers (octal, decimal, or hexadecimal).
- Except for the unusual cases of shifting an *int* by a multiple of 32 or a *long* by a multiple of 64 (these shifts result in no change to the original values), bits are filled as follows:
  - << fills the right bits with zeros.
  - >> fills the left bits with whatever value the original sign bit (leftmost bit) held.
  - >>> fills the left bits with zeros (negative numbers will become positive).
- All bit shift operands are promoted to at least an int.
- For int shifts > 32 or long shifts > 64, the actual shift value is the remainder of the right operand / divided by 32 or 64, respectively.

Bitwise Operators

- ❑ There are three bitwise operators—&, ^, |—and a bitwise complement, operator ~.
- ❑ The & operator sets a bit to 1 if both operand's bits are set to 1.
- ❑ The ^ operator sets a bit to 1 if exactly one operand's bit is set to 1.
- ❑ The | operator sets a bit to 1 if at least one operand's bit is set to 1.
- ❑ The ~ operator reverses the value of every bit in the single operand.

Ternary (Conditional Operator)

- ❑ Returns one of two values based on whether a boolean expression is *true* or *false*.
- ❑ The value after the ? is the 'if *true* return'.
- ❑ The value after the : is the 'if *false* return'.

Casting

- ❑ Implicit casting (you write no code) happens when a widening conversion occurs.
- ❑ Explicit casting (you write the cast) is required when a narrowing conversion is desired.
- ❑ Casting a floating point to an integer type causes all digits to the right of the decimal point to be lost (truncated).
- ❑ Narrowing conversions can cause loss of data—the most significant bits (leftmost) can be lost.

Logical Operators (Sun Objective 5.3)

- ❑ There are four logical operators: &, |, &&, ||.
- ❑ Logical operators work with two expressions that must resolve to boolean values.
- ❑ The && and & operators return `true` only if both operands are *true*.
- ❑ The || and | operators return `true` if either or both operands are *true*.

- ❑ The && and || operators are known as short-circuit operators.
- ❑ The && operator does not evaluate the right operand if the left operand is *false*.
- ❑ The || does not evaluate the right operand if the left operand is *true*.
- ❑ The & and | operators always evaluate both operands.

Passing Variables into Methods (Sun Objective 5.4)

- ❑ Methods can take primitives and/or object references as arguments.
- ❑ Method arguments are always copies—of either primitive variables or reference variables.
- ❑ Method arguments are never actual objects (they can be references to objects).
- ❑ In practice, a primitive argument is a completely detached copy of the original primitive.
- ❑ In practice, a reference argument is another copy of a reference to the original object.

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.

Java Operators (Sun Objective 5.1)

1. Which two are equal? (Choose two.)

A. `32 / 4;`

B. `(8 >> 2) << 4;`

C. `2 ^ 5;`

D. `128 >>> 2;`

E. `(2 << 1) * (32 >> 3);`

F. `2 >> 5;`

2. Given the following,

```
1.  import java.awt.*;
2.  class Ticker extends Component {
3.     public static void main (String [] args) {
4.        Ticker t = new Ticker();
5.
6.     }
7.  }
```

which two of the following statements, inserted independently, could legally be inserted into line 5 of this code? (Choose two.)

A. `boolean test = (Component instanceof t);`

B. `boolean test = (t instanceof Ticker);`

C. `boolean test = t.instanceof(Ticker);`

D. `boolean test = (t instanceof Component);`

E. `boolean test = t.instanceof(Object);`

F. `boolean test = (t instanceof String);`

3. Given the following,

```
1.  class Equals {
2.      public static void main(String [] args) {
3.         int x = 100;
4.         double y = 100.1;
5.         boolean b = (x = y);
6.         System.out.println(b);
7.      }
8.  }
```

what is the result?

A. `true`

B. `false`

C. Compilation fails

D. An exception is thrown at runtime

4. Given the following,

```
1.  import java.awt.Button;
2.   class CompareReference {
3.      public static void main(String [] args) {
4.         float f = 42.0f;
5.         float [] f1 = new float[2];
6.         float [] f2 = new float[2];
7.         float [] f3 = f1;
8.         long x = 42;
9.         f1[0] = 42.0f;
10.     }
11.  }
```

which three statements are true? (Choose three.)

A. `f1 == f2`

B. `f1 == f3`

C. `f2 == f1[1]`

D. `x == f1[0]`

E. `f == f1[0]`

5. Given the following,

```
1.  class BitShift {
2.      public static void main(String [] args) {
```

```
3.         int x = 0x80000000;
4.         System.out.print(x + " and  ");
5.         x = x >>> 31;
6.         System.out.println(x);
7.      }
8.   }
```

what is the output from this program?

A. `-2147483648 and 1`

B. `0x80000000 and 0x00000001`

C. `-2147483648 and -1`

D. `1 and -2147483648`

E. None of the above

6. Given the following,

```
1.  class Bitwise {
2.      public static void main(String [] args) {
3.         int x = 11 & 9;
4.         int y = x ^ 3;
5.         System.out.println( y | 12 );
6.      }
7.  }
```

what is the result?

A. `0`

B. `7`

C. `8`

D. `14`

E. `15`

7. Which of the following are legal lines of code? (Choose all that apply.)

A. `int w = (int)888.8;`

B. `byte x = (byte)1000L;`

C. `long y = (byte)100;`

D. `byte z = (byte)100L;`

Logical Operators (Sun Objective 5.3)

8. Given the following,

```
1.   class Test {
2.     public static void main(String [] args) {
3.       int x= 0;
4.       int y= 0;
5.       for (int z = 0; z < 5; z++) {
6.          if (( ++x > 2 ) || (++y > 2)) {
7.             x++;
8.          }
9.       }
10.      System.out.println(x + " " + y);
11.    }
12.  }
```

what is the result?

A. 5 3

B. 8 2

C. 8 3

D. 8 5

E. 10 3

F. 10 5

9. Given the following,

```
1.   class Test {
2.     public static void main(String [] args) {
3.       int x= 0;
4.       int y= 0;
5.       for (int z = 0; z < 5; z++) {
6.          if (( ++x > 2 ) && (++y > 2)) {
7.             x++;
8.          }
9.       }
10.      System.out.println(x + " " + y);
11.    }
12.  }
```

What is the result?

A. 5 2

B. 5 3

C. 6 3

D. 6 4

E. 7 5

F. 8 5

10. Given the following,

```
1.   class SSBool {
2.     public static void main(String [] args) {
3.       boolean b1 = true;
4.       boolean b2 = false;
5.       boolean b3 = true;
6.       if ( b1 & b2 | b2 & b3 | b2 )
7.         System.out.print("ok ");
8.       if ( b1 & b2 | b2 & b3 | b2 | b1 )
9.         System.out.println("dokey");
10.    }
11.  }
```

what is the result?

A. `ok`

B. `dokey`

C. `ok dokey`

D. No output is produced

E. Compilation error

F. An exception is thrown at runtime

11. Given the following,

```
1.  class Test {
2.     public static void main(String [] args) {
3.        int x=20;
4.        String sup = (x<15)?"small":(x<22)?"tiny":"huge";
5.        System.out.println(sup);
6.     }
7.  }
```

what is the result of compiling and running this code?

A. `small`

B. `tiny`

C. `huge`

D. Compilation fails

12. Given the following,

```
1.   class BoolArray {
2.     boolean [] b = new boolean[3];
3.     int count = 0;
4.
5.     void set(boolean [] x, int i) {
6.       x[i] = true;
7.       ++count;
8.     }
9.
10.    public static void main(String [] args) {
11.      BoolArray ba = new BoolArray();
12.      ba.set(ba.b, 0);
13.      ba.set(ba.b, 2);
14.      ba.test();
15.    }
16.
17.    void test() {
18.      if ( b[0] && b[1] | b[2] )
19.        count++;
20.      if ( b[1] && b[(++count - 2)] )
21.        count += 7;
22.      System.out.println("count = " + count);
23.    }
24.  }
```

what is the result?

A. `count = 0`

B. `count = 2`

C. `count = 3`

D. `count = 4`

E. `count = 10`

F. `count = 11`

Passing Variables into Methods (Sun Objective 5.4)

13. Given the following,

```
1.   class Test {
2.      static int s;
3.
```

```
4.      public static void main(String [] args) {
5.        Test p = new Test();
6.        p.start();
7.        System.out.println(s);
8.      }
9.
10.     void start() {
11.       int x = 7;
12.       twice(x);
13.       System.out.print(x + " ");
14.     }
15.
16.     void twice(int x) {
17.       x = x*2;
18.       s = x;
19.     }
20.   }
```

what is the result?

A. 7 7

B. 7 14

C. 14 0

D. 14 14

E. Compilation fails

F. An exception is thrown at runtime

14. Given the following,

```
1.   class Test {
2.      public static void main(String [] args) {
3.        Test p = new Test();
4.        p.start();
5.      }
6.
7.      void start() {
8.        boolean b1 = false;
9.        boolean b2 = fix(b1);
10.       System.out.println(b1 + " " + b2);
11.     }
12.
13.     boolean fix(boolean b1) {
14.       b1 = true;
```

```
15.       return b1;
16.     }
17.   }
```

what is the result?

A. `true true`
B. `false true`
C. `true false`
D. `false false`
E. Compilation fails
F. An exception is thrown at runtime

15. Given the following,

```
1.   class PassS {
2.      public static void main(String [] args) {
3.        PassS p = new PassS();
4.        p.start();
5.      }
6.
7.      void start() {
8.        String s1 = "slip";
9.        String s2 = fix(s1);
10.       System.out.println(s1 + " " + s2);
11.     }
12.
13.     String fix(String s1) {
14.       s1 = s1 + "stream";
15.       System.out.print(s1 + " ");
16.       return "stream";
17.     }
18.   }
```

what is the result?

A. `slip stream`
B. `slipstream stream`
C. `stream slip stream`
D. `slipstream slip stream`
E. Compilation fails
F. An exception is thrown at runtime

16. Given the following,

```
1.   class SC2 {
2.     public static void main(String [] args) {
3.       SC2 s = new SC2();
4.       s.start();
5.     }
6.
7.     void start() {
8.       int a = 3;
9.       int b = 4;
10.      System.out.print(" " + 7 + 2 + " ");
11.      System.out.print(a + b);
12.      System.out.print(" " + a + b + " ");
13.      System.out.print(foo() + a + b + " ");
14.      System.out.println(a + b + foo());
15.    }
16.
17.    String foo() {
18.      return "foo";
19.    }
20.  }
```

what is the result?

A. `9 7 7 foo 7 7foo`

B. `72 34 34 foo34 34foo`

C. `9 7 7 foo34 34foo`

D. `72 7 34 foo34 7foo`

E. `9 34 34 foo34 34foo`

17. Given the following,

```
1.   class PassA {
2.      public static void main(String [] args) {
3.        PassA p = new PassA();
4.        p.start();
5.      }
6.
7.      void start() {
8.        long [] a1 = {3,4,5};
9.        long [] a2 = fix(a1);
10.       System.out.print(a1[0] + a1[1] + a1[2] + " ");
11.       System.out.println(a2[0] + a2[1] + a2[2]);
```

```
12.     }
13.
14.     long [] fix(long [] a3) {
15.       a3[1] = 7;
16.       return a3;
17.     }
18.  }
```

what is the result?

A. 12 15

B. 15 15

C. 3 4 5 3 7 5

D. 3 7 5 3 7 5

E. Compilation fails

F. An exception is thrown at runtime

18. Given the following,

```
1.   class Two {
2.     byte x;
3.   }
4.
5.   class Pass0 {
6.      public static void main(String [] args) {
7.        Pass0 p = new Pass0();
8.        p.start();
9.      }
10.
11.     void start() {
12.       Two t = new Two();
13.       System.out.print(t.x + " ");
14.       Two t2 = fix(t);
15.       System.out.println(t.x + " " + t2.x);
16.     }
17.
18.     Two fix(Two tt) {
19.       tt.x = 42;
20.       return tt;
21.     }
22.   }
```

what is the result?

A. `null null 42`
B. `0 0 42`
C. `0 42 42`
D. `0 0 0`
E. Compilation fails
F. An exception is thrown at runtime

SELF TEST ANSWERS

Java Operators (Sun Objective 5.1)

1. ☑ **B** and **D**. **B** and **D** both evaluate to 32. **B** is shifting bits right then left using the signed bit shifters >> and <<. **D** is shifting bits using the unsigned operator >>>, but since the beginning number is positive the sign is maintained.
   ☒ **A** evaluates to 8, **C** looks like 2 to the 5th power, but ^ is the Exclusive OR operator so **C** evaluates to 7. **E** evaluates to 16, and **F** evaluates to 0 (2 >> 5 is not 2 to the 5th).

2. ☑ **B** and **D**. **B** is correct because class type Ticker is part of the class hierarchy of *t*; therefore it is a legal use of the *instanceof* operator. **D** is also correct because Component is part of the hierarchy of *t,* because Ticker extends Component in line 2.
   ☒ **A** is incorrect because the syntax is wrong. A variable (or `null`) always appears before the *instanceof* operator, and a type appears after it. **C** and **E** are incorrect because the statement is used as a method, which is illegal. **F** is incorrect because the String class is not in the hierarchy of the *t* object.

3. ☑ **C.** The code will not compile because in line 5, the line will work only if we use (`x == y`) in the line. The == operator compares values to produce a `boolean`, whereas the = operator assigns a value to variables.
   ☒ **A, B,** and **D** are incorrect because the code does not get as far as compiling. If we corrected this code, the output would be *false*.

4. ☑ **B, D,** and **E. B** is correct because the reference variables *f1* and *f3* refer to the same array object. **D** is correct because it is legal to compare integer and floating-point types. **E** is correct because it is legal to compare a variable with an array element.
   ☒ **C** is incorrect because *f2* is an array object and *f1[1]* is an array element.

5. ☑ **A.** The >>> operator moves all bits to the right, zero filling the left bits. The bit transformation looks like this:

   Before: `1000 0000 0000 0000 0000 0000 0000 0000`
   After: `0000 0000 0000 0000 0000 0000 0000 0001`

   ☒ **C** is incorrect because the >>> operator zero fills the left bits, which in this case changes the sign of *x*, as shown. **B** is incorrect because the output method `print()` always displays integers in base 10. **D** is incorrect because this is the reverse order of the two output numbers. **E** is incorrect because there was a correct answer.

6. ☑ **D.** The & operator produces a 1 bit when both bits are 1. The result of the & operation is 9. The ^ operator produces a 1 bit when exactly one bit is 1; the result of this operation is 10. The | operator produces a 1 bit when at least one bit is 1; the result of this operation is 14.
☒ **A, B, C,** and **E,** are incorrect based on the program logic described above.

7. ☑ **A, B, C,** and **D. A** is correct because when a floating-point number (a *double* in this case) is cast to an *int*, it simply loses the digits after the decimal. **B** and **D** are correct because a *long* can be cast into a *byte*. If the *long* is over 127, it loses its most significant (leftmost) bits. **C** actually works, even though a cast is not necessary, because a *long* can store a *byte*.
☒ There are no incorrect answer choices.

Logical Operators (Sun Objective 5.3)

8. ☑ **B.** The first two iterations of the *for* loop both *x* and *y* are incremented. On the third iteration *x* is incremented, and for the first time becomes greater than 2. The short circuit `or` operator || keeps *y* from ever being incremented again and *x* is incremented twice on each of the last three iterations.
☒ **A, C, D, E,** and **F** are incorrect based on the program logic described above.

9. ☑ **C.** In the first two iterations *x* is incremented once and *y* is not because of the short circuit && operator. In the third and fourth iterations *x* and *y* are each incremented, and in the fifth iteration *x* is doubly incremented and *y* is incremented.
☒ **A, B, D, E,** and **F** are incorrect based on the program logic described above.

10. ☑ **B.** The & operator has a higher precedence than the | operator so that on line 6 *b1* and *b2* are evaluated together as are *b2* & *b3*. The final *b1* in line 8 is what causes that *if* test to be *true*.
☒ **A, C,** and **D** are incorrect based on the program logic described above.

11. ☑ **B.** This is an example of a nested ternary operator. The second evaluation `(x < 22)` is true, so the "tiny" value is assigned to `sup`.
☒ **A, C,** and **D** are incorrect based on the program logic described above.

12. ☑ **C.** The reference variables *b* and *x* both refer to the same `boolean` array. `Count` is incremented for each call to the `set()` method, and once again when the first *if* test is true. Because of the && short circuit operator, *count* is not incremented during the second *if* test.
☒ **A, B, D, E,** and **F** are incorrect based on the program logic described above.

Passing Variables into Methods (Sun Objective 5.4)

13. ☑ **B.** The `int` *x* in the `twice()` method is not the same `int` *x* as in the `start()` method. `Start()'s` *x* is not affected by the `twice()` method. The instance variable *s* is updated by `twice()`'s *x*, which is 14.
☒ **A, C,** and **D** are incorrect based on the program logic described above.

14. ☑ **B.** The `boolean` *b1* in the `fix()` method is a different `boolean` than the *b1* in the `start()` method. The *b1* in the `start()` method is not updated by the `fix()` method.
☒ **A, C, D, E,** and **F** are incorrect based on the program logic described above.

15. ☑ **D.** When the `fix()` method is first entered, `start()`'s *s1* and `fix()`'s *s1* reference variables both refer to the same String object (with a value of "slip"). `Fix()`'s *s1* is reassigned to a new object that is created when the concatenation occurs (this second String object has a value of "slipstream"). When the program returns to `start()`, another String object is created, referred to by *s2* and with a value of "stream".
☒ **A, B, C,** and **E** are incorrect based on the program logic described above.

16. ☑ **D.** Because all of these expressions use the + operator, there is no precedence to worry about and all of the expressions will be evaluated from left to right. If either operand being evaluated is a String, the + operator will concatenate the two operands; if both operands are numeric, the + operator will add the two operands.
☒ **A, B, C,** and **E** are incorrect based on the program logic described above.

17. ☑ **B.** The reference variables *a1* and *a3* refer to the same `long` array object. When the `[1]` element is updated in the `fix()` method, it is updating the array referred to by *a1*. The reference variable *a2* refers to the same array object.
☒ **A, C, D, E,** and **F** are incorrect based on the program logic described above.

18. ☑ **C.** In the `fix()` method, the reference variable *tt* refers to the same object (class Two) as the *t* reference variable. Updating *tt.x* in the `fix()` method updates *t.x* (they are one in the same object). Remember also that the instance variable *x* in the Two class is initialized to 0.
☒ **A, B, D, E,** and **F** are incorrect based on the program logic described above.

EXERCISE ANSWERS

Exercise 3-1: Using Shift Operators

The program should look something like the following:

```
class BitShift {
   public static void main(String [] args) {
      int x = 0x00000001; // or simply 1
      x <<= 31;
      x >>= 31;
      System.out.println("After shift x equals " + x);
   }
}
```

The number should now equal -1. In bits, this number is

```
1111 1111 1111 1111 1111 1111 1111 1111
```

Exercise 3-2: Casting Primitives

The program should look something like the following:

```
class Cast {
   public static void main(String [] args) {
      float f = 234.56F;
      short s = (short)f;
   }
}
```