

# Chapter 2

## Operators

---

### OCF EXAM OBJECTIVES COVERED IN THIS CHAPTER:


- **Handling Date, Time, Text, Numeric and Boolean Values**
    - Use primitives and wrapper classes. Evaluate arithmetic and boolean expressions, using the Math API and by applying precedence rules, type conversions, and casting.
- 

The previous chapter talked a lot about defining variables, but what can you do with a variable once it is created? This chapter introduces operators and shows how you can use them to combine existing variables and create new values. It shows you how to apply operators to various primitive data types, including introducing you to operators that can be applied to objects.

## Understanding Java Operators

Before we get into the fun stuff, let's cover a bit of terminology. A Java *operator* is a special symbol that can be applied to a set of variables, values, or literals—referred to as *operands*—and that returns a result. The term *operand*, which we use throughout this chapter, refers to the value or variable the operator is being applied to. [Figure 2.1](#) shows the anatomy of a Java operation.

The output of the operation is simply referred to as the *result*. [Figure 2.1](#) actually contains a second operation, with the assignment operator (=) being used to store the result in variable *c*.

 An image depicts the function of a Java operation. It reads output functions as `var c = a + b` where `var c` is the result assigned to `c`, `a` and `b` are operands, and the plus symbol is an operator.

**[FIGURE 2.1](#)** Java operation

We're sure you have been using the addition (+) and subtraction (-) operators since you were a little kid. Java supports many other operators that you need to know for the exam. While many should be review for you, some (such as the compound assignment operators) may be new to you.

## Types of Operators

Java supports three flavors of operators: unary, binary, and ternary. These types of operators can be applied to one, two, or three operands, respectively. For the exam, you need to know a specific subset of Java operators, how to apply them, and the order in which they should be applied.

Java operators are not necessarily evaluated from left-to-right order. In this following example, the second expression is actually evaluated from right to left, given the specific operators involved:

```
int cookies = 4;
double reward = 3 + 2 * --cookies;
System.out.print("Zoo animal receives: "+reward+" reward points");
```

In this example, you first decrement cookies to 3, then multiply the resulting value by 2, and finally add 3. The value then is automatically promoted from 9 to 9.0 and assigned to reward. The final values of reward and cookies are 9.0 and 3, respectively, with the following printed:

```
Zoo animal receives: 9.0 reward points
```

If you didn't follow that evaluation, don't worry. By the end of this chapter, solving problems like this should be second nature.

## Operator Precedence

When reading a book or a newspaper, some written languages are evaluated from left to right, while some are evaluated from right to left. In mathematics, certain operators can override other operators and be evaluated first. Determining which operators are evaluated in what order is referred to as *operator precedence*. In this manner, Java more closely follows the rules for mathematics. Consider the following expression:

```
var perimeter = 2 * height + 2 * length;
```

Let's apply some optional parentheses to demonstrate how the compiler evaluates this statement:

```
var perimeter = ((2 * height) + (2 * length));
```

The multiplication operator (\*) has a higher precedence than the addition operator (+), so the *height* and *length* are both multiplied by 2 before being added together. The assignment operator (=) has the lowest order of precedence, so the assignment to the *perimeter* variable is performed last.

Unless overridden with parentheses, Java operators follow *order of operation*, listed in [Table 2.1](#), by decreasing order of operator precedence. If two operators have the same

level of precedence, then Java guarantees left-to-right evaluation for most operators other than the ones marked in the table.

We recommend keeping [Table 2.1](#) handy throughout this chapter. For the exam, you need to memorize the order of precedence in this table. Note that you won't be tested on some operators, like the shift operators, although we recommend that you be aware of their existence.

**TABLE 2.1** Order of operator precedence

Operator	Symbols and examples	Evaluation
Post-unary operators	<i>expression++</i> , <i>expression--</i>	Left-to-right
Pre-unary operators	<i>++expression</i> , <i>--expression</i>	Left-to-right
Other unary operators	<i>~</i> , <i>!</i> , <i>~</i> , <i>+</i> , ( <i>type</i> )	Right-to-left
Cast	( <i>Type</i> ) <i>reference</i>	Right-to-left
Multiplication/division/modulus	<i>*</i> , <i>/</i> , <i>%</i>	Left-to-right
Addition/subtraction	<i>+</i> , <i>-</i>	Left-to-right
Shift operators	<i>&lt;&lt;</i> , <i>&gt;&gt;</i> , <i>&gt;&gt;&gt;</i>	Left-to-right
Relational operators	<i>&lt;</i> , <i>&gt;</i> , <i>&lt;=</i> , <i>&gt;=</i> , <i>instanceof</i>	Left-to-right
Equal to/not equal to	<i>==</i> , <i>!=</i>	Left-to-right
Logical AND	<i>&amp;</i>	Left-to-right
Logical exclusive OR	<i>^</i>	Left-to-right
Logical inclusive OR	<i> </i>	Left-to-right
Conditional AND	<i>&amp;&amp;</i>	Left-to-right
Conditional OR	<i>  </i>	Left-to-right

Operator	Symbols and examples	Evaluation
Ternary operators	<i>boolean expression ? expression1 : expression2</i>	Right-to-left
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=,  =, <<=, >>=, >>>=	Right-to-left
Arrow operator	->	Right-to-left

---

The arrow operator (->), sometimes called the arrow function or lambda operator, is a binary operator that represents a relationship between two operands. Although we won't cover the arrow operator in this chapter, you will see it used in switch expressions in [Chapter 3](#), “Making Decisions,” and in lambda expressions starting in [Chapter 8](#), “Lambdas and Functional Interfaces.”

---

## Applying Unary Operators

By definition, a *unary* operator is one that requires exactly one operand, or variable, to function. As shown in [Table 2.2](#), they often perform simple tasks, such as increasing a numeric variable by one or negating a boolean value.

**TABLE 2.2** Unary operators

Operator	Examples	Description
Logical complement	!a	Inverts a boolean's logical value
Bitwise complement	~b	Inverts all 0s and 1s in a number
Plus	+c	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
Negation or minus	-d	Indicates a literal number is negative or negates an expression
Increment	++e f++	Increments a value by 1
Decrement	--f h--	Decrements a value by 1
Cast	(String)i	Casts a value to a specific type

Even though [Table 2.2](#) includes the casting operator, we postpone discussing casting until the “Assigning Values” section later in this chapter, since that is where it is commonly used.

## Complement and Negation Operators

The *logical complement operator* (!) flips the value of a boolean expression. For example, if the value is true, it will be converted to false, and vice versa. To illustrate this, compare the outputs of the following statements:

```
boolean isAnimalAsleep = false;
System.out.print(isAnimalAsleep); // false
isAnimalAsleep = !isAnimalAsleep;
System.out.print(isAnimalAsleep); // true
```

Next, the *bitwise negation operator* (~) turns all the zeros into ones and vice versa. You can figure out the new value by negating the original and subtracting one. For example:

```
int number = 70;
int negated = ~number;
System.out.println(negated); // -71
System.out.println(~negated); // 70
```

Next, the *negation operator* (-) reverses the sign of a numeric expression, as shown in these statements:

```
double zooTemperature = 1.21;
System.out.println(zooTemperature); // 1.21
zooTemperature = -zooTemperature;
System.out.println(zooTemperature); // -1.21
zooTemperature = -(-zooTemperature);
System.out.println(zooTemperature); // -1.21
```

Notice that in the previous example we used parentheses, ( ), for the negation operator, -, to apply the negation twice. If we had instead written --, then it would have been interpreted as the decrement operator and printed -2.21. You will see more of that decrement operator shortly.

Based on the description, it might be obvious that some operators require the variable or expression they’re acting on to be of a specific type. For example, you cannot apply a negation operator (-) to a boolean expression, nor can you apply a logical complement operator (!) to a numeric expression. Be wary of questions on the exam that try to do this, as they cause the code to fail to compile. For example, none of the following lines of code will compile:

```
int pelican = !5;           // DOES NOT COMPILE
boolean penguin = -true;    // DOES NOT COMPILE
boolean parrot = ~true;     // DOES NOT COMPILE
boolean peacock = !0;       // DOES NOT COMPILE
```

The first statement will not compile because in Java you cannot perform a logical inversion of a numeric value. The second and third statements do not compile because you cannot numerically negate or complement a boolean value; you need to use the logical inverse operator. Finally, the last statement does not compile because you cannot take the logical complement of a numeric value, nor can you assign an integer to a boolean variable.

---

Keep an eye out for questions on the exam that use numeric values (such as 0 or 1) with boolean expressions. Unlike in some other programming languages, in Java, 1 and true are not related in any way, just as 0 and false are not related.

---

## Increment and Decrement Operators

Increment and decrement operators, (++) and (--), respectively, can be applied to numeric variables and have a high order of precedence compared to binary operators. In other words, they are often applied first in an expression.

Increment and decrement operators require special care because the order in which they are attached to their associated variable can make a difference in how an expression is processed. [Table 2.3](#) lists each of these operators.

**TABLE 2.3** Increment and decrement operators

Operator	Example	Description
Pre-increment	++w	Increases the value by 1 and returns the <i>new</i> value
Pre-decrement	--x	Decreases the value by 1 and returns the <i>new</i> value
Post-increment	y++	Increases the value by 1 and returns the <i>original</i> value
Post-decrement	z--	Decreases the value by 1 and returns the <i>original</i> value

The following code snippet illustrates this distinction:

```
int parkAttendance = 0;
System.out.println(parkAttendance);    // 0
System.out.println(++parkAttendance);  // 1
System.out.println(parkAttendance);    // 1
System.out.println(parkAttendance--);  // 1
System.out.println(parkAttendance);    // 0
```

The first pre-increment operator updates the value for parkAttendance and outputs the new value of 1. The next post-decrement operator also updates the value of

parkAttendance but outputs the value before the decrement occurs.

---

For the exam, it is critical that you know the difference between expressions like `parkAttendance++` and `++parkAttendance`. The increment and decrement operators will be in multiple questions, and confusion about which value is returned could cause you to lose a lot of points on the exam.

---

## Working with Binary Arithmetic Operators

Next, we move on to operators that take two operands, called *binary operators*. Binary operators are by far the most common operators in the Java language. They can be used to perform mathematical operations on variables, create logical expressions, and perform basic variable assignments. Binary operators are often combined in complex expressions with other binary operators; therefore, operator precedence is very important in evaluating expressions containing binary operators. In this section, we start with binary arithmetic operators; we expand to other binary operators in later sections.

### Arithmetic Operators

*Arithmetic operators* are those that operate on numeric values. They are shown in [Table 2.4](#).

**TABLE 2.4** Binary arithmetic operators

Operator	Example	Description
Addition	<code>a + b</code>	Adds two numeric values
Subtraction	<code>c - d</code>	Subtracts two numeric values
Multiplication	<code>e * f</code>	Multiplies two numeric values
Division	<code>g / h</code>	Divides one numeric value by another
Modulus	<code>i % j</code>	Returns the remainder after division of one numeric value by another

You should know all but modulus from early mathematics. If you don't know what modulus is, though, don't worry—we'll cover that shortly. Arithmetic operators also include the unary operators, `++` and `--`, which we covered already. As you may have noticed in [Table 2.1](#), the *multiplicative* operators (`*`, `/`, `%`) have a higher order of precedence than the *additive* operators (`+`, `-`). Take a look at the following expression:

```
int price = 2 * 5 + 3 * 4 - 8;
```

First, you evaluate the  $2 * 5$  and  $3 * 4$ , which reduces the expression to this:

```
int price = 10 + 12 - 8;
```

Then, you evaluate the remaining terms in left-to-right order, resulting in a value of price being 14. Make sure you understand why the result is 14 because you will likely see this kind of operator precedence question on the exam.

---

All of the arithmetic operators may be applied to any Java primitives, with the exception of boolean. Furthermore, only the addition operators `+` and `+=` may be applied to `String` values, which results in `String` concatenation. You will learn more about these operators and how they apply to `String` values in [Chapter 4](#), “Core APIs.”

---

## Adding Parentheses

You might have noticed we said “Unless overridden with parentheses” prior to presenting [Table 2.1](#) on operator precedence. That’s because you can change the order of operation explicitly by wrapping parentheses around the sections you want evaluated first.

## Changing the Order of Operation

Let’s return to the previous price example. The following code snippet contains the same values and operators, in the same order, but with two sets of parentheses added:

```
int price = 2 * ((5 + 3) * 4 - 8);
```

This time you would evaluate the addition operator  $5 + 3$ , which reduces the expression to the following:

```
int price = 2 * (8 * 4 - 8);
```

You can further reduce this expression by multiplying the first two values within the parentheses.

```
int price = 2 * (32 - 8);
```

Next, you subtract the values within the parentheses before applying terms outside the parentheses.

```
int price = 2 * 24;
```

Finally, you would multiply the result by 2, resulting in a value of 48 for price.



Parentheses can appear in nearly any question on the exam involving numeric values, so make sure you understand how they are changing the order of operation when you see them.

---

When you encounter code in your professional career in which you are not sure about the order of operation, feel free to add optional parentheses. While often not required, they can improve readability, especially as you'll see with ternary operators.

---

### Verifying Parentheses Syntax

When working with parentheses, you need to make sure they are always valid and balanced. Consider the following examples:

```
long pigeon = 1 + ((3 * 5) / 3;           // DOES NOT COMPILE
int blueJay = (9 + 2) + 3) / (2 * 4;      // DOES NOT COMPILE
```

The first example does not compile because the parentheses are not balanced. There is a left parenthesis with no matching right parenthesis. The second example has an equal number of left and right parentheses, but they are not balanced properly. When reading from left to right, a new right parenthesis must match a previous left parenthesis. Likewise, all left parentheses must be closed by right parentheses before the end of the expression.

Let's try another example:

```
short robin = 3 + [(4 * 2) + 4];          // DOES NOT COMPILE
```

This example does not compile because Java, unlike some other programming languages, does not allow brackets, `[]`, to be used in place of parentheses. If you replace the brackets with parentheses, the previous example will compile just fine.

### Division and Modulus Operators

As we said earlier, the modulus operator, `%`, may be new to you. The modulus operator, sometimes called the *remainder operator*, is simply the remainder when two numbers are divided. For example, 9 divided by 3 divides evenly and has no remainder; therefore, the result of `9 % 3` is 0. On the other hand, 11 divided by 3 does not divide evenly; therefore, the result of `11 % 3` is 2.

The following examples illustrate this distinction:

```
System.out.println(9 / 3);    // 3
System.out.println(9 % 3);    // 0
```

```
System.out.println(10 / 3); // 3
System.out.println(10 % 3); // 1

System.out.println(11 / 3); // 3
System.out.println(11 % 3); // 2

System.out.println(12 / 3); // 4
System.out.println(12 % 3); // 0
```

As you can see, the division results increase only when the value on the left side goes from 11 to 12, whereas the modulus remainder value increases by 1 each time the left side is increased until it wraps around to zero. For a given divisor  $y$ , the modulus operation results in a value between 0 and  $(y - 1)$  for positive dividends, or 0, 1, 2 in this example.

Be sure to understand the difference between arithmetic division and modulus. For integer values, division results in the floor value of the nearest integer that fulfills the operation, whereas modulus is the remainder value. If you hear the phrase *floor value*, it just means the value without anything after the decimal point. For example, the floor value is 4 for each of the values 4.0, 4.5, and 4.9999999. Unlike rounding, which we'll cover in [Chapter 4](#), you just take the value before the decimal point, regardless of what is after the decimal point.

You can also use modulus with negative numbers. If the divisor is negative, then the negative sign is ignored. Negative values do change the behavior of modulus when applied to the dividend, though. For example, if the divisor is 5, then the modulus value of a negative number is between -4 and 0. The following examples show how this works:

```
System.out.println(2 % 5); // 2
System.out.println(7 % 5); // 2
System.out.println(2 % -5); // 2
System.out.println(7 % -5); // 2

System.out.println(-2 % 5); // -2
System.out.println(-7 % 5); // -2
System.out.println(-2 % -5); // -2
System.out.println(-7 % -5); // -2
```

---

The modulus operation may also be applied to floating-point numbers although that is out of scope for the exam.

---

## Numeric Promotion

Now that you understand the basics of arithmetic operators, it is vital to talk about primitive *numeric promotion*, as Java may do things that seem unusual to you at first. As we showed in [Chapter 1](#), “Building Blocks,” each primitive numeric type has a bit-length.

You don't need to know the exact size of these types for the exam, but you should know which are bigger than others. For example, you should know that a `long` takes up more space than an `int`, which in turn takes up more space than a `short`, and so on.

You need to memorize certain rules that Java will follow when applying operators to data types.

### **Numeric Promotion Rules**

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
3. Smaller data types, namely, `byte`, `short`, and `char`, are first promoted to `int` any time they're used with a Java binary arithmetic operator with a variable (as opposed to a value), even if neither of the operands is `int`.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

The last two rules are the ones most people have trouble with and the ones likely to trip you up on the exam. For the third rule, note that unary increment/decrement operators are excluded. For example, applying `++` to a `short` value results in a `short` value.

Let's tackle some examples for illustrative purposes.

- What is the data type of `x * y`?

```
int x = 1;
long y = 33;
var z = x * y;
```

In this case, we follow the first rule. Since one of the values is `int` and the other is `long` and since `long` is larger than `int`, the `int` value `x` is first promoted to a `long`. The result `z` is then a `long` value.

- What is the data type of `x + y`?

```
double x = 39.21;
float y = 2.1;
var z = x + y;
```

This is actually a trick question, as the second line does not compile! As you may remember from [Chapter 1](#), floating-point literals are assumed to be `double` unless postfixed with an `f`, as in `2.1f`. If the value of `y` was set properly to `2.1f`, then the promotion would be similar to the previous example, with both operands being promoted to a `double`, and the result `z` would be a `double` value.

- What is the data type of `x * y`?

```
short x = 10;  
short y = 3;  
var z = x * y;
```

On the last line, we must apply the third rule: that `x` and `y` will both be promoted to `int` before the binary multiplication operation, resulting in an output of type `int`. If you were to try to assign the value to a `short` variable `z` without casting, then the code would not compile. Pay close attention to the fact that the resulting output is not a `short`, as we'll come back to this example in the upcoming "Assigning Values" section.

- What is the data type of `w * x / y`?

```
short w = 14;  
float x = 13;  
double y = 30;  
var z = w * x / y;
```

In this case, we must apply all of the rules. First, `w` will automatically be promoted to `int` solely because it is a `short` and is being used in an arithmetic binary operation. The promoted `w` value will then be automatically promoted to a `float` so that it can be multiplied with `x`. The result of `w * x` will then be automatically promoted to a `double` so that it can be divided by `y`, resulting in a `double` value.

When working with arithmetic operators in Java, you should always be aware of the data type of variables, intermediate values, and resulting values. You should apply operator precedence and parentheses and work outward, promoting data types along the way. In the next section, we'll discuss the intricacies of assigning these values to variables of a particular type.

## Assigning Values

Compilation errors from assignment operators are often overlooked on the exam, in part because of how subtle these errors can be. To be successful with the assignment operators, you should be fluent in understanding how the compiler handles numeric promotion and when casting is required. Being able to spot these issues is critical to passing the exam, as assignment operators appear in nearly every question with a code snippet.

### Assignment Operator

An *assignment operator* is a binary operator that modifies, or *assigns*, the variable on the left side of the operator with the result of the value on the right side of the equation. Unlike most other Java operators, the assignment operator is evaluated from right to left.

The simplest assignment operator is the (=) assignment, which you have seen already:

```
int herd = 1;
```

This statement assigns the herd variable the value of 1.

Java will automatically promote from smaller to larger data types, as you saw in the previous section on arithmetic operators, but it will throw a compiler exception if it detects that you are trying to convert from larger to smaller data types without casting. [Table 2.5](#) lists the first assignment operator that you need to know for the exam. We present additional assignment operators later in this section.

**TABLE 2.5** Simple assignment operator

Operator	Example	Description
Assignment	int a = 50;	Assigns the value on the right to the variable on the left

## Casting Values

Seems easy so far, right? Well, we can't really talk about the assignment operator in detail until we've covered casting. *Casting* is a unary operation where one data type is explicitly interpreted as another data type. Casting is optional and unnecessary when converting to a larger or widening data type, but it is required when converting to a smaller or narrowing data type. Without casting, the compiler will generate an error when trying to put a larger data type inside a smaller one.

Casting is performed by placing the data type, enclosed in parentheses, to the left of the value you want to cast. Here are some examples of casting:

```
int fur = (int)5;
int hair = (short) 2;
String type = (String) "Bird";
short tail = (short)(4 + 10);
long feathers = 10(long); // DOES NOT COMPILE
```

Spaces between the cast and the value are optional. As shown in the second-to-last example, it is common for the right side to also be in parentheses. Since casting is a unary operation, it would only be applied to the 4 if we didn't enclose 4 + 10 in parentheses. The last example does not compile because the type is on the wrong side of the value.

On the one hand, it is convenient that the compiler automatically casts smaller data types to larger ones. On the other hand, it makes for great exam questions when they do the opposite to see whether you are paying attention. See if you can figure out why none of the following lines of code compiles:

```
float egg = 2.0 / 9;           // DOES NOT COMPILE
int tadpole = (int)5 * 2L;     // DOES NOT COMPILE
short frog = 3 - 2.0;          // DOES NOT COMPILE
```

All of these examples involve putting a larger value into a smaller data type. Don't worry if you don't follow this quite yet; we cover more examples like this shortly.

In this chapter, casting is primarily concerned with converting numeric data types into other data types. As you will see in later chapters, casting can also be applied to objects and references. In those cases, though, no conversion is performed. Put simply, casting a numeric value may change the data type, while casting an object only changes the reference to the object, not the object itself.

## Reviewing Primitive Assignments

See if you can figure out why each of the following lines does not compile:

```
int fish = 1.0;                // DOES NOT COMPILE
short bird = 1921222;          // DOES NOT COMPILE
int mammal = 9f;               // DOES NOT COMPILE
long reptile = 192_301_398_193_810_323; // DOES NOT COMPILE
```

The first statement does not compile because you are trying to assign a double `1.0` to an integer value. Even though the value is a mathematic integer, by adding `.0`, you're instructing the compiler to treat it as a double. The second statement does not compile because the literal value `1921222` is outside the range of `short`, and the compiler detects this. The third statement does not compile because the `f` added to the end of the number instructs the compiler to treat the number as a floating-point value, but the assignment is to an `int`. Finally, the last statement does not compile because Java interprets the literal as an `int` and notices that the value is larger than `int` allows. The literal would need a postfix `L` or `l` to be considered a `long`.

## Applying Casting

We can fix three of the previous examples by casting the results to a smaller data type. Remember, casting primitives is required any time you are going from a larger numerical data type to a smaller numerical data type, or converting from a floating-point number to an integral value.

```
int fish = (int)1.0;
short bird = (short)1921222; // Stored as 20678
int mammal = (int)9f;
```

What about applying casting to an earlier example?

```
long reptile = (long)192301398193810323; // DOES NOT COMPILE
```

This still does not compile because the value is first interpreted as an `int` by the compiler and is out of range. The following fixes this code without requiring casting:

```
long reptile = 192301398193810323L;
```

---

## Real World Scenario

### Overflow and Underflow

The expressions in the previous example now compile, although there's a cost. The second value, 1,921,222, is too large to be stored as a `short`, so numeric overflow occurs, and it becomes 20,678. *Overflow* is when a number is so large that it will no longer fit within the data type, so the system “wraps around” to the lowest negative value and counts up from there, similar to how modulus arithmetic works. There's also an analogous *underflow*, when the number is too low to fit in the data type, such as storing -200 in a byte field.

This is beyond the scope of the exam but something to be careful of in your own code. For example, the following statement outputs a negative number:

```
System.out.print(2147483647+1); // -2147483648
```

Since 2147483647 is the maximum `int` value, adding any strictly positive value to it will cause it to wrap to the smallest negative number.

---

Let's return to a similar example from the “Numeric Promotion” section earlier in the chapter.

```
short mouse = 10;  
short hamster = 3;  
short copybara = mouse * hamster; // DOES NOT COMPILE
```

Based on everything you have learned up until now about numeric promotion and casting, do you understand why the last line of this statement will not compile? As you may remember, `short` values are automatically promoted to `int` when applying any arithmetic operator, with the resulting value being of type `int`. Trying to assign a `short` variable with an `int` value results in a compiler error, as Java thinks you are trying to implicitly convert from a larger data type to a smaller one.

We can fix this expression by casting, as there are times that you may want to override the compiler's default behavior. In this example, we know the result of `10 * 3` is 30, which can easily fit into a `short` variable, so we can apply casting to convert the result back to a `short`:

```
short mouse = 10;
short hamster = 3;
short capybara = (short)(mouse * hamster);
```

By casting a larger value into a smaller data type, you instruct the compiler to ignore its default behavior. In other words, you are telling the compiler that you have taken additional steps to prevent overflow or underflow. It is also possible that in your particular application and scenario, overflow or underflow would result in acceptable values.

Last but not least, casting can appear anywhere in an expression, not just on the assignment. For example, let's take a look at a modified form of the previous example:

```
short mouse = 10;
short hamster = 3;
short capybara = (short)mouse * hamster;           // DOES NOT COMPILE
```

So, what's happening on the last line? Well, remember when we said casting was a unary operation? That means the cast in the last line is applied to `mouse`, and `mouse` alone. After the cast is complete, both operands are promoted to `int` since they are used with the binary multiplication operator (`*`), making the result an `int` and causing a compiler error.

What if we changed the last line to the following?

```
short capybara = 1 + (short)(mouse * hamster);    // DOES NOT COMPILE
```

In the example, casting is performed successfully, but the resulting value is automatically promoted to `int` because it is used with the binary arithmetic operator (`+`).

## Casting Values vs. Variables

Revisiting our third numeric promotional rule, the compiler doesn't require casting when working with literal values that fit into the data type. Consider these examples:

```
byte hat = 1;
byte gloves = 7 * 10;
short scarf = 5;
short boots = 2 + 1;
```

All of these statements compile without issue. On the other hand, neither of these statements compiles:

```
short boots = 2 + hat;    // DOES NOT COMPILE
byte gloves = 7 * 100;    // DOES NOT COMPILE
```

The first statement does not compile because `hat` is a variable, not a value, and both operands are automatically promoted to `int`. When working with values, the compiler had enough information to determine the writer's intent. When working with variables,



though, there is ambiguity about how to proceed, so the compiler reports an error. The second expression does not compile because 700 triggers an overflow for byte, which has a maximum value of 127.

## Compound Assignment Operators

Besides the simple assignment operator (=), Java supports numerous *compound assignment operators*. For the exam, you should be familiar with the compound operators in [Table 2.6](#).

**TABLE 2.6** Compound assignment operators

Operator	Example	Description
Addition assignment	a += 5	Adds the value on the right to the variable on the left and assigns the sum to the variable
Subtraction assignment	b -= 0.2	Subtracts the value on the right from the variable on the left and assigns the difference to the variable
Multiplication assignment	c *= 100	Multiplies the value on the right with the variable on the left and assigns the product to the variable
Division assignment	d /= 4	Divides the variable on the left by the value on the right and assigns the quotient to the variable

Compound operators are really just glorified forms of the simple assignment operator, with a built-in arithmetic or logical operation that applies the left and right sides of the statement and stores the resulting value in the variable on the left side of the statement. For example, the following two statements after the declaration of camel and giraffe are equivalent when run independently:

```
int camel = 2, giraffe = 3;
camel = camel * giraffe;    // Simple assignment operator
camel *= giraffe;          // Compound assignment operator
```

The left side of the compound operator can be applied only to a variable that is already defined and cannot be used to declare a new variable. In this example, if camel were not already defined, the expression camel \*= giraffe would not compile.

Compound operators are useful for more than just shorthand—they can also save you from having to explicitly cast a value. For example, consider the following. Can you figure out why the last line does not compile?

```
long goat = 10;
int sheep = 5;
sheep = sheep * goat;    // DOES NOT COMPILE
```

From the previous section, you should be able to spot the problem in the last line. We are trying to assign a long value to an int variable. This last line could be fixed with an explicit cast to (int), but there's a better way using the compound assignment operator:

```
long goat = 10;  
int sheep = 5;  
sheep *= goat;
```

The compound operator will first cast sheep to a long, apply the multiplication of two long values, and then cast the result to an int. Unlike the previous example, in which the compiler reported an error, the compiler will automatically cast the resulting value to the data type of the value on the left side of the compound operator.

## Return Value of Assignment Operators

One final thing to know about assignment operators is that the result of an assignment is an expression in and of itself equal to the value of the assignment. For example, the following snippet of code is perfectly valid, if a little odd-looking:

```
long wolf = 5;  
long coyote = (wolf = 3);  
System.out.println(wolf);    // 3  
System.out.println(coyote);  // 3
```

The key here is that (wolf=3) does two things. First, it sets the value of the variable wolf to be 3. Second, it returns a value of the assignment, which is also 3.

The exam creators are fond of inserting the assignment operator (=) in the middle of an expression and using the value of the assignment as part of a more complex expression. For example, don't be surprised if you see an if statement on the exam similar to the following:

```
boolean healthy = false;  
if(healthy = true)  
    System.out.print("Good!");
```

While this may look like a test if healthy is true, it's actually assigning healthy a value of true. The result of the assignment is the value of the assignment, which is true, resulting in this snippet printing Good!. We'll cover this in more detail in the upcoming "Equality Operators" section.

## Comparing Values

The last set of binary operators revolves around comparing values. They can be used to check if two values are the same, check if one numeric value is less than or greater than

another, and perform Boolean arithmetic. Chances are, you have used many of the operators in this section in your development experience.

## Equality Operators

Determining equality in Java can be a nontrivial endeavor as there's a semantic difference between "two objects are the same" and "two objects are equivalent." It is further complicated by the fact that for numeric and boolean primitives, there is no such distinction.

[Table 2.7](#) lists the equality operators. The equals operator (==) and not equals operator (!=) compare two operands and return a boolean value determining whether the expressions or values are equal or not equal, respectively.

**TABLE 2.7** Equality operators

Operator	Example	Apply to primitives	Apply to objects
Equality	<code>a == 10</code>	Returns true if the two values represent the same value	Returns true if the two values reference the same object
Inequality	<code>b != 3.14</code>	Returns true if the two values represent different values	Returns true if the two values do not reference the same object

The equality operator can be applied to numeric values, boolean values, and objects (including `String` and `null`). When applying the equality operator, you cannot mix these types. Each of the following results in a compiler error:

```
boolean monkey = true == 3;           // DOES NOT COMPILE
boolean ape = false != "Grape";       // DOES NOT COMPILE
boolean gorilla = 10.2 == "Koko";     // DOES NOT COMPILE
```

Pay close attention to the data types when you see an equality operator on the exam. As mentioned in the previous section, the exam creators also have a habit of mixing assignment operators and equality operators.

```
boolean bear = false;
boolean polar = (bear = true);
System.out.println(polar); // true
```

At first glance, you might think the output should be `false`, and if the expression were `(bear == true)`, then you would be correct. In this example, though, the expression is assigning the value of `true` to `bear`, and as you saw in the section on assignment operators, the assignment itself has the value of the assignment. Therefore, `polar` is also assigned a value of `true`, and the output is `true`.

For object comparison, the equality operator is applied to the references to the objects, not the objects they point to. Two references are equal if and only if they point to the same object or both point to null. Let's take a look at some examples:

```
var monday = new File("schedule.txt");
var tuesday = new File("schedule.txt");
var wednesday = tuesday;
System.out.println(monday == tuesday);    // false
System.out.println(tuesday == wednesday); // true
```

Even though all of the variables point to the same file information, only two references, tuesday and wednesday, are equal in terms of == since they point to the same object.

---

Wait, what's the File class? In this example, as well as during the exam, you may be presented with class names that are unfamiliar, such as File. Many times you can answer questions about these classes without knowing the specific details of these classes. In the previous example, you should be able to answer questions that indicate monday and tuesday are two separate and distinct objects because the new keyword is used, even if you are not familiar with the data types of these objects.

---

In some languages, comparing null with any other value is always false, although this is not the case in Java.

```
System.out.print(null == null);    // true
```

In [Chapter 4](#), we'll continue the discussion of object equality by introducing what it means for two different objects to be equivalent. We'll also cover String equality and show how this can be a nontrivial topic.

## Relational Operators

We now move on to *relational operators*, which compare two expressions and return a boolean value. [Table 2.8](#) describes the relational operators you need to know for the exam.

**TABLE 2.8** Relational operators

Operator	Example	Description
Less than	a < 5	Returns true if the value on the left is strictly <i>less than</i> the value on the right
Less than or equal to	b <= 6	Returns true if the value on the left is <i>less than or equal</i> to the value on the right

Operator	Example	Description
Greater than	<code>c &gt; 9</code>	Returns true if the value on the left is strictly <i>greater than</i> the value on the right
Greater than or equal to	<code>3 &gt;= d</code>	Returns true if the value on the left is <i>greater than or equal to</i> the value on the right
Type comparison	<code>e instanceof String</code>	Returns true if the reference on the left side is an instance of the type on the right side (class, interface, record, enum, annotation)

## Numeric Comparison Operators

The first four relational operators in [Table 2.8](#) apply only to numeric values. If the two numeric operands are not of the same data type, the smaller one is promoted, as previously discussed.

Let's look at examples of these operators in action:

```
int gibbonNumFeet = 2, wolfNumFeet = 4, ostrichNumFeet = 2;
System.out.println(gibbonNumFeet < wolfNumFeet);      // true
System.out.println(gibbonNumFeet <= wolfNumFeet);     // true
System.out.println(gibbonNumFeet >= ostrichNumFeet);  // true
System.out.println(gibbonNumFeet > ostrichNumFeet);   // false
```

Notice that the last example outputs false, because although `gibbonNumFeet` and `ostrichNumFeet` have the same value, `gibbonNumFeet` is not strictly greater than `ostrichNumFeet`.

## *instanceof* Operator

The final relational operator you need to know for the exam is the `instanceof` operator, shown in [Table 2.8](#). It is useful for determining whether an arbitrary object is a member of a particular class or interface at runtime.

Why wouldn't you know what class or interface an object is? As we will get into in [Chapter 6](#), "Class Design," Java supports polymorphism. For now, all you need to know is objects can be passed around using a variety of references. For example, all classes inherit from `java.lang.Object`. This means that any instance can be assigned to an object reference. For example, how many objects are created and used in the following code snippet?

```
Integer zooTime = Integer.valueOf(9);
Number num = zooTime;
Object obj = zooTime;
```

In this example, only one object is created in memory, but there are three different references to it because `Integer` inherits both `Number` and `Object`. This means you can call `instanceof` on any of these references with three different data types, and it will return `true` for each of them.

Where polymorphism often comes into play is when you create a method that takes a data type with many possible subclasses. For example, imagine that we have a function that opens the zoo and prints the time. As input, it takes a `Number` as an input parameter.

```
public void openZoo(Number time) {}
```

Now, we want the function to add `O'clock` to the end of output if the value is a whole number type, such as an `Integer`; otherwise, it just prints the value.

```
public void openZoo(Number time) {  
    if (time instanceof Integer)  
        System.out.print((Integer)time + " O'clock");  
    else  
        System.out.print(time);  
}
```

We now have a method that can intelligently handle both `Integer` and other values. A good exercise left for the reader is to add checks for other numeric data types such as `Short`, `Long`, `Double`, and so on.

Notice that we cast the `Integer` value in this example. It is common to use casting with `instanceof` when working with objects that can be various different types, since casting gives you access to fields available only in the more specific classes. It is considered a good coding practice to use the `instanceof` operator prior to casting from one object to a narrower type.

---

For the exam, you only need to focus on when `instanceof` is used with classes and interfaces. Although it can be used with other high-level types, such as records, enums, and annotations, it is not common.

---

### **Invalid *instanceof***

One area the exam might try to trip you up on is using `instanceof` with incompatible types. For example, `Number` cannot possibly hold a `String` value, so the following causes a compilation error:

```
public void openZoo(Number time) {  
    if(time instanceof String) // DOES NOT COMPILE
```

```
        System.out.print(time);  
    }
```

If the compiler can determine that a variable cannot possibly be cast to a specific class, it reports an error.

### ***null and the instanceof operator***

What happens if you call `instanceof` on a `null` variable? For the exam, you should know that calling `instanceof` on the `null` literal or a `null` reference always returns `false`.

```
System.out.print(null instanceof Object);           // false  
  
String noObjectHere = null;  
System.out.print(noObjectHere instanceof String);  // false
```

The preceding examples both print `false`. It almost doesn't matter what the right side of the expression is. We say "almost" because there are exceptions. This example does not compile, since `null` is used on the right side of the `instanceof` operator:

```
System.out.print(null instanceof null); // DOES NOT COMPILE
```

---

Although it may feel like you've learned everything there is about the `instanceof` operator, there's a lot more coming! In [Chapter 3](#), we introduce pattern matching with the `instanceof` operator, and in [Chapter 7](#), "Beyond Classes," we apply it to record patterns and also see how it impacts polymorphism.

---

## **Logical Operators**

If you have studied computer science, you may have already come across logical operators before. If not, no need to panic—we'll be covering them in detail in this section.


The logical operators, `(&)`, `(|)`, and `(^)`, may be applied to both numeric and boolean data types; they are listed in [Table 2.9](#). When they're applied to boolean data types, they're referred to as *logical operators*. Alternatively, when they're applied to numeric data types, they're referred to as *bitwise operators*, as they perform bitwise comparisons of the bits that compose the number.

**[TABLE 2.9](#)** Logical operators

Operator	Example	Description
Logical AND	<code>a &amp; b</code>	The value is true only if both values are true.

Operator	Example	Description
Logical inclusive OR	<code>c   d</code>	The value is true if at least one of the values is true.
Logical exclusive OR	<code>e ^ f</code>	The value is true only if one value is true and the other is false.

You should familiarize yourself with the truth tables in [Figure 2.2](#), where `x` and `y` are assumed to be boolean data types.

 An image illustrates the three logic truth tables using variables `a` and `b`. The rows depict the `x` function and the column depicts the `y` function. The operation includes if AND is only true if both operands are true, Inclusive OR is only false if both operands are false, and Exclusive OR is only true if the operands are different.

**FIGURE 2.2** The logical truth tables for `&`, `|`, and `^`

Here are some tips to help you remember this table:

- AND is only true if both operands are true.
- Inclusive OR is only false if both operands are false.
- Exclusive OR is only true if the operands are different.

Let's take a look at some examples:

```
boolean eyesClosed = true;
boolean breathingSlowly = true;

boolean resting = eyesClosed | breathingSlowly;
boolean asleep = eyesClosed & breathingSlowly;
boolean awake = eyesClosed ^ breathingSlowly;
System.out.println(resting); // true
System.out.println(asleep); // true
System.out.println(awake); // false
```

You should try these yourself, changing the values of `eyesClosed` and `breathingSlowly` and studying the results.

## Bitwise Operators

Bits are the 0 and 1 that you would see if you looked at a number in binary. For example, the number 2 is represented as 10 in binary. [Table 2.10](#) includes three bitwise operations that compare the 0s and 1s of a number, and return a new number based on these comparisons. For the exam, you don't need to do a lot of bitwise conversions, but you do need to know some basics.



**TABLE 2.10** Bitwise operators

Operator	Example	Description
Bitwise AND	a & b	Compares the bits of two numbers, returning a number that has a 1 in each digit in which <i>both</i> operands have a 1, and 0 otherwise.
Bitwise OR	c   d	Compares the bits of two numbers, returning a number that has a 1 in each digit in which <i>either</i> operand has a 1, and 0 otherwise.
Bitwise exclusive OR	e ^ f	Compares the bits of two numbers, returning a number that has a 0 in each digit that <i>matched</i> , and 1 otherwise.

First, we have the bitwise AND operator (&) and the bitwise OR operator (|). These operators return 1 when both or either corresponding binary digits are 1, respectively. First, you need to know that the original number is returned if both operands are the same.

```
int number = 70;
System.out.println(number);           // 70
System.out.println(number & number);  // 70
System.out.println(number | number);  // 70
```

You also need to know how bitwise operations work on a number and its negation, which returns 0 for bitwise AND (&), and -1 for bitwise OR (|).

```
int negated = ~number;
System.out.println(negated);           // -71

System.out.println(number & negated);  // 0
System.out.println(number | negated);  // -1
```

Finally, we have the binary exclusive OR operator (^). It works like the boolean version except with 0 as false, and 1 as true. However, it checks at each position in the number. You should know that it returns 0 if both numbers are the same (bits are all 0s), and -1 (bits are all 1s) for a value with its negation.

```
System.out.println(number ^ number);  // 0
System.out.println(number ^ negated);  // -1
```

## Conditional Operators

Next, we present the conditional operators, && and ||, in [Table 2.11](#).

**TABLE 2.11** Conditional operators

--

Operator	Example	Description
Conditional AND	a && b	The value is true only if both values are true. If the left side is false, then the right side will not be evaluated.
Conditional OR	c    d	The value is true if at least one of the values is true. If the left side is true, then the right side will not be evaluated.

The *conditional operators*, often called short-circuit operators, are nearly identical to the logical operators, & and |, except that the right side of the expression may never be evaluated if the final result can be determined by the left side of the expression. For example, consider the following snippet:

```
int hour = 10;
boolean zooOpen = true || (hour < 4);
System.out.println(zooOpen); // true
```

Referring to the truth tables, the value zooOpen can be false only if both sides of the expression are false. Since we know the left side is true, there's no need to evaluate the right side, since no value of hour will ever make this code print false. In other words, hour could have been -10 or 892; the output would have been the same. Try it yourself with different values for hour!

### Avoiding a *NullPointerException*

A more common example of where conditional operators are used is checking for null objects before performing an operation. In the following example, if duck is null, the program will throw a *NullPointerException* at runtime:

```
if(duck != null & duck.getAge() < 5) { // Could throw a NullPointerException
    // Do something
}
```

The issue is that the logical AND (&) operator evaluates both sides of the expression. We could add a second if statement, but this could get unwieldy if we have a lot of variables to check. An easy-to-read solution is to use the conditional AND operator (&&):

```
if(duck != null && duck.getAge() < 5) {
    // Do something
}
```

In this example, if duck is null, the conditional prevents a *NullPointerException* from ever being thrown, since the evaluation of duck.getAge() < 5 is never reached.

### Checking for Unperformed Side Effects

Be wary of short-circuit behavior on the exam, as questions are known to alter a variable on the right side of the expression that may never be reached. This is referred to as an *unperformed side effect*. For example, what is the output of the following code?

```
int rabbit = 6;
boolean bunny = (rabbit >= 6) || (++rabbit <= 7);
System.out.println(rabbit);
```

Because `rabbit >= 6` is true, the increment operator on the right side of the expression is never evaluated, so the output is 6.

## Making Decisions with the Ternary Operator

The final operator you should be familiar with for the exam is the conditional operator, (`? :` ), otherwise known as the *ternary operator*. It is notable in that it is the only operator that takes three operands. The ternary operator has the following form:

```
booleanExpression ? expression1 : expression2
```

The first operand must be a boolean expression, and the second and third operands can be any expression that returns a value. The ternary operation is really a condensed form of a combined `if` and `else` statement that returns a value. We cover `if/else` statements in a lot more detail in [Chapter 3](#), so for now we just use simple examples.

For example, consider the following code snippet that calculates the food amount for an owl:

```
int owl = 5;
int food;
if(owl < 2) {
    food = 3;
} else {
    food = 4;
}
System.out.println(food); // 4
```

Compare the previous code snippet with the following ternary operator code snippet:

```
int owl = 5;
int food = owl < 2 ? 3 : 4;
System.out.println(food); // 4
```

These two code snippets are equivalent. Note that it is often helpful for readability to add parentheses around the expressions in ternary operations, although doing so is certainly not required. It is especially helpful when multiple ternary operators are used together, though. Consider the following two equivalent expressions:

```
int food1 = owl < 4 ? owl > 2 ? 3 : 4 : 5;  
int food2 = (owl < 4 ? ((owl > 2) ? 3 : 4) : 5);
```

While they are equivalent, we find the second statement far more readable. That said, it is possible the exam could use multiple ternary operators in a single line.

For the exam, you should know that there is no requirement that second and third expressions in ternary operations have the same data types, although it does come into play when combined with the assignment operator. Compare the two statements following the variable declaration:

```
int stripes = 7;  
  
System.out.print((stripes > 5) ? 21 : "Zebra");  
  
int animal = (stripes < 9) ? 3 : "Horse"; // DOES NOT COMPILE
```

Both expressions evaluate similar boolean values and return an `int` and a `String`, although only the first one will compile. `System.out.print()` does not care that the expressions are completely different types, because it can convert both to `Object` values and call `toString()` on them. On the other hand, the compiler does know that "Horse" is of the wrong data type and cannot be assigned to an `int`; therefore, it does not allow the code to be compiled.

---

## Ternary Expression and Unperformed Side Effects

As we saw with the conditional operators, a ternary expression can contain an unperformed side effect, as only one of the expressions on the right side will be evaluated at runtime. Let's illustrate this principle with the following example:

```
int sheep = 1;  
int zzz = 1;  
int sleep = zzz < 10 ? sheep++ : zzz++;  
System.out.print(sleep + "," + zzz); // 2,1
```

Notice that since the left-hand boolean expression was true, only `sheep` was incremented. Contrast the preceding example with the following modification:

```
int sheep = 1;  
int zzz = 1;  
int sleep = sheep >= 10 ? sheep++ : zzz++;  
System.out.print(sleep + "," + zzz); // 1,2
```

Now that the left-hand boolean expression evaluates to false, only `zzz` is incremented. In this manner, we see how the expressions in a ternary operator may not be applied if the particular expression is not used.

For the exam, be wary of any question that includes a ternary expression in which a variable is modified in one of the expressions on the right-hand side.

---

## Summary

This chapter covered a wide variety of Java operator topics for unary, binary, and ternary operators. Ideally, most of these operators were review for you. If not, you need to study them in detail. It is important that you understand how to use all of the required Java operators covered in this chapter and know how operator precedence and parentheses influence the way a particular expression is interpreted.

There will likely be numerous questions on the exam that appear to test one thing, such as NIO.2 or exception handling, when in fact the answer is related to the misuse of a particular operator that causes the application to fail to compile. When you see an operator involving numbers on the exam, always check that the appropriate data types are used and that they match each other where applicable.

Operators are used throughout the exam, in nearly every code sample, so the better you understand this chapter, the more prepared you will be for the exam.

## Exam Essentials

**Be able to write code that uses Java operators.** This chapter covered a wide variety of operator symbols. Go back and review them several times so that you are familiar with them throughout the rest of the book.

**Be able to recognize which operators are associated with which data types.** Some operators may be applied only to numeric primitives, some only to boolean values, and some only to objects. It is important that you notice when an operator and operand(s) are mismatched, as this issue is likely to come up in a couple of exam questions.

**Understand when casting is required or numeric promotion occurs.** Whenever you mix operands of two different data types, the compiler needs to decide how to handle the resulting data type. When you're converting from a smaller to a larger data type, numeric promotion is automatically applied. When you're converting from a larger to a smaller data type, casting is required.

**Understand Java operator precedence.** Most Java operators you'll work with are binary, but the number of expressions is often greater than two. Therefore, you must understand the order in which Java will evaluate each operator symbol.