2 Working with Java data types

Exam objectives covered in this chapter	What you need to know
[2.2] Differentiate between object reference variables and primitive variables.	The primitive data types in Java, including scenarios when a particular primitive data type should or can't be used. Similarities and differences between the primitive data types. Similarities and differences between primitive and object reference variables.
[2.1] Declare and initialize variables.	Declaration and initialization of primitives and object reference variables. Literal values for primitive and object reference variables.
[3.1] Use Java operators.	Use of assignment, arithmetic, relational, and logical operators with primitives and object reference variables. Valid operands for an operator. Output of an arithmetic expression. Determine the equality of two primitives.
[3.2] Use parentheses to override operator precedence.	How to override the default operator precedence by using parentheses.

Imagine that you've just purchased a new home! Unfortunately, it didn't come with all the kitchen stuff. You'll likely need to buy different-sized containers to store different types of food items, because one size can't fit all! Also, you might move around food items in your home—perhaps because of a change in the requirements over time (you wish to eat it or you wish to store it).

Your new kitchen is an analogy for how Java stores its data using different data types and manipulates the data using operators. The food items are like data types in Java, and the containers used to store the food are like variables in Java. The change in the requirements that triggers a change in the state of food items can be compared to the processing logic. The agents of change (fire, heat, or cooling) that change the state of the food items can be compared to Java operators. You need these agents of change so that you can process the raw food items to create delicacies.

In the OCA Java SE 7 Programmer I exam, you'll be asked questions on the various data types in Java, such as how to create and initialize them and what their similarities and differences are. The exam will also question you on using the Java operators. This chapter covers the following:

- Primitive data types in Java
- Literal values of primitive Java data types
- Object reference variables in Java
- Valid and invalid identifiers
- Usage of Java operators
- Modification of default operator precedence via parentheses

2.1 Primitive variables

[2.1] Declare and initialize variables

[2.2] Differentiate between object reference variables and primitive variables

In this section, you'll learn all the primitive data types in Java, their literal values, and the process of creating and initializing primitive variables. A variable defined as one of the primitive data types is a *primitive variable*.

Primitive data types, as the name suggests, are the simplest data types in a programming language. In the Java language, they're predefined. The names of the primitive types are quite descriptive of the values that they can store. Java defines the following eight primitive data types:

- `char`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `boolean`

Examine figure 2.1 and try to match the given value with the corresponding type.

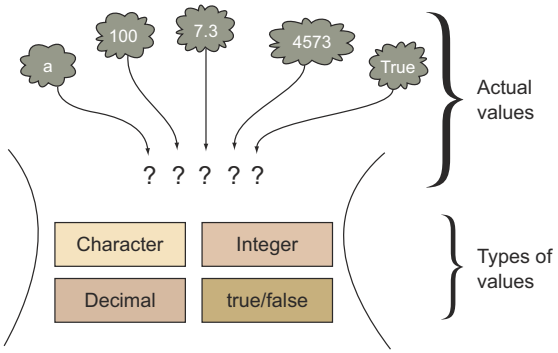


Figure 2.1 Matching a value with its corresponding type

This should be a simple exercise. Table 2.1 provides the answers.

Table 2.1 Matching a value with its corresponding data type

Character values	Integer values	Decimal values	true/false
a	100 4573	7.3	true

In the preceding exercise, I categorized the data that you need to store as follows: character, integer, decimal, and true/false values. This categorization will make your life simpler when confronted with selecting the most appropriate primitive data type to store a value. For example, to store an integer value, you need a primitive data type that is capable of storing integer values; to store decimal numbers, you need a primitive data type that can store decimal numbers. Simple, isn't it?

Let's map the types of data that the primitive data types can store, because it's always easy to group and remember information. The primitive data types can be categorized as follows: Boolean, character, and numeric (further categorized as integral and floating-point) types. Take a look at this categorization in figure 2.2.

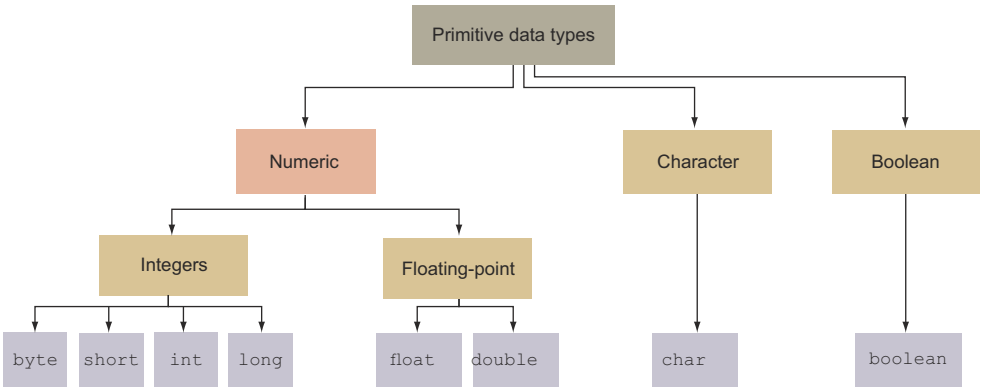


Figure 2.2 Categorization of primitive data types

This categorization in figure 2.2 will help you further associate each data type with the value that it can store. Let's start with the Boolean category.

2.1.1 Category: Boolean

The Boolean category has only one data type: `boolean`. A `boolean` variable can store one of two values: `true` or `false`. It is used in scenarios where only two states can exist. See table 2.2 for a list of questions and their probable answers.

Table 2.2 Suitable data that can be stored using a `boolean` data type

Question	Probable answers
Did you purchase the exam voucher?	Yes/No
Did you log in to your email account?	Yes/No
Did you tweet about your passion today?	Yes/No
Tax collected in financial year 2001–2002	Good question! But it can't be answered as yes/no.

EXAM TIP In this exam, the questions test your ability to select the best suitable data type for a condition that can only have two states: yes/no or true/false. The correct answer here is the `boolean` type.

Here's some code that defines `boolean` primitive variables:

```
boolean voucherPurchased = true;
boolean examPrepStarted = false;
boolean result = false;
boolean longDrive = true;
```

In some languages—such as JavaScript—you don't need to define the type of a variable before you use it. In JavaScript, the compiler defines the type of the variable according to the value that you assign to it. Java, in contrast, is a strongly typed language. You must declare a variable and define its type before you can assign a value to it. Figure 2.3 illustrates defining a `boolean` variable and assigning a value to it.

Another point to note here is the value that's assigned to a `boolean` variable. I used the literals `true` and `false` to initialize the `boolean` variables. A *literal* is a fixed value that doesn't need further calculations in order for it to be assigned to any variable. `true` and `false` are the only two `boolean` literals.

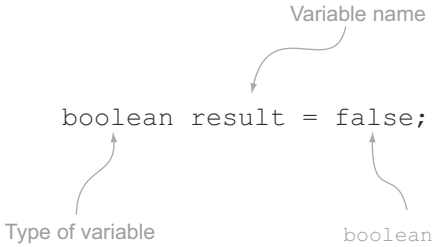


Figure 2.3 Defining and assigning a primitive variable

NOTE There are only two `boolean` literal values: `true` and `false`.

2.1.2 Category: Numeric

The numeric category defines two subcategories: integers and floating point (also called decimals). Let's start with the integers.

INTEGERS: BYTE, INT, SHORT, LONG

When you can count a value in whole numbers, the result is an integer. It includes both negative and positive numbers. Table 2.3 lists probable scenarios in which the data can be stored as integers.

Table 2.3 Data that can be categorized as numeric (nondecimal numbers) data type

Situation	Can be stored as integers?
Number of friends on Facebook	Yes
Number of tweets posted today	Yes
Number of photographs uploaded for printing	Yes
Your body temperature	Not always

You can use the `byte`, `short`, `int`, and `long` data types to store integer values. Wait a minute: why do you need so many types to store integers?

Each one of these can store a different range of values. The benefits of the smaller ones are obvious: they need less space in memory and are faster to work with. Table 2.4 lists all these data types, along with their sizes and the ranges of the values that they can store.

Table 2.4 Ranges of values stored by the numeric Java primitive data types

Data type	Size	Range of values
`byte`	8 bits	–128 to 127, inclusive
`short`	16 bits	–32,768 to 32,767, inclusive
`int`	32 bits	–2,147,483,648 to 2,147,483,647, inclusive
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, inclusive

The OCA Java SE 7 Programmer I exam may ask you questions about the range of integers that can be assigned to a `byte` data type, but it won't include questions on the ranges of integer values that can be stored by `short`, `int`, or `long` data types. Don't worry—you don't have to memorize the ranges for all these data types!

Here's some code that assigns literal values to primitive numeric variables within their acceptable ranges:

```
byte num = 100;
short sum = 1240;
int total = 48764;
long population = 214748368;
```

The default type of a nondecimal number is `int`. To designate an integer literal value as a `long` value, add the suffix `L` or `l` (`L` in lowercase), as follows:

```
long fishInSea = 764398609800L;
```

Integer literal values come in four flavors: binary, decimal, octal, and hexadecimal.

- *Binary number system*—A base-2 system, which uses only 2 digits, 0 and 1.
- *Octal number system*—A base-8 system, which uses digits 0 through 7 (a total of 8 digits). Here the decimal number 8 is represented as octal 10, decimal 9 as 11, and so on.
- *Decimal number system*—The base-10 number system that you use every day. It's based on 10 digits, from 0 through 9 (a total of 10 digits).
- *Hexadecimal number system*—A base-16 system, which uses digits 0 through 9 and the letters A through F (a total of 16 digits and letters). Here the number 10 is represented as A, 11 as B, 12 as C, 13 as D, 14 as E, and 15 as F.

Let's take quick look at how you can convert integers in the decimal number system to the other number systems. Figures 2.4, 2.5, and 2.6 show how to convert the decimal number 267 to the octal, hexadecimal, and binary number systems.

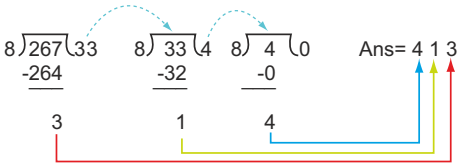


Figure 2.4 Converting an integer from decimal to octal

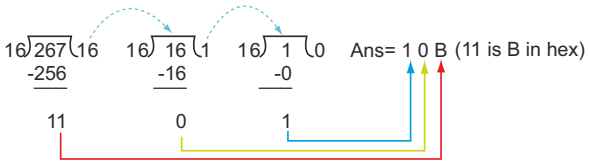


Figure 2.5 Converting an integer from decimal to hexadecimal

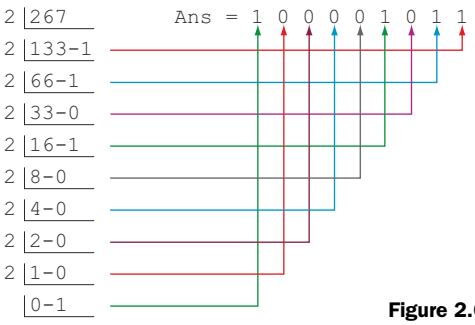


Figure 2.6 Converting an integer from decimal to binary

EXAM TIP In the OCA Java SE 7 Programmer I exam, you will not be asked to convert a number from the decimal number system to the octal and hexadecimal number systems and vice versa. But you can expect questions that ask you to select valid literals for integers. The previous figures will help you understand these number systems better and retain this information longer, which will in turn enable you to answer questions correctly during the exam.

You can assign integer literals in base decimal, binary, octal, and hexadecimal. For octal literals, use the prefix `0`; for binary, use the prefix `0B` or `0b`; and for hexadecimal, use the prefix `0x` or `0X`. Here's an example of each of these:

```
int baseDecimal = 267;
int octVal = 0413;
int hexVal = 0x10B;
int binVal = 0b100001011;
```

267 in decimal number system

267 in decimal number system is equal to 413 in octal number system

267 in decimal number system is equal to 10B in hexadecimal number system

276 in decimal number system is equal to 100001011 in binary number system

With Java version 7, you can also use underscores as part of the literal values, which helps group the individual digits or letters of the literal values and is much more readable. The underscores have no effect on the values. The following is valid code:

```
long baseDecimal = 100_267_760;
long octVal = 04_13;
long hexVal = 0x10_BA_75;
long binVal = 0b1_0000_10_11;
```

More readable literal values in binary, decimal, octal, and hexadecimal that use underscores to group digits and letters

RULES TO REMEMBER

Pay attention to the use of underscores in the numeric literal values. Here are some of the rules:

- You can't start or end a literal value with an underscore.
- You can't place an underscore right after the prefixes `0b`, `0B`, `0x`, and `0X`, which are used to define binary and hexadecimal literal values.
- You can place an underscore right after the prefix `0`, which is used to define an octal literal value.
- You can't place an underscore prior to an `L` suffix (the `L` suffix is used to mark a literal value as `long`).
- You can't use an underscore in positions where a string of digits is expected.

Because you are likely to be questioned on valid and invalid uses of underscores in literal values on the exam, let's look at some examples:

```
int intLiteral = _100;
int intLiteral2 = 100_999_;
long longLiteral = 100_L;
```

Can't start or end a literal value with an underscore

Can't place an underscore prior to suffix L

The following line of code will compile successfully but will fail at runtime:

```
int i = Integer.parseInt("45_98");
```

Invalid use of underscore where a string of digits is expected

Because a `String` value can accept underscores, the compiler will compile the previous code. But the runtime will throw an exception stating that an invalid format of value was passed to the method `parseInt`.

Here's the first Twist in the Tale exercise of this chapter for you to attempt. It uses multiple combinations of underscores in numeric literal values. See if you can get all of them right (answers in the appendix).

Twist in the Tale 2.1

Let's use the primitive variables `baseDecimal`, `octVal`, `hexVal`, and `binVal` defined earlier in this section and introduce additional code for printing the values of all these variables. Determine the output of the following code:

```
class TwistInTaleNumberSystems {
public static void main (String args[]) {
        int baseDecimal = 267;
        int octVal = 0413;
        int hexVal = 0x10B;
        int binVal = 0b100001011;
        System.out.println (baseDecimal + octVal);
        System.out.println (hexVal + binVal);
    }
}
```

Here's another quick exercise—let's define and initialize some `long` primitive variables that use underscores in the literal values assigned to them. Determine which of these does this job correctly:

```
long var1 = 0_100_267_760;
long var2 = 0_x_4_13;
long var3 = 0b_x10_BA_75;
long var4 = 0b_10000_10_11;
long var5 = 0xa10_AG_75;
long var6 = 0x1_0000_10;
long var7 = 100__12_12;
```

FLOATING-POINT NUMBERS: FLOAT AND DOUBLE

You need floating-point numbers where you expect decimal numbers. For example, can you define the probability of an event occurring as an integer? Table 2.5 lists probable scenarios in which the corresponding data is stored as a floating-point number.

In Java, you can use the `float` and `double` primitive data types to store decimal numbers. `float` requires less space than `double`, but it can store a smaller range of values than `double`. A `float` data type also has less precision, so even some of the numbers that are in the range of a `float` still can't be accurately represented when using `double`. Table 2.6 lists the sizes and ranges of values for `float` and `double`.

Table 2.5 Data that is stored as floating-point numbers

Situation	Is the answer a floating-point number?
Orbital mechanics of a spacecraft	Yes (very precise values are required)
Probability of your friend request being accepted	Yes; probability is between 0.0 (none) to 1.0 (sure)
Speed of Earth revolving around the Sun	Yes
Magnitude of an earthquake on the Richter scale	Yes

Table 2.6 Range of values for decimal numbers

Data type	Size	Range of values
`float`	32 bits	+/–1.4E–45 to +/–3.4028235E+38, +/–infinity, +/–0, `NaN`
`double`	64 bits	+/–4.9E–324 to +/–1.7976931348623157E+308, +/–infinity, +/–0, `NaN`

Here's some code in action:

```
float average = 20.129F;
float orbit = 1765.65f;
double inclination = 120.1762;
```

Did you notice the use of the suffixes `F` and `f` while initializing the variables `average` and `orbit` in the preceding code? The default type of a decimal literal is `double`, but by suffixing a decimal literal value with `F` or `f`, you tell the compiler that the literal value should be treated like a `float` and not a `double`.

You can also assign a literal decimal value in scientific notation as follows:

```
double inclination2 = 1.201762e2;
```

120.1762 is same as 1.201762e2 (the latter is expressed in scientific notation)

You can also add the suffix `D` or `d` to a decimal number value to specify that it is a `double` value. But because the default type of a decimal number is `double`, the use of the suffix `D` or `d` is redundant. Examine the following line of code:

```
double inclination = 120.1762D;
```

120.1762D is same as 120.1762

Starting with Java version 7, you can also use underscores with the floating-point literal values. The rules are generally the same as previously mentioned for numeric literal values; the following rules are specific to floating-point literals:

- You can't place an underscore prior to a `D`, `d`, `F`, or `f` suffix (these suffixes are used to mark a floating-point literal as `double` or `float`).
- You can't place an underscore adjacent to a decimal point.

Let's look at some examples that demonstrate the invalid use of underscores in floating-point literal values:

```
float floatLiteral = 100._48F;
double doubleLiteral = 100_.87;
```

Can't use underscore adjacent to a decimal point

```
float floatLiteral2 = 100.48_F;
double doubleLiteral2 = 100.87_d;
```

Can't use underscore prior to suffix F, f, D, or d

2.1.3 Category: Character

The character category defines only one data type: `char`. A `char` can store a single 16-bit Unicode character; that is, it can store characters from virtually all the existing scripts and languages, including Japanese, Korean, Chinese, Devanagari, French, German, Spanish, and others. Because your keyboard may not have keys to represent all these characters, you can use a value from `\u0000` (or `0`) to a maximum value of `\uffff` (or `65,535`) inclusive. The following code shows the assignment of a value to a `char` variable:

```
char c1 = 'D';
```

Use single quotes to assign a char, not double quotes

A very common mistake is using double quotes to assign a value to a `char`. The correct option is single quotes. Figure 2.7 shows a conversation between two (hypothetical) programmers, Paul and Harry.

What happens if you try to assign a `char` using double quotes? The code will fail to compile, with this message:

```
Type mismatch: cannot convert from String to char
```

EXAM TIP Never use double quotes to assign a letter to a `char` variable. Double quotes are used to assign a value to a variable of type `String`.

Internally, Java stores `char` data as an unsigned integer value (positive integer). It's therefore perfectly acceptable to assign a positive integer value to a `char`, as follows:

```
char c1 = 122;
```

Assign z to cl

The integer value `122` is equivalent to the letter `z`, but the integer value `122` is not equal to the Unicode value `\u0122`. The former is a number in base 10 (uses digits 0–9) and the latter is a number in base 16 (uses digits 0–9 and letters a–f). `\u` is used to mark

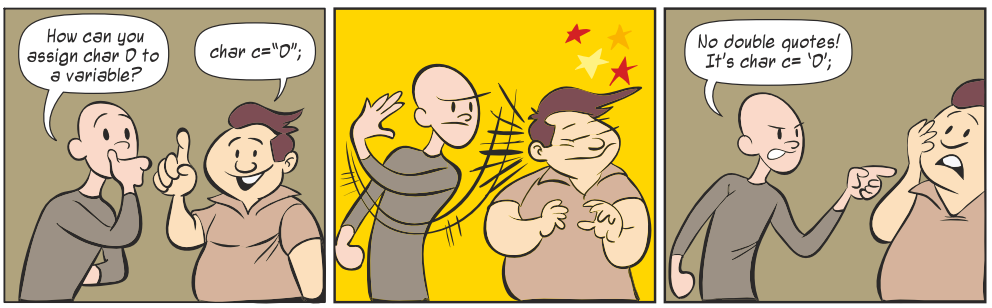


Figure 2.7 Never use double quotes to assign a letter as a `char` value.

the value as a Unicode value. You must use quotes to assign Unicode values to `char` variables. Here's an example:

```
char c2 = '\u0122';
System.out.println("c1 = " + c1);
System.out.println("c2 = " + c2);
```

Figure 2.8 shows the output of the preceding code on a system that supports Unicode characters.

```
c1 = z
c2 = Ģ
```

Figure 2.8 The output of assigning a character using the integer value `122` versus the Unicode value `\u0122`

As mentioned earlier, `char` values are unsigned integer values, so if you try to assign a negative number to one, the code will not compile. Here's an example:

```
char c3 = -122;          <-- Fails to compile
```

But you can forcefully assign a negative number to a char by casting it to `char`, as follows:

```
char c3 = (char)-122;                    <-- Compiles successfully
System.out.println("c3 = " + c3);
```

In the previous code, note how the literal value `-122` is prefixed by `(char)`. This practice is called *casting*. Casting is the forceful conversion of one data type to another data type.

You can cast only compatible data types. For example, you can cast a `char` to an `int` and vice versa. But you can't cast an `int` to a `boolean` value or vice versa. When you cast a bigger value to a data type that has a smaller range, you tell the compiler that you know what you're doing, so the compiler proceeds by chopping off any extra bits that may not fit into the smaller variable. Use casting with caution—it may not always give you the correct converted values.

Figure 2.9 shows the output of the preceding code that cast a value to `c3` (the value looks weird!).

```
c3 = ¬
```

Figure 2.9 The output of assigning a negative value to a character variable

The `char` data type in Java doesn't allocate space to store the sign of an integer. If you try to forcefully assign a negative integer to `char`, the sign bit is stored as the part of the integer value, which results in the storage of unexpected values.

EXAM TIP The exam will test your understanding of the possible values that can be assigned to a variable of type `char`, including whether an assignment will result in a compilation error. Don't worry—it won't test you on the value that is actually displayed after assigning arbitrary integer values to a `char`!

2.1.4 Confusion with the names of the primitive data types

If you've previously worked in another programming language, there's a good chance that you might get confused with the names of the primitive data types in Java and other languages. For example, C defines a primitive `short int` data type. But

short and int are two separate primitive data types in Java. The OCA Java SE 7 Programmer I exam will test you on your ability to recognize the names of the primitive data types, and the answers to these questions may not be immediately obvious. An example follows:

Question: What is the output of the following code?

```
public class MyChar {
    public static void main(String[] args) {
        int myInt = 7;
        bool result = true;
        if (result == true)
            do
                System.out.println(myInt);
            while (myInt > 10);

    }
}
```

a It prints 7 once.
b It prints nothing.
c Compilation error.
d Runtime error.

The correct answer is (c). This question tries to trick you with complex code that doesn't use any if constructs or do-while loops! As you can see, it uses an incorrect data type name, bool, to declare and initialize the variable result. Therefore, the code will fail to compile.

EXAM TIP Watch out for questions that use incorrect names for the primitive data types. For example, there isn't any bool primitive data type in Java. The correct data type is boolean. If you've worked with other programming languages, you might get confused trying to remember the exact names of all the primitive data types used in Java. Remember that just two of the primitive data types—int and char—are shortened; the rest of the primitive data types (byte, short, long, float, and double) are not.

2.2 Identifiers

Identifiers are names of packages, classes, interfaces, methods, and variables. Though identifying a valid identifier is not explicitly included in the OCA Java SE 7 Programmer I exam objectives, there's a good chance that you'll encounter a question similar to the following that will require you to identify valid and invalid identifiers:

Question: Which of the following lines of code will compile successfully?

a `byte exam_total = 7;`
b `int exam-Total = 1090;`

The correct answer is (a). Option (b) is incorrect because hyphens aren't allowed in the name of a Java identifier. Underscores are allowed. If you've worked with another programming language, this may be different from what you're used to.

2.2.1 Valid and invalid identifiers

Table 2.7 contains a list of rules that will enable you to correctly define valid (and invalid) identifiers, along with some examples.

Table 2.7 Ingredients of valid and invalid identifiers

Properties of valid Identifiers	Properties of invalid identifiers
Unlimited length	Same spelling as a Java reserved word or keyword (see table 2.8)
Starts with a letter (a–z, upper- or lowercase), a currency sign, or an underscore	Uses special characters: `!`, `@`, `#`, `%`, `^`, `&`, `*`, `(`, `)`, `'`, `:`, `;`, `[`, `/`, `\`, `}`
Can use a digit (not at the starting position)	Starts with a Java digit (`0–9`)
Can use an underscore (in any position)	
Can use a currency sign (in any position): `$`, `£`, `¢`, `¥` and others	
Examples of valid identifiers	**Examples of invalid identifiers**
`customerValueObject`	`7world` (identifier can't start with a digit)
`$rate, £Value, _sine`	`%value` (identifier can't use special char `%`)
`happy2Help, nullValue`	`Digital!, books@manning` (identifier can't use special char `!` or `@`)
`Constant`	`null`, `true`, `false`, `goto` (identifier can't have the same name as a Java keyword or reserved word)

You can't define a variable with the same name as Java keywords or reserved words. As these names suggest, they're reserved for specific purposes. Table 2.8 lists Java keywords, reserved words, and literals that you can't use as an identifier name.

Table 2.8 Java keywords and reserved words that can't be used as names for Java variables

`abstract`	`default`	`goto`	`package`	`this`
`assert`	`do`	`if`	`private`	`throw`
`boolean`	`double`	`implements`	`protected`	`throws`
`break`	`else`	`import`	`public`	`transient`
`byte`	`enum`	`instanceof`	`return`	`true`
`case`	`extends`	`int`	`short`	`try`
`catch`	`false`	`interface`	`static`	`void`
`char`	`final`	`long`	`strictfp`	`volatile`
`class`	`finally`	`native`	`super`	`while`
`const`	`float`	`new`	`switch`	
`continue`	`for`	`null`	`synchronized`	

Let's combat some of the common mistakes when determining correct and incorrect variables using the following variable declarations:

```
int falsetrue;
int javaseminar, javaSeminar;
int DATA-COUNT;
int DATA_COUNT;
int car.count;
int %ctr;
int ¥to£And$¢;
```

Valid: combination of two or more keywords

Valid (but using both these together can be very confusing)

Invalid: hyphen isn't allowed

Valid: underscore is allowed

Invalid: a dot in a variable name is not allowed

Invalid: % sign isn't allowed

Valid (though strange)

Next, let's look at the object reference variables and how they differ from the primitive variables.

2.3 Object reference variables

[2.1] Declare and initialize variables

[2.2] Differentiate between object reference variables and primitive variables

The variables in Java can be categorized into two types: *primitive variables* and *reference variables*. In this section, along with a quick introduction to reference variables, we'll cover the basic differences between reference variables and primitive variables.

Reference variables are also known as *object reference variables* or *object references*. I use all of these terms interchangeably in this text.

2.3.1 What are object reference variables?

Objects are instances of classes, including both predefined and user-defined classes. For a reference type in Java, the variable name evaluates to the address of the location in memory where the object referenced by the variable is stored. An object reference is, in fact, a memory address that points to a memory area where an object's data is located.

Let's quickly define a barebones class, `Person`, as follows:

```
class Person {}
```

When an object is instantiated with the `new` operator, a heap-memory address value to that object is returned. That address is usually assigned to the reference variable. Figure 2.10 shows a line of code that creates a reference variable `person` of type `Person` and assigns an object to it.

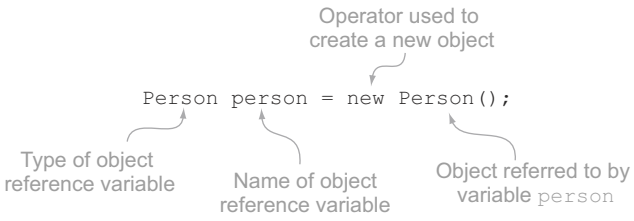


Figure 2.10 The creation and assignment of a reference variable

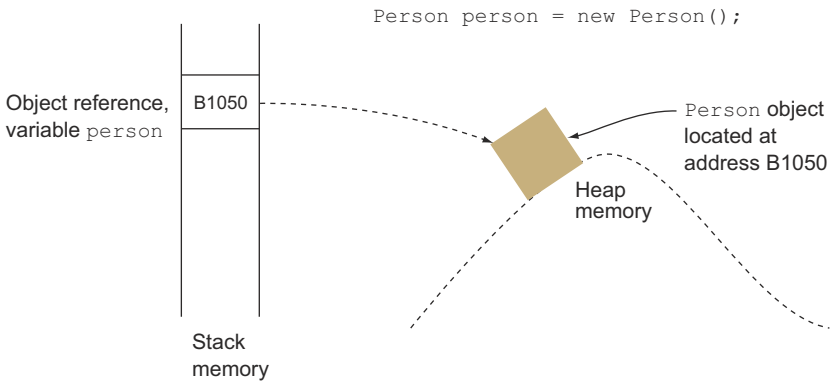


Figure 2.11 An object reference variable and the referenced object in memory

When the statement shown in figure 2.10 executes, three things happen:

- A new `Person` object is created.
- A variable named `person` is created in the stack with an empty (`null`) value.
- The variable `person` is assigned the memory address value where the object is located.

Figure 2.11 contains a pictorial representation of a reference variable and the object it refers to in memory.

You can think of an object reference variable as a *handle* to an object that allows you access to that object's attributes. The following analogy will help you understand object reference variables, the objects that they refer to, and their relationship. Think of objects as analogous to *dogs*, and think of object references as analogous to *leashes*. Although this analogy does not bear too much analysis, the following comparisons are valid:

- A leash not attached to a dog is a reference object variable with a `null` value.
- A dog without a leash is a Java object that's not referred to by any object reference variable.
- Just as an unleashed dog might be picked up by animal control, an object that is not referred to by a reference variable is liable to be garbage collected (removed from memory by the JVM).
- Several leashes may be tethered to a single dog. Similarly, several Java objects may be referenced by multiple object reference variables.

Figure 2.12 illustrates this analogy.

The literal value of all types of object reference variables is `null`. You can also assign a `null` value to a reference variable explicitly. Here's an example:

```
Person person = null;
```

In this case, the reference variable `person` can be compared to a leash without a dog.

NOTE The literal value for all types of object reference variables is `null`.

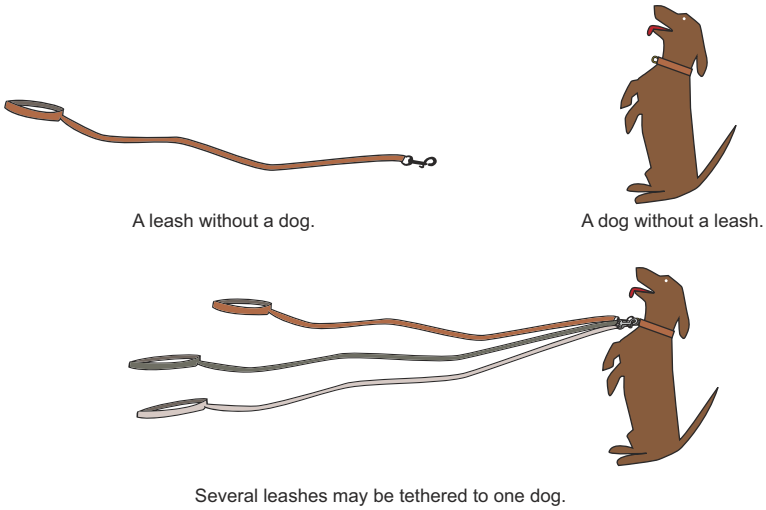


Figure 2.12 Dog leash analogy for understanding objects

2.3.2 Differentiating between object reference variables and primitive variables

Just as men and women are fundamentally different (according to John Gray, author of *Men Are from Mars, Women Are from Venus*), primitive variables and object reference variables differ from each other in multiple ways. The basic difference is that primitive variables store the actual values, whereas reference variables store the addresses of the objects they refer to.

Let's assume that a class `Person` is already defined. If you create an `int` variable `a`, and an object reference variable `person`, they will store their values in memory as shown in figure 2.13.

```
int a = 77;
Person person = new Person();
```

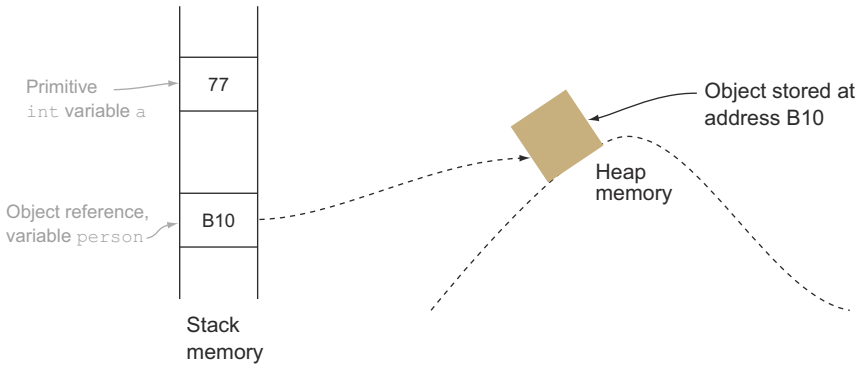


Figure 2.13 Primitive variables store the actual values, whereas object reference variables store the addresses of the objects they refer to.

Other important differences between primitive variables and object reference variables are shown in figure 2.14 as a conversation between a girl and a boy. The girl represents an object reference variable and the boy represents a primitive variable. (Don't worry if you don't understand all of these analogies. They'll make much more sense after you read related topics in later chapters.)

In the next section, you'll start manipulating these variables using operators.

2.4 Operators

[3.1] Use Java operators

[3.2] Use parentheses to override operator precedence

In this section, you'll use different types of operators—assignment, arithmetic, relational, and logical—to manipulate the values of variables. You'll also write code to

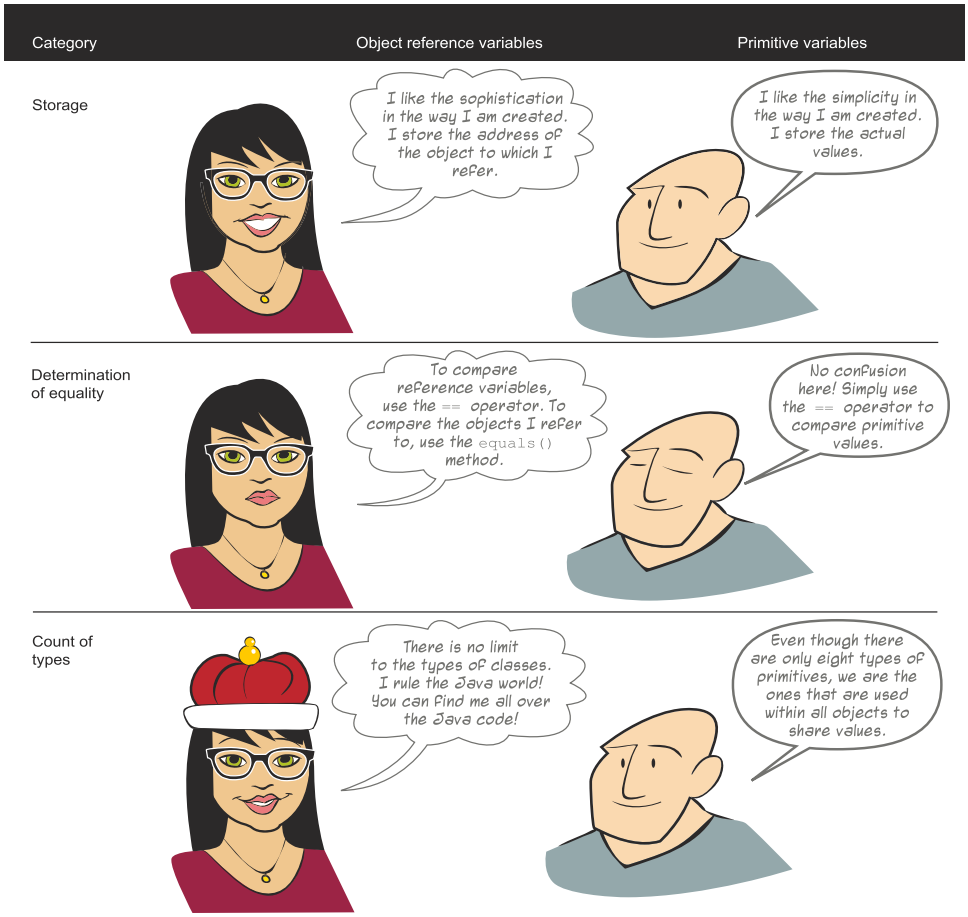


Figure 2.14 Differences between object reference variables and primitive variables

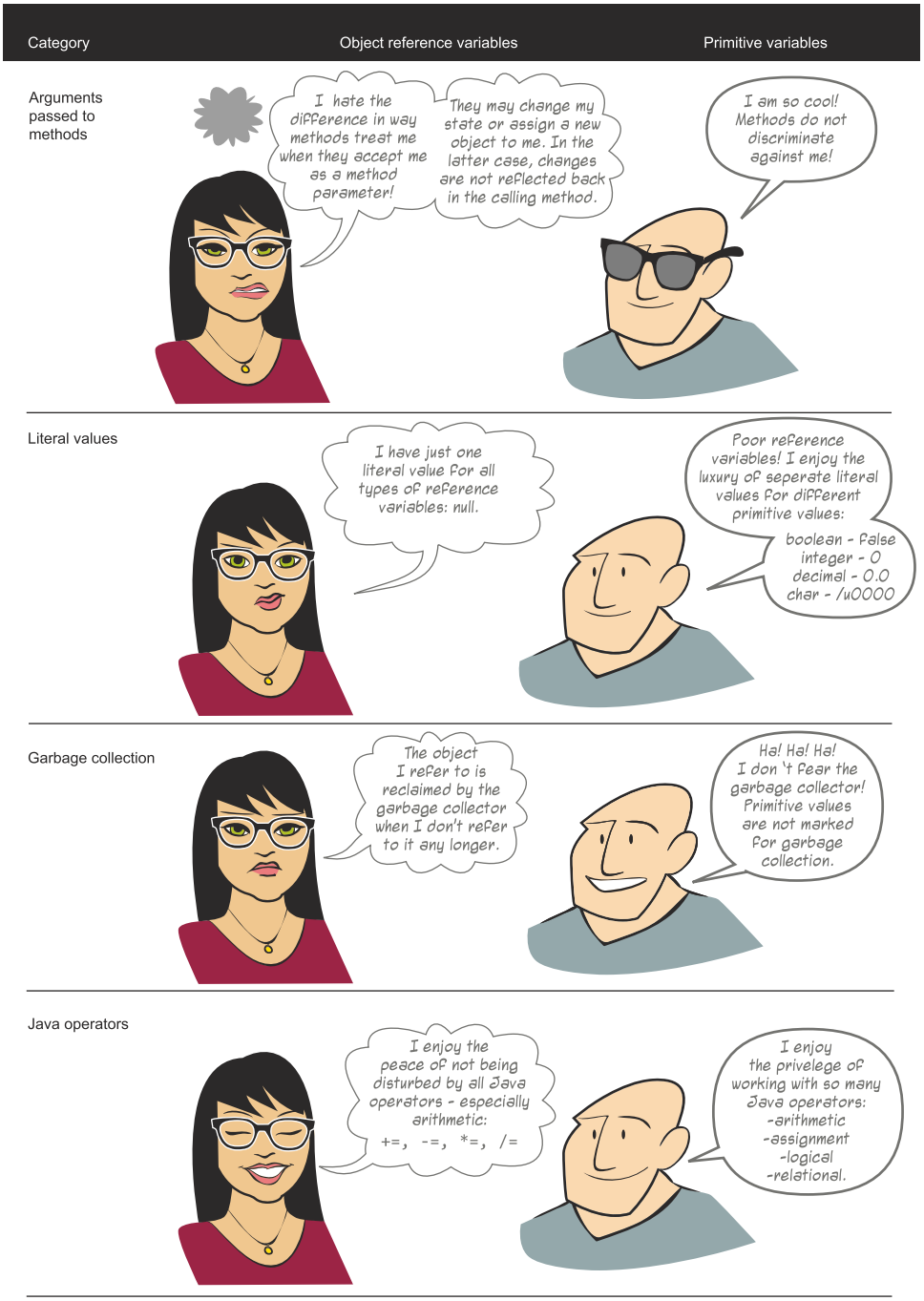


Figure 2.14 Differences between object reference variables and primitive variables *(continued)*

determine the equality of two primitive data types. You'll also learn how to modify the default precedence of an operator by using parentheses. For the OCA Java SE 7 Programmer I exam, you should be able to work with the operators listed in table 2.9.

Table 2.9 Operator types and the relevant operators

Operator type	Operators	Purpose
Assignment	`=, +=, -=, *=, /=`	Assign value to a variable
Arithmetic	`+, -, *, /, %, ++, --`	Add, subtract, multiply, divide, and modulus primitives
Relational	`<, <=, >, >=, ==, !=`	Compare primitives
Logical	`!, &&, ||`	Apply NOT, AND, and OR logic to primitives

NOTE Not all operators can be used with all types of operands. For example, you can determine whether a number is greater than another number, but you can't determine whether `true` is greater than `false` or a number is greater than `true`. Take note of this as you learn the usage of all the operators on the OCA Java SE 7 Programmer I exam.

2.4.1 Assignment operators

The assignment operators that you need to know for the exam are `=`, `+=`, `-=`, `*=`, and `/=`.

The simple assignment operator, `=`, is the most frequently used operator. It's used to initialize variables with values and to reassign new values to them.

The `+=`, `-=`, `*=`, and `/=` operators are short forms of addition, subtraction, multiplication and division with assignment. The `+=` operator can be read as "first add and then assign," and `-=` can be read as "first subtract and then assign." Similarly, `*=` can be read as "first multiply and then assign" and `/=` can be read as "first divide and then assign." If you apply these operators to two operands, `a` and `b`, they can be represented as follows:

```
a -= b is equal to a = a - b
a += b is equal to a = a + b
a *= b is equal to a = a * b
a /= b is equal to a = a / b
```

Let's have a look at some valid lines of code:

```
double myDouble2 = 10.2;

int a = 10;
int b = a;

float float1 = 10.2F;
float float2 = float1;
```

OK to assign variables of same type

OK to assign literal 10.2 to variable of type double

OK to assign literal 10 to variable of type int

OK to assign literal 10.2F to variable of type float

OK to assign variables of same type

```
b += a;

a = b = 10;
b -= a;

a = b = 10;
b *= a;

a = b = 10;
b /= a;
```

Reassign a value of 10 to both variables a and b

OK; b is assigned a value of 20. b = 10 + 10.

OK; b is assigned a value of 10. b = 20 – 10.

b is assigned a value of 100. b = 10 * 10.

b is assigned a value of 1. b = 10 / 10.

Let's have a look at some invalid lines of code:

```
double myDouble2 = true;
boolean b = 'c';
boolean b1 = 0;
boolean b2 -= b1;
```

Ouch! boolean can't be assigned to double

Ouch! char can't be assigned to boolean

Ouch! boolean can't be assigned a literal value other than true or false

Ouch! You can't add or subtract boolean values

Now let's try to squeeze the variables that can store a larger range of values into variables with a shorter range. Try the following assignments:

```
long num = 100976543356L;
int val = num;
```

Compiler won't allow this

It's similar to what is shown in figure 2.15, where someone is forcefully trying to squeeze a bigger value (`long`) into a smaller container (`int`).

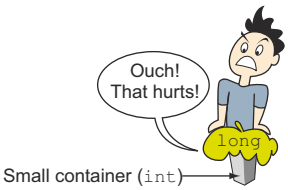


Figure 2.15 Assigning a bigger value (`long`) to a variable (`int`) that is only capable of storing smaller value range

You can still assign a bigger value to a variable that can only store smaller ranges by explicitly casting the bigger value to a smaller value. By doing so, you tell the compiler that you know what you're doing. In that case, the compiler proceeds by chopping off any extra bits that may not fit into the smaller variable. Beware! This approach may not always give you the correct converted values.

Compare the previous assignment example (assigning a `long` to an `int`) with the following example that assigns a smaller value (`int`) to a variable (`long`) that is capable of storing bigger value ranges:

```
int intVal = 1009;
long longVal = intVal;
```

Allowed

An `int` can easily fit into a `long` because there's enough room for it (as shown in figure 2.16).

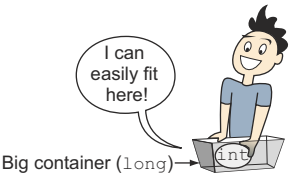


Figure 2.16 Assigning a smaller value (`int`) to a variable (`long`) that is capable of storing a larger value range

EXAM TIP You can't use the assignment operators to assign a `boolean` value to variables of type `char`, `byte`, `int`, `short`, `long`, `float`, or `double`, or vice versa.

You can also assign multiple values on the same line using the assignment operator. Examine the following lines of code:

Define and initialize variables on the same line

```
int a = 7, b = 10, c = 8;
a = b = c;
System.out.println(a);
```

Prints 8

❶ Assignment starts from right; the value of c is assigned to b and the value of b is assigned to a

On line ❶, the assignment starts from right to left. The value of variable `c` is assigned to the variable `b`, and the value of variable `b` (which is already equal to `c`) is assigned to the variable `a`. This is proved by the fact that line 3 prints `8`, and not `7`!

The next Twist in the Tale throws in a few twists with variable assignment and initialization. Let's see if you can identify the incorrect ones (answers in the appendix).

Twist in the Tale 2.2

Let's modify the assignment and initialization of the `boolean` variables used in previous sections. Examine the following code initializations and select the incorrect answers:

```
public class Foo {
    public static void main (String args[]) {
        boolean b1, b2, b3, b4, b5, b6;    // line 1
        b1 = b2 = b3 = true;               // line 2
        b4 = 0;                            // line 3
        b5 = 'false';                      // line 4
        b6 = yes;                          // line 5
    }
}
```

a. The code on line 1 will fail to compile.
b. Can't initialize multiple variables like the code on line 2.
c. The code on line 3 is correct.
d. Can't assign `'false'` to a `boolean` variable.
e. The code on line 5 is correct.

2.4.2 Arithmetic operators

Let's take a quick look at each of these operators, together with a simple example, in table 2.10.

Table 2.10 Use of arithmetic operators with examples

Operator	Purpose	Usage	Answer
+	Addition	`12 + 10`	`22`
-	Subtraction	`19 - 29`	`-10`
*	Multiplication	`101 * 45`	`4545`
/	Division (quotient)	`10 / 6` `10.0 / 6.0`	`1` `1.6666666666666667`

Table 2.10 Use of arithmetic operators with examples *(continued)*

Operator	Purpose	Usage	Answer
`%`	Modulus (remainder in division)	`10 % 6` `10.0 % 6.0`	`4` `4.0`
`++`	Unary increment operator; increments value by 1	`++10`	`11`
`--`	Unary decrement operator; decrements value by 1	`--10`	`9`

++ AND -- (UNARY INCREMENT AND DECREMENT OPERATORS)

The operators `++` and `--` are unary operators; they work with a single operand. They're used to increment or decrement the value of a variable by `1`.

Unary operators can also be used in prefix and postfix notation. In *prefix notation*, the operator appears before its operand:

```
int a = 10;
++a;
```

Operator ++ in prefix notation

In *postfix notation*, the operator appears after its operand:

```
int a = 10;
a++;
```

Operator ++ in postfix notation

When these operators are not part of an expression, the postfix and prefix notations behave in exactly the same manner:

```
int a = 20;
int b = 10;
++a;
b++;
System.out.println(a);
System.out.println(b);
```

Assign 20 to a
Assign 10 to b
Prints 21
Prints 11

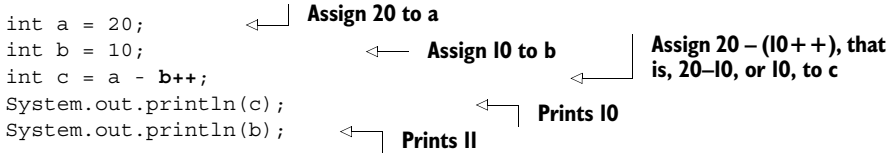
When a unary operator is used in an expression, its placement with respect to its operand decides whether its value will increment or decrement before the evaluation of the expression or after the evaluation of the expression. See the following code, where the operator `++` is used in prefix notation:

```
int a = 20;
int b = 10;
int c = a - ++b;
System.out.println(c);
System.out.println(b);
```

Assign 20 to a
Assign 10 to b
Assign 20 – (++10), that is, 20–11, or 9, to c
Prints 9
Prints 11

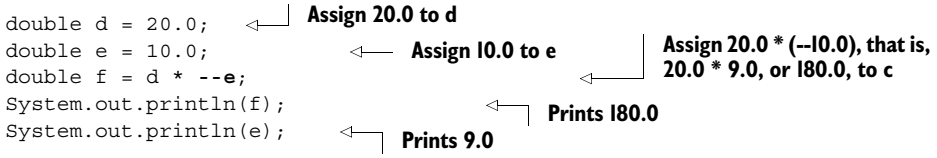
In the preceding example, the expression `a - ++b` uses the increment operator (`++`) in prefix notation. Therefore, the value of variable `b` increments to 11 before it is subtracted from 20, assigning the result `9` to variable `c`.

When ++ is used in postfix notation with an operand, its value increments after it has been used in the expression:

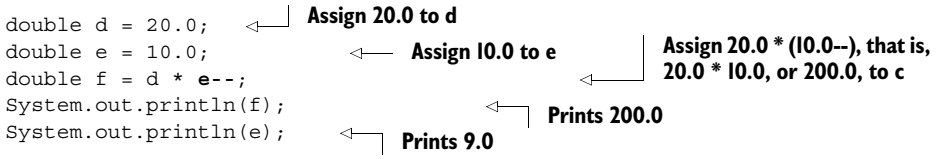


The interesting part here is that the value of `b` is printed as `11` in both cases because the value of the variable increments (or decrements) as soon as the expression in which it's used is evaluated.

The same logic applies to the unary operator, `--`. Here's an example:



Let's use the unary decrement operator (`--`) in postfix notation and see what happens:



Let's check out some example code that uses unary increment and decrement operators in both prefix and postfix notation in the same line of code. What do you think the output of the following code will be?

```
int a = 10;
a = a++ + a + a-- - a-- + ++a;
System.out.println(a);
```

The output of this code is `32`. The expression on the right-hand side evaluates from left to right, with the following values, which evaluate to 32:

```
a = 10 + 11 + 11 - 10 + 10;
```

The evaluation of an expression starts from left to right. For a prefix unary operator, the value of its operand increments or decrements just before its value is used in an expression. For a postfix unary operator, the value of its operand increments or decrements just after its value is used in an expression. Figure 2.17 illustrates what is happening in the preceding expression.

For the exam, it's important for you to have a good understanding of, and practice in, using postfix and prefix operators. In addition to the expressions shown in the previous examples, you can also find them in use as conditions in `if` statements, `for` loops, and `do-while` and `while` loops.

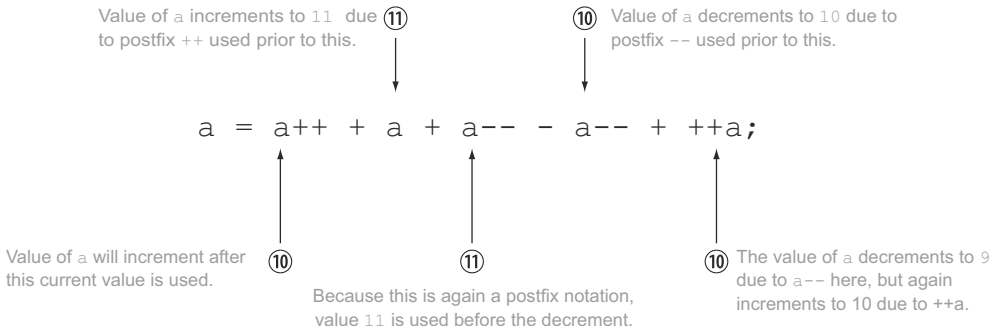


Figure 2.17 Evaluation of an expression that has multiple occurrences of unary operators in postfix and prefix notation

The next Twist in the Tale exercise will give you practice with unary operators used in prefix and postfix notation (answer in the appendix).

Twist in the Tale 2.3

Let's modify the expression used in figure 2.17 by replacing all occurrences of unary operators in prefix notation with postfix notations, and vice versa. So, `++a` changes to `a++`, and vice versa. Similarly, `--a` changes to `a--`, and vice versa. Your task is to evaluate the modified expression and determine the output of the following code:

```
int a = 10;
a = ++a + a + --a - --a + a++;
System.out.println (a);
```

Try to form the expression by replacing the values of variable `a` in the expression and explain each of them, the way it was done for you in figure 2.17.

2.4.3 Relational operators

Relational operators are used to check one condition. You can use these operators to determine whether a primitive value is equal to another value or whether it is less than or greater than the other value.

These relational operators can be divided into two categories:

- Comparing greater (`>`, `>=`) and lesser values (`<`, `<=`)
- Comparing values for equality (`==`) and nonequality (`!=`)

The operators `<`, `<=`, `>`, and `>=` work with all types of numbers, both integers (including `char`) and floating point, that can be added and subtracted. Examine the following code:

```
int i1 = 10;
int i2 = 20;
System.out.println(i1 >= i2);        <-- Prints false
```

```
long long1 = 10;
long long2 = 20;
System.out.println(long1 <= long2);      <-- Prints true
```

The second category of operators is covered in the following section.

EXAM TIP You cannot compare incomparable values. For example, you cannot compare a `boolean` with an `int`, a `char`, or a floating-point number. If you try to do so, your code will not compile.

COMPARING PRIMITIVES FOR EQUALITY (USING == AND !=)

The operators `==` (equal to) and `!=` (not equal to) can be used to compare all types of primitives: `char`, `byte`, `short`, `int`, `long`, `float`, `double`, and `boolean`. The operator `==` returns the `boolean` value `true` if the primitive values that you're comparing are equal, and `false` otherwise. The operator `!=` returns `true` if the primitive values that you're comparing are *not* equal, and `false` otherwise. For the same set of values, if `==` returns `true`, `!=` will return `false`. Sounds interesting!

Examine the following code:

```
int a = 10;
int b = 20;
System.out.println(a == b);          <-- Prints false
System.out.println(a != b);          <-- Prints true

boolean b1 = false;
System.out.println(b1 == true);      <-- Prints false
System.out.println(b1 != true);      <-- Prints true

System.out.println(b1 == false);     <-- Prints true
System.out.println(b1 != false);     <-- Prints false
```

Remember that you can't apply these operators to incomparable types. In the following code snippet, the code that compares an `int` variable to a `boolean` variable will fail to compile:

```
int a = 10;
boolean b1 = false;
System.out.println(a == b1);         <-- Causes compilation error
```

Here's the compilation error:

```
incomparable types: int and boolean
System.out.println(a == b1);
                      ^
```

EXAM TIP The result of the relational operation is always a `boolean` value. You can't assign the result of a relational operation to a variable of type `char`, `int`, `byte`, `short`, `long`, `float`, or `double`.

COMPARING PRIMITIVES USING THE ASSIGNMENT OPERATOR (=)

It's a very common mistake to use the assignment operator, `=`, in place of the equality operator, `==`, to compare primitive values. Before reading any further, check out the following code:

```
int a = 10;
int b = 20;
System.out.println(a = b);

boolean b1 = false;
System.out.println(b1 = true);
System.out.println(b1 = false);
```

❶ Prints 20 (this is not a boolean value!)

❷ Prints true

Prints false

❶ in the previous example isn't comparing the variables `a` and `b`. It's assigning the value of the variable `b` to `a` and then printing out the value of the variable `a`, which is `20`. Similarly, ❷ isn't comparing the variable `b1` with the `boolean` literal `true`. It's assigning the `boolean` literal `true` to variable `b1` and printing out the value of the variable `b1`.

NOTE You can't compare primitive values by using the assignment operator, `=`.

2.4.4 Logical operators

Logical operators are used to evaluate one or more expressions. These expressions should return a `boolean` value. You can use the logical operators `AND`, `OR`, and `NOT` to check multiple conditions and proceed accordingly. Here are a few real-life examples:

- *Case 1 (for managers)*—Request promotion if customer is extremely happy with the delivered project `AND` you think you deserve to be in your boss's seat!
- *Case 2 (for students)*—Accept job proposal if handsome pay and perks `OR` awesome work profile.
- *Case 3 (for entry-level Java programmers)*—If `NOT` happy with current job, change it.

In each of these example cases, you're making a decision (request promotion, accept job proposal, or change job) only if a set of conditions is satisfied. In case 1, a manager may request a promotion only if *both* the specified conditions are met. In case 2, a student may accept a new job if *either* of the conditions is true. In case 3, an entry-level Java programmer may change his or her current job if *not* happy with the current job; that is, if the specified condition (being happy with the current job) is false.

As illustrated in these examples, if you wish to proceed with a task when *both* the conditions are true, use the logical `AND` operator, `&&`. If you wish to proceed with a task when *either* of the conditions is true, use the logical `OR` operator, `||`. If you wish to wish to reverse the outcome of a `boolean` value, use the negation operator, `!`.

Time to look at some code in action:

```
int a = 10;
int b = 20;
System.out.println(a > 20 && b > 10);
System.out.println(a > 20 || b > 10);
System.out.println(! (b > 10));
System.out.println(! (a > 20));
```

❶ Prints false

❷ Prints true

❸ Prints false

❹ Prints true

❶ prints `false` because both the conditions, `a > 20` and `b > 10`, are not `true`. The first one (`a > 20`) is `false`. ❷ prints `true` because one of these conditions (`b > 10`) is `true`.

❸ prints `false` because the specified condition, `b > 10`, is `true`. ❹ prints `true` because the specified condition, `a > 20`, is `false`.

Table 2.11 will help you understand the result of using all these logical operators.

Table 2.11 Outcome of using `boolean` literal values with the logical operators `AND`, `OR`, and `NOT`

Operators `&&` (`AND`)	Operator `||` (`OR`)	Operator `!` (`NOT`)

`true && true → true`
`true && false → false`
`false && true → false`
`false && false → false`
`true && true && false → false`
	`true || true → true`
	`true || false → true`
	`false || true → true`
	`false || false → false`
	`false || false || true → true`
		`!true → false`
		`!false → true`

Here's a summary of this table:

- *Logical* `AND` *(`&&`)*—Evaluates to `true` if *all* operands are `true`; `false` otherwise.
- *Logical* `OR` *(`||`)*—Evaluates to `true` if *any* or all of the operands is `true`.
- *Logical negation (`!`)*—Negates the `boolean` value. Evaluates to `true` for `false`, and vice versa.

The operators `|` and `&` can also be used to manipulate individual bits of a number value, but I will not cover this usage here, because they are not on this exam.

&& AND || ARE SHORT-CIRCUIT OPERATORS

Another interesting point to note with respect to the logical operators `&&` and `||` is that they're also called *short-circuit* operators because of the way they evaluate their operands to determine the result. Let's start with the operator `&&`.

The `&&` operator returns `true` only if both the operands are `true`. If the first operand to this operator evaluates to `false`, the result can *never* be `true`. Therefore, `&&` does not evaluate the second operand. Similarly, the `||` operator does not evaluate the second operator if the first operand evaluates to `true`.

```
int marks = 8;
int total = 10;
System.out.println(total < marks && ++marks > 5);
System.out.println(marks);
System.out.println(total == 10 || ++marks > 10);
System.out.println(marks);
```

❶ Prints false
❷ Prints 8
❸ Prints true
❹ Prints 8

In the first print statement at ❶, because the first condition, `total < marks`, evaluates to `false`, the next condition, `++marks > 5`, is not even evaluated. As you can see ❷, the output value of `marks` is still `8` (the value to which it was initialized on line 1)! Similarly, in the next comparison ❸, because `total == 10` evaluates to `true`, the second condition, `++marks > 10`, isn't evaluated. Again, this can be verified when the value of `marks` is printed again ❹, and the output is `8`.

NOTE All the relational and logical operators return a `boolean` value, which can be assigned to a primitive `boolean` variable.

The purpose of the next Twist in Tale is to encourage you to play with code that uses short-circuit operators. To determine whether a `boolean` expression passed as an operand to the short-circuit operators evaluates, you can apply a unary increment operator (in postfix notation) to the variable used in the expression. Compare the new variable value with the old value to verify whether the expression was evaluated (answers in the appendix).

Twist in the Tale 2.4

As you know, the short-circuit operators `&&` and `||` may not evaluate both their operands if they can determine the result of the expression by evaluating just the first operand. Examine the following code and circle the expressions that you think will evaluate. Draw a square around the expressions that you think may not execute. (For example, on line 1, both `a++ > 10` and `++b < 30` will evaluate .)

```
class TwistInTaleLLogicalOperators {
    public static void main (String args[]) {
        int a = 10;
        int b = 20;
        int c = 40;

        System.out.println(a++ > 10 || ++b < 30);     // line1

        System.out.println(a > 90 && ++b < 30);

        System.out.println(!(c>20) && a==10 );

        System.out.println(a >= 99 || a <= 33 && b == 10);

        System.out.println(a >= 99 && a <= 33 || b == 10);

    }
}
```

Example use of the short-circuit operator `&&` in real projects

The logical operator `&&` is often used in code to check whether an object reference variable has been assigned a value before invoking a method on it:

```
String name = "hello";
if (name != null && name.length() > 0)
      System.out.println(name.toUpperCase());
```

2.4.5 Operator precedence

What happens if you use multiple operators within a single line of code with multiple operands? Which one should be treated like the king and given preference over the others?

Don't worry. Java already has a rule in place for just such a situation. Table 2.12 lists the precedence of operators: the operator on top has the highest precedence, and operators within the same group have the same precedence and are evaluated from left to right.

Table 2.12 Precedence of operators

Operator	Precedence
Postfix	`Expression++, expression--`
Unary	`++expression, --expression, +expression, -expression, !`
Multiplication	`*` (multiply), `/` (divide), `%` (remainder)
Addition	`+` (add), `-` (subtract)
Relational	`<, >, <=, >=`
Equality	`==, !=`
Logical AND	`&&`
Logical OR	`||`
Assignment	`=, +=, -=, *=, /=, %=`

Let's execute an expression that uses multiple operators (with different precedence) in an expression:

```
int int1 = 10, int2 = 20, int3 = 30;
System.out.println(int1 % int2 * int3 + int1 / int2);
```

❶ **Prints 300**

Because this expression ❶ defines multiple operators with different precedence, it's evaluated as follows:

```
(((int1 % int2) * int3)) + (int1 / int2)
(((10 % 20) * 30)) + (10 / 20)
( (10       * 30)) + (0)
( 300 )
```

What if you don't want to evaluate the expression in this way? The remedy is simple: use parentheses to override the default operator precedence. Here's an example that adds `int3` and `int1` before multiplying by `int2`:

```
int int1 = 10, int2 = 20, int3 = 30;
System.out.println(int1 % int2 * (int3 + int1) / int2);
```

Prints 20!

NOTE You can use parentheses to override the default operator precedence. If your expression defines multiple operators and you are unsure how your expression will be evaluated, use parentheses to evaluate in your preferred order. The inner parentheses are evaluated prior to the outer ones.

2.5 Summary

In this chapter, we started with the primitive data types in Java, including examples of where to use each of the kinds and their literal values. We also categorized the primitives into character type, integer type, and floating type. Then we covered the ingredients of valid and invalid Java identifiers.

We discussed the operators used to manipulate primitives (limited to the ones required for the OCA Java SE 7 Programmer I exam). We also covered the conditions in which a particular operator can be used. For example, if you wish to check whether a set of conditions is true, you can use the logical operators. It's also important to understand the operand types that can be used for each of these operators. For example, you can't use `boolean` operands with the operators `>`, `>=`, `=<`, and `<`.

2.6 Review notes

Primitive data types:

- Java defines eight primitive data types: `char`, `byte`, `short`, `int`, `long`, `float`, `double`, and `boolean`.
- Primitive data types are the simplest data types.
- Primitive data types are predefined by the programming language. A user can't define a primitive data type in Java.
- It's helpful to categorize the primitive data types as Boolean, numeric, and character data types.

The Boolean data type:

- The `boolean` data type is used to store data with only two possible values. These two possible values may be thought of as yes/no, 0/1, true/false, or any other combination. The actual values that a `boolean` can store are `true` and `false`.
- `true` and `false` are literal values.
- A literal is a fixed value that doesn't need further calculations to be assigned to any variable.

Numeric data types:

- Numeric values can be stored either as integers or decimal numbers.
- `byte`, `short`, `int`, and `long` can be used to store integers.
- The `byte`, `short`, `int`, and `long` data types use 8, 16, 32, and 64 bits, respectively, to store their values.
- `float` and `double` can be used to store decimal numbers.
- The float and double data types use 32 and 64 bits, respectively, to store their values.
- The default type of integers—that is, nondecimal numbers—is `int`.
- To designate an integer literal value as a `long` value, add the suffix `L` or `l` to the literal value.

- Numeric values can be stored in binary, octal, decimal, and hexadecimal number formats. The OCA Java SE 7 Programmer I exam will not ask you to convert a number from one number system to another.
- Literal values in the decimal number system use digits from 0 to 9 (a total of 10 digits).
- Literal values in the octal number system use digits from 0 to 7 (a total of 8 digits).
- Literal values in the hexadecimal number system use digits from 0 to 9 and letters from A to F (a total of 16 digits and letters).
- Literal values in the binary number system use digits 0 and 1 (a total of 2 digits).
- The literal values in the octal number system start with the prefix `0`. For example, `0413` in the octal number system is `267` in the decimal number system.
- The literal values in the hexadecimal number system start with the prefix `0x`. For example, `0x10B` in the hexadecimal number system is `267` in the decimal number system.
- The literal values in the binary number system start with the prefix `0b` or `0B`. For example, the decimal value `267` is `0B100001011` in the binary system.
- Starting with Java 7, you can use underscores within the Java literal values to make them more readable. `0B1_0000_10_11`, `0_413`, and `0x10_B` are valid binary, octal, and hexadecimal literal values.
- The default type of a decimal number is `double`.
- To designate a decimal literal value as a `float` value, add the suffix `F` or `f` to the literal value.
- The suffixes `D` and `d` can be used to mark a literal value as a `double` value. Though it's allowed, doing so is not required because the default value of decimal literals is `double`.

Character primitive data types:

- A `char` data type can store a single 16-bit Unicode character; that is, it can store characters from virtually all the world's existing scripts and languages.
- You can use values from `\u0000` (or 0) to a maximum of `\uffff` (or 65,535 inclusive) to store a `char`. Unicode values are defined in the hexadecimal number system.
- Internally, the `char` data type is stored as an unsigned integer value (only positive integers).
- When you assign a letter to a `char`, Java stores its integer equivalent value. You may assign a positive integer value to a `char` instead of a letter, such as 122.
- The literal value 122 is not the same as the Unicode value \u0122. The former is a decimal number and the latter is a hexadecimal number.
- Single quotes, not double quotes, are used to assign a letter to a `char` variable.

Valid identifiers:

- A valid identifier starts with a letter (a–z, upper- or lowercase), a currency sign, or an underscore. There is no limit to its length.
- A valid identifier can contain digits, but not in the starting place.
- A valid identifier can use the underscore and currency sign at any position of the identifier.
- A valid identifier can't have the same spelling as a Java keyword, such as `switch`.
- A valid identifier can't use any special characters, including `!`, `@`, `#`, `%`, `^`, `&`, `*`, `(`, `)`, `'`, `:`, `;`, `[`, `/`, `\`, or `}`

Operators:

- The OCA Java SE 7 Programmer I exam covers assignment, arithmetic, relational, and logical operators.

Assignment operators:

- Assignment operators can be used to assign or reassign values to all types of variables.
- A variable can't be assigned to an incompatible value. For example, character and numeric values cannot be assigned to a `boolean` variable, and vice versa.
- `+=` and `-=` are short forms of addition/subtraction and assignment.
- `+=` can be read as "first add and then assign" and `-=` can be read as "first subtract and then assign."

Arithmetic operators:

- Arithmetic operators can't be used with the `boolean` data type. Attempting to do so will make the code fail to compile.
- `++` and `--` are unary increment and decrement operators. These operators work with single operands.
- Unary operators can be used in prefix or postfix notation.
- When the unary operators `++` and `--` are used in prefix notation, the value of the variable increments/decrements just before the variable is used in an expression.
- When the unary operators `++` and `--` are used in postfix notation, the value of the variable increments/decrements just after the variable is used in an expression.
- By default, unary operators have a higher precedence than multiplication operators and addition operators.

Relational operators:

- Relational operators are used to compare values for equality (`==`) and non-equality (`!=`). They are also used to determine whether two numeric values are greater than (`>`, `>=`) or less than (`<`, `<=`) each other.
- You can't compare incomparable values. For example, you can't compare a `boolean` with an `int`, a `char`, or a floating-point number. If you try to do so, your code will not compile.

- The operators equal to (==) and not equal to (!=) can be used to compare all types of primitives: char, byte, short, int, long, float, double, and boolean.
- The operator == returns true if the primitive values being compared are equal.
- The operator != returns true if the primitive values being compared are *not* equal.
- The result of the relational operator is always a boolean value.

Logical operators:

- You can use the logical operators to determine whether a set of conditions is true or false and proceed accordingly.
- Logical AND (&&) evaluates to true if all operands are true, and false otherwise.
- Logical OR (||) evaluates to true if any or all of the operands is true.
- Logical negation (!) negates the boolean value. It evaluates to true for false, and vice versa.
- The result of a logical operation is always a boolean value.
- The logical operators && and || are also called short-circuit operators. If these operators can determine the output of the expression with the evaluation of the first operand, they don't evaluate the second operand.
- The && operator returns true only if both of the operands are true. If the first operand to this operator evaluates to false, the result can never be true. Therefore, && does not evaluate the second operand.
- Similarly, the || operator returns true if any of the operands is true. If the first operand to this operator evaluates to true, the result can never be false. Therefore, || does not evaluate the second operator.

2.7 Sample exam questions

Q2-1. Select all incorrect statements:

a. A programmer can't define a new primitive data type.
b. A programmer can define a new primitive data type.
c. Once assigned, the value of a primitive can't be modified.
d. A value can't be assigned to a primitive variable.

Q2-2. Which of the options are correct for the following code?

```
public class Prim {                                    // line 1
    public static void main(String[] args) {           // line 2
        char a = 'a';                                  // line 3
        char b = -10;                                  // line 4
        char c = '1';                                  // line 5
        integer d = 1000;                              // line 6
        System.out.println(++a + b++ * c - d);         // line 7
    }                                                  // line 8
}                                                      // line 9
```

a. Code at line 4 fails to compile.
b. Code at line 5 fails to compile.

c Code at line 6 fails to compile.

d Code at line 7 fails to compile.

Q2-3. What is the output of the following code?

```
public class Foo {
    public static void main(String[] args) {
        int a = 10;
        long b = 20;
        short c = 30;
        System.out.println(++a + b++ * c);
    }
}
```

a 611

b 641

c 930

d 960

Q2-4. Select the option(s) that is/are the best choice for the following: _____ should be used to store a count of cars manufactured by a car manufacturing company. _____ should be used to store whether this car manufacturing company modifies the interiors on the customer's request. _____ should be used to store the maximum speed of a car.

a `long`, `boolean`, `double`

b `long`, `int`, `float`

c `char`, `int`, `double`

d `long`, `boolean`, `float`

Q2-5. Which of the following options contain correct code to declare and initialize variables to store whole numbers?

a `bit a = 0;`

b `integer a2 = 7;`

c `long a3 = 0x10C;`

d `short a4 = 0512;`

e `double a5 = 10;`

f `byte a7 = -0;`

g `long a8 = 123456789;`

Q2-6. Select the options that, when inserted at `// INSERT CODE HERE`, will make the following code output a value of `11`:

```
public class IncrementNum {
    public static void main(String[] args) {
        int ctr = 50;
        // INSERT CODE HERE
        System.out.println(ctr % 20);
    }
}
```

a `ctr += 1;`

b `ctr =+ 1;`

c `++ctr;`

d `ctr = 1;`

Q2-7. What is the output of the following code?

```
int a = 10;
int b = 20;
int c = (a * (b + 2)) - 10-4 * ((2*2) - 6;
System.out.println(c);
```

a 218

b 232

c 246

d Compilation error

Q2-8. What is true about the following lines of code?

```
boolean b = false;
int i = 90;
System.out.println(i >= b);
```

a Code prints `true`

b Code prints `false`

c Code prints `90 >= false`

d Compilation error

Q2-9. Examine the following code and select the correct options:

```
public class Prim {                                              // line 1
    public static void main(String[] args) {                     // line 2
        int num1 = 12;                                           // line 3
        float num2 = 17.8f;                                      // line 4
        boolean eJavaResult = true;                              // line 5
        boolean returnVal = num1 >= 12 && num2 < 4.567           // line 6
                            || eJavaResult == true;
        System.out.println(returnVal);                           // line 7
    }                                                            // line 8
}                                                                // line 9
```

a Code prints `false`

b Code prints `true`

c Code will print `true` if code on line 6 is modified to the following:

```
boolean returnVal = (num1 >= 12 && num2 < 4.567) || eJavaResult ==
true;
```

d Code will print `true` if code on line 6 is modified to the following:

```
boolean returnVal = num1 >= 12 && (num2 < 4.567 || eJavaResult ==
false);
```

Q2-10. If the functionality of the operators = and > were to be swapped in Java (for the code on line numbers 4, 5, and 6), what would be the result of the following code?

```
boolean myBool = false;                              // line 1
int yourInt = 10;                                    // line 2
float hisFloat = 19.54f;                             // line 3

System.out.println(hisFloat > yourInt);              // line 4
System.out.println(yourInt = 10);                    // line 5
System.out.println(myBool > false);                  // line 6
```

a
```
true
true
false
```

b
```
10.0
false
false
```

c
```
false
false
false
```

d Compilation error

2.8 Answers to sample exam questions

Q2-1. Select all incorrect statements:

a A programmer can't define a new primitive data type.

b **A programmer can define a new primitive data type.**

c **Once assigned, the value of a primitive can't be modified.**

d **A value can't be assigned to a primitive variable.**

Answer: b, c, d

Explanation: Only option (a) is a correct statement. Java primitive data types are predefined by the programming language. They can't be defined by a programmer.

Q2-2. Which of the options are correct for the following code?

```
public class Prim {                                  // line 1
    public static void main(String[] args) {         // line 2
        char a = 'a';                                // line 3
        char b = -10;                                // line 4
        char c = '1';                                // line 5
        integer d = 1000;                            // line 6
        System.out.println(++a + b++ * c - d);       // line 7
    }                                                // line 8
}                                                    // line 9
```

a **Code at line 4 fails to compile.**

b Code at line 5 fails to compile.

c **Code at line 6 fails to compile.**

d **Code at line 7 fails to compile.**

Answer: a, c, d

Explanation:

Option (a) is correct. The code at line 4 fails to compile because you can't assign a negative value to a primitive `char` data type without casting.

Option (c) is correct. There is no primitive data type with the name "integer." The valid data types are `int` and `Integer` (a wrapper class with *I* in uppercase).

Option (d) is correct. The variable `d` remains undefined on line 7 because its declaration fails to compile on line 6. So the arithmetic expression `(++a + b++ * c - d)` that uses variable `d` fails to compile. There are no issues with using the variable `c` of the `char` data type in an arithmetic expression. The `char` data types are internally stored as unsigned integer values and can be used in arithmetic expressions.

Q2-3. What is the output of the following code?

```
public class Foo {
    public static void main(String[] args) {
        int a = 10;
        long b = 20;
        short c = 30;
        System.out.println(++a + b++ * c);
    }
}
```

a **611**
b 641
c 930
d 960

Answer: a

Explanation: The prefix increment operator (`++`) used with the variable `a` will increment its value before it is used in the expression `++a + b++ * c`. The postfix increment operator (`++`) used with the variable `b` will increment its value *after* its initial value is used in the expression `++a + b++ * c`.

Therefore, the expression `++a + b++ * c`, evaluates with the following values:

```
11 + 20 * 30
```

Because the multiplication operator has a higher precedence than the addition operator, the values 20 and 30 are multiplied before the result is added to the value 11. The example expression evaluates as follows:

```
(++a + b++ * c)
= 11 + 20 * 30
= 11 + 600
= 611
```

EXAM TIP Although questions 2-2 and 2-3 seemed to test you on your understanding of operators, they actually tested you on different topics. Question 2-2 tested you on the name of the primitive data types. Beware! The real exam has a lot of such questions. A question that may seem to test you on threads may actually be testing you on the use of a `do-while` loop!

Q2-4. Select the option(s) that is/are the best choice for the following: _____ should be used to store a count of cars manufactured by a car manufacturing company. _____ should be used to store whether this car manufacturing company modifies the interiors on the customer's request. _____ should be used to store the maximum speed of a car.

a **long, boolean, double**
b long, int, float
c char, int, double
d **long, boolean, float**

Answer: a, d

Explanation:

Options (a) and (d) are correct. Use a `long` data type to store big number values, a `boolean` data type to store yes/no values as `true/false`, and a `double` or `float` to store decimal numbers.

Option (b) is incorrect. You can't use an `int` to store yes/no or `true/false` values.

Option (c) is incorrect. You can't use a `char` data type to store very long values (such as the count of cars manufactured by the car manufacturer until a certain date). Also, it's conceptually incorrect to track counts using the `char` data type.

Q2-5. Which of the following options contain correct code to declare and initialize variables to store whole numbers?

a `bit a = 0;`
b `integer a2 = 7;`
c **`long a3 = 0x10C;`**
d **`short a4 = 0512;`**
e `double a5 = 10;`
f **`byte a7 = -0;`**
g **`long a8 = 123456789;`**

Answer: c, d, f, g

Explanation:

Options (a) and (b) are incorrect. There are no primitive data types in Java with the names `bit` and `integer`. The correct names are `byte` and `int`.

Option (c) is correct. It assigns a hexadecimal literal value to the variable `a3`.

Option (d) is correct. It assigns an octal literal value to the variable `a4`.

Option (e) is incorrect. It defines a variable of type `double`, which is used to store decimal numbers, not integers.

Option (f) is correct. `-0` is a valid literal value.

Option (g) is correct. `123456789` is a valid integer literal value that can be assigned to a variable of type `long`.

Q2-6. Select the options that, when inserted at `// INSERT CODE HERE`, will make the following code output a value of `11`:

```
public class IncrementNum {
    public static void main(String[] args) {
        int ctr = 50;
        // INSERT CODE HERE
        System.out.println(ctr % 20);
    }
}
```

a **`ctr += 1;`**

b `ctr =+ 1;`

c **`++ctr;`**

d `ctr = 1;`

Answer: a, c

Explanation: To output a value of `11`, the value of the variable `ctr` should be `51` because `51%20` is `11`. Operator `%` outputs the remainder from a division operation. The current value of the variable `ctr` is `50`. It can be incremented by `1` using the correct assignment or increment operator.

Option (b) is incorrect. Java does not define a `=+` operator. The correct operator is `+=`.

Option (d) is incorrect because it's assigning a value of `1` to the variable `result`, not incrementing it by `1`.

Q2-7. What is the output of the following code?

```
int a = 10;
int b = 20;
int c = (a * (b + 2)) - 10-4 * ((2*2) - 6;
System.out.println(c);
```

a `218`

b `232`

c `246`

d **Compilation error**

Answer: d

Explanation: First of all, whenever you answer any question that uses parentheses to override operator precedence, check whether the number of opening parentheses matches the number of closing parentheses. This code will not compile because the number of opening parentheses does not match the number of closing parentheses.

Second, you may not have to answer complex expressions in the real exam. Whenever you see overly complex code, look for other possible issues in the code. Complex code may be used to distract your attention from the real issue.

Q2-8. What is true about the following lines of code?

```
boolean b = false;
int i = 90;
System.out.println(i >= b);
```

a Code prints `true`
b Code prints `false`
c Code prints `90 >= false`
d **Compilation error**

Answer: d

Explanation: The code will fail to compile; hence, it can't execute. You can't compare incomparable types, such as a `boolean` value with a number.

Q2-9. Examine the following code and select the correct options:

```
public class Prim {                                              // line 1
    public static void main(String[] args) {                     // line 2
        int num1 = 12;                                           // line 3
        float num2 = 17.8f;                                      // line 4
        boolean eJavaResult = true;                              // line 5
        boolean returnVal = num1 >= 12 && num2 < 4.567           // line 6
                            || eJavaResult == true;
        System.out.println(returnVal);                           // line 7
    }                                                            // line 8
}                                                                // line 9
```

a Code prints `false`
b **Code prints `true`**
c **Code will print `true` if code on line 6 is modified to the following:**

```
boolean returnVal = (num1 >= 12 && num2 < 4.567) || eJavaResult ==
true;
```

d Code will print `true` if code on line 6 is modified to the following:

```
boolean returnVal = num1 >= 12 && (num2 < 4.567 || eJavaResult ==
false);
```

Answer: b, c

Explanation:

Option (a) is incorrect because the code prints `true`.

Option (d) is incorrect because the code prints `false`.

Both the short-circuit operators `&&` and `||` have the same operator precedence. In the absence of any parentheses, they are evaluated from left to right. The first expression, `num1 >= 12`, evaluates to `true`. The `&&` operator evaluates the second operand only if the first evaluates to `true`. Because `&&` returns `true` for its first operand, it evaluates the second operand, which is (`num2 < 4.567 || eJavaResult == true`). The second operand evaluates to `true`; hence the variable `returnVal` is assigned `true`.

Q2-10. If the functionality of the operators `=` and `>` were to be swapped in Java (for the code on line numbers 4, 5, and 6), what would be the result of the following code?

```
boolean myBool = false;                                          // line 1
int yourInt = 10;                                                // line 2
float hisFloat = 19.54f;                                         // line 3
```

```
System.out.println(hisFloat > yourInt);              // line 4
System.out.println(yourInt = 10);                    // line 5
System.out.println(myBool > false);                  // line 6
```

a
```
true
true
false
```

b
```
10.0
false
false
```

c
```
false
false
false
```

d Compilation error

Answer: b

Explanation: Because the question mentioned swapping the functionality of the operator `>` with `=`, the code on lines 4, 5, and 6 will actually evaluate to the following:

```
System.out.println(hisFloat = yourInt);
System.out.println(yourInt > 10);
System.out.println(myBool = false);
```

The result is shown in b.

Note that the expression `myBool = false` uses the assignment operator (`=`) and not a comparison operator (`==`). This expression assigns `boolean` literal `false` to `myBool`; it doesn't compare `false` with `myBool`. Watch out for similar (trick) assignments in the exam, which may *seem* to be comparing values.