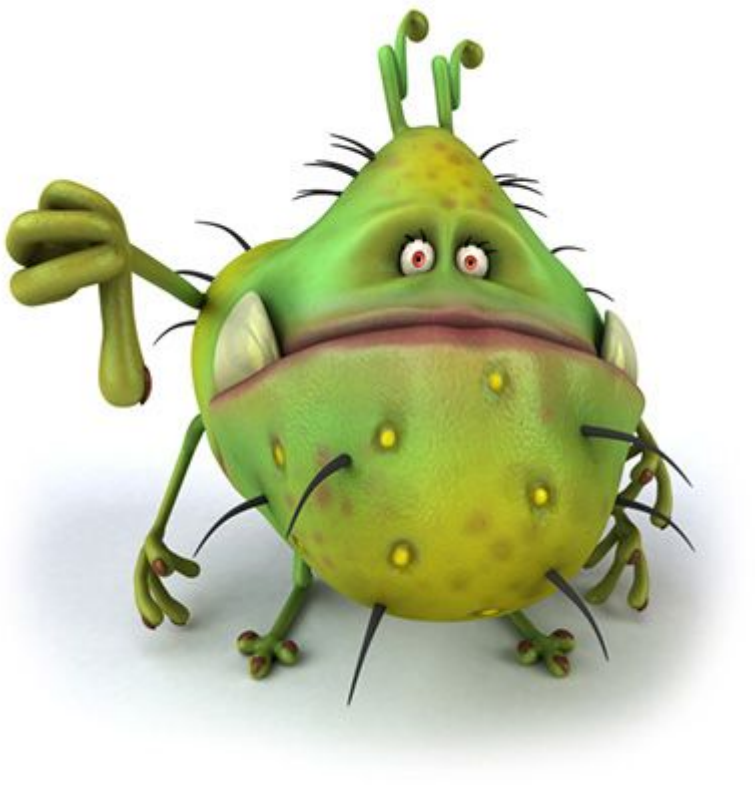
Lesson 3: Working With Java's Primitive Data Types

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

—Edsger Dijkstra, prominent Dutch computer scientist

Introduction

In this lesson, we'll dig a little further into how Java works. We'll start by looking at how it uses its *data types* (in other words, its different information formats). I included the quotation above because data types are confusing for new programmers, and many of the bugs in their programs are related to data types.



Avoiding bugs in Java

Programming languages like Java deal with many kinds of information: text, numbers, pictures, audio, video, and so on. Some of these are complex and change from time to time.

For example, if you want to store a picture on your computer, your format choices include bitmaps, JPEG, GIF, and others. One graphics program I use handles more than 20 formats. Audio and video storage also involve a number of formats.

No programming language can accommodate every type. Java's designers kept its basic data types simple so the language can stay as simple as possible. For the complex types like images, Java provides libraries of classes rather than trying to make those data types part of the language itself. That way, when a complex type changes, the library class is easier to change than the language itself would be.

But every type in Java, no matter how complex, comes from a few basic types that are in the language itself. And that's where we'll start in the next chapter.

Java's Primitive Data Types

What's a primitive data type? It's a simple information format included in the Java language. You might hear programmers call these "built-in data types" or "basic data types," but the formal term is primitive data types. Every other data type in Java has one or more primitive types as its basis.

Java has eight primitive types. They haven't changed much over time—there are six numeric primitives, one character primitive, and one logical primitive. They're the only types actually built into the language. (All other data types in Java are *reference data types*.)

Here's a list of the eight Java primitives, along with some information about each one.

Data Type	Size	Minimum Value	Maximum Value
byte	1 byte (8 bits)	-128	127

short	2 bytes (16 bits)	-32,768	32,767
int	4 bytes (32 bits)	-2,147,483,648	2,147,483,647
long	8 bytes (64 bits)	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	4 bytes (32 bits)	$\pm 1.4 \times 10^{-45}$	$\pm 3.4028235 \times 10^{38}$
double	8 bytes (64 bits)	$\pm 4.9 \times 10^{-324}$	$\pm 1.7976931348623157 \times 10^{308}$
char	2 bytes (16 bits)	Not applicable	Not applicable
boolean	1 byte (8 bits)	Not applicable	Not applicable

This table shows Java's eight primitive data types. For each type, it gives the size, and the minimum and the maximum values, if applicable.

Let's go through these in detail.

The Numeric Types

The first four types in the list (byte, short, int, and long) are for whole numbers. If you remember your high-school algebra, the "proper" term for whole numbers is *integers*. They can be positive, negative, or zero. The largest and smallest values for each within Java are in the table above.

The most commonly used of the four integer types is int. It's pretty much the standard whole number type unless there are reasons for using the others, like values that are too large for an int, in which case you can use long. Or if you're using only small values because you're worried about storage, perhaps byte or short makes more sense.

The next two types (float and double) store *floating-point numbers*, which are numbers that include decimal places. The minimum and maximum values for each of those are also in the table.

The primary concern with using these types is their accuracy. A float value is accurate to eight digits at most. So if I stored a number with more significant digits in it, like 12,345.09876, it would get stored as 12,345.099. A large number like 1,234,567,890 would get stored as 1.2345679×10^9.

Because of the limit on the number of digits, programmers don't use float often, unless accuracy isn't important or they're dealing with only a few significant digits. You'll see the double type much more frequently; it stores up to 16 significant digits.

> **What If These Numeric Types Aren't Big Enough?**
>
> Note
>
> If these types aren't large or accurate enough, Java has two classes named BigInteger and BigDecimal that can store very large numbers, like a googol. (By the way, that's the original spelling. Use Google to look up "googol" if you don't know how big it is. It's a word invented by a nine-year-old boy and used by his uncle, a mathematician. There's information about it in the Supplementary Material too.)
>
> The trade-off in using BigInteger or BigDecimal is speed. Programs that do lots of calculations will be slower if they use those two types.

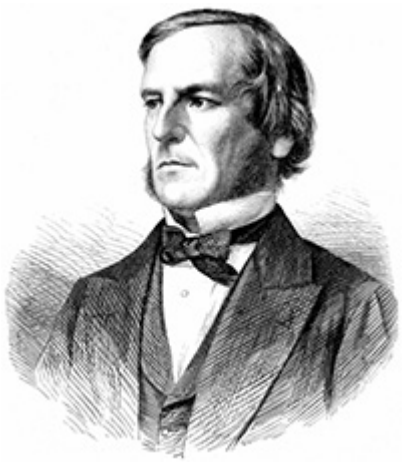
The Non-Numeric Types

Java's char type holds any single character, like *A* or *z* or *#* or *$*. This type stores characters using an encoding system called Unicode. Each character needs 2 bytes of storage space (that's 16 bits), so another name for this system is *Unicode-16* or *UTF-16* for short.

There are roughly 131,000 possible codes in Unicode, and programmers use about 110,000. Many alphabets are already encoded in the system, including Roman (which we use for English), Greek, Hebrew, Cyrillic, Arabic, Korean, and more. A number of special characters and symbols are also included, so we can display just about any character we'd want to see.

The last Java primitive is the boolean type. It's for the logical values true and false. Those are the only two values a boolean can hold.

Boolean values are named after George Boole, a 19th-century mathematician who developed an algebra system of two values. Programmers most often use this type in formal logic systems, where statements are either true or false. In Java, we'll use boolean values in comparisons and in making decisions.



George Boole, mathematician

I didn't include minimums and maximums for char and boolean types, since they use words rather than numbers.

You are welcome to share your questions with other students in the Discussion Area.

Now that you know what primitive data types are, let's start working with them. On to Chapter 3!

Using Primitive Data Types

You can use primitive types in Java programs in two ways: as a *literal* or with a *named value*.

A *literal* is the way we express a fixed value in Java. Programmers call it a literal because it looks exactly like what it is, whether that's a number, a letter, or a word. Examples are 3.14, 'a', and "pancakes".

A named value, by contrast, is a data field that's been given a name. Java has two types of named values: variables and constants.

A *variable* has a value that can change. Instead of having a fixed value, it's basically a placeholder. You may be familiar with variables from algebra; examples include x, y, and z.

A *named constant* is a name whose data value can't change. For example, Math.PI represents the mathematical value of pi, the ratio of a circle's diameter to its circumference, which is approximately 3.141592653589793.

Text equivalent start.

Chart showing that a primitive value is either a Literal or a Named Value. A Named Value can be Constant (doesn't change) or Variable (can change).

Text equivalent stop.

Literals

Java has several types of literals. Let's go through them quickly. Please feel free to share your questions with other students in the Discussion Area.

- **Integers:** Most whole number, or integer, literals are type int by default. That includes any whole number between the minimum and maximum values for an int. If you need to represent a larger long

literal, you can do that by adding the letter "L" to the number, like this: 100000000000L.

- **Floating-Point Numbers:** Any number with a decimal point is a floating-point literal. They are treated as double values unless you add an "F" to them to make them float values. So 1.2345 is a double literal, and 1.2345F is a float literal.

- **Characters:** A single character inside single quotation marks is a *character literal*. Examples include 'a', 'B', and '@'. If you put more than one character in single quotation marks, Java will give you a compile error.

- **Booleans:** Only two boolean literals exist—true and false. You can't use those two words for any other purpose, like names, in a Java program.

- **Binary Numbers:** You probably already know that *binary* is a numeral system with only two symbols: 0 and 1. Java allows a programmer to specify integers in binary if needed. Java treats any number beginning with 0b or 0B as a binary number and will allow only ones and zeros after 0b or 0B. Here's an example of a valid binary number: 0b0101010. That's the binary representation of 42. Binary literals aren't used often. You might use binary numbers if you needed to manipulate individual bits in a variable.

Binary code

- **Hexadecimal Numbers:** Most programmers don't learn about these right away because people don't use them very often. Hexadecimal (or just hex) numbers are base-16 numbers, using digits 0 through 9 and A through F. We won't use any in this course, but since they're valid, I just want you to be aware of them. Any number beginning with 0x or 0X is a hexadecimal numbers. Here's an example of a valid hexadecimal literal: 0x13BD.

- **Strings:** Character strings aren't a primitive type, but we use them so often that Java includes *string literals* to make our lives easier. Java considers any characters between double quotation marks to be a single character string. So "This is a character string." is a single string literal in Java. We already saw one string literal in our HelloWorld program, and we'll use them in just about every program we write.

The last thing I need to discuss with you about literals is that we can put special characters in string and character literals by putting a backslash (\) in front of them. For example, a *newline character* is \n, and a *tab character* is \t. To put a double quotation mark inside a string, we can use \" since without the backslash the quotation mark would end the string. Likewise, we can use \' to put a single quotation mark in a character literal. So the literal "This is a string\non two lines" would display in text output as

This is a string
on two lines

because of the "\n" newline character in the middle of the string.

Text equivalent start.

Instructions: Read the clue in the first column, and guess the term. Then read the second column for the answer.

Clue	Answer
A single character inside single quotation marks.	Characters
Whole number, or integer.	Integers
true and false.	Booleans
Number with a decimal point.	Floating-Point Numbers
0 and 1.	Binary Numbers
Base-16 numbers, using digits 0 through 9 and A through F.	Hexadecimal Numbers
Any characters between double quotation marks.	Strings

Text equivalent stop.

Now You Try

I've been telling you a lot so far this lesson. How about if I let you do something? Try to write a program that displays each type of literal on the screen.

Java's System.out.print() and System.out.println() methods can display any of the literal types above. If you'd like to look at a program that does that, I've included one at the link below, but please try it on your own before looking.

See my answer

How'd you do? If you have questions, or if you solved the problem in a different way, please feel free to share the code you wrote or your questions with other students in the Discussion Area. You'll also find a link to videos that take you through the steps of creating, compiling, and running the practice programs discussed in this lesson.

Let's move on from literals to variables.

Variables

Now that you know what Java's built-in types are, you need to learn how to use those types as variables. Again, a variable is just a name that we use to hold a value and whose value can change.

In Java, before we can use a variable, we have to *declare* it. That means we need to tell Java what name we want to use and what type that name should have. We do that with a statement that includes a type followed by one or more names we want to associate with that type. If you put more than one name in a declaration, separate them with commas.

Here are some examples of variable declarations:

```
int i;            // declaring an int variable
double x, y, z;   // declaring three doubles
boolean b;        // declaring a boolean
char c;           // declaring a char
String s;         // declaring a String
```

Once we've declared our variables, we can use them to store information—as long as the information is the right type. We store information in a variable using Java's *assignment statement* (a variable name) followed by the *assignment operator* (the value we want to assign to the variable). Java's assignment operator is the equal sign (=). To assign values to the variables I just declared, I could use these statements:

```
i = 99;           // an integer assignment
x = 0.0;          // one double assignment
y = z = 1.0;      // two double assignments
b = false;        // a boolean assignment
c = 'A';          // a character assignment
s = "a string";   // a string assignment
```

In each case, the value on the right gets stored in the variable on the left. The first statement sets the value of i to 99. The second sets x to 0.0. The only odd-looking statement is the third one, which assigns the same value to two variables at once. The value 1.0 gets stored in both y and z. That statement is a shorthand version of these two statements:

```
z = 1.0;
y = z;
```

In this example, the second statement assigns the value in z to y. So I can copy the value from one variable into another very easily that way.

Before Java can use a variable, we have to *initialize* it—in other words, give it a value to start off with, even though that value may change later. Java allows me to declare a variable and initialize it in one statement if I want to, like this:

```
int j = 0;                // int declaration and assignment
boolean p = true;         // boolean declaration and assignment
char d = 'x', e = '*';    // char declaration and assignment
```

I have just a couple more observations about declarations, and then I'll move on.

Remember that I can name a variable anything I want as long as it follows a few rules. Variable names . . .

- should use letters and numbers.
- should start with a letter.
- can't be the same as any of Java's keywords. To help you avoid this problem, I've included a link to Oracle's list of Java keywords in the Supplementary Material at the end of the lesson.
- should describe the information they will hold. My examples above wouldn't be good names in a Java program since they don't describe their contents. For example, in a program that uses x-coordinates and y-coordinates for a graph, I might name variables xCoordinate and yCoordinate.
- should start with a lowercase letter, and the first letter of each word in the name after that should be uppercase.

Text equivalent start.

Instructions: Read the clue in the first column, and guess the data type. Then read the second column for the answer.

Clue	Answer
'0', '1', 'g' 'n' 'r' 'y' 'f'	CHARACTER
-20.5555, 1025.6543, 9.555, -13.775	FLOAT-POINT-NUMBER
"a string", "String", "Some Text"	STRING
0XFFCC00, 0XFF0000	HEXADECIMAL-NUMBER
TRUE, FALSE	BOOLEAN
1010101, 00110011	BINARY-NUMBER
-300, 15, 18, 100	AN-INTEGER

Text equivalent stop.

After you declare a variable name, you can use it anywhere within the same set of curly brackets you declared it in. In programming, we call this the *scope* of a name. Java won't compile a program that uses a variable name outside those brackets because that would be outside the name's scope.

My last topic in this chapter is constants.

Constants

A constant in Java is a name whose value can't change. We don't use as many constants in programs as variables. The main difference in declaring constants in a program is that we use the final keyword to tell Java that once the constant's value is set, it can't change.

We also name constants a little differently. We use all uppercase letters and put underscores between words in the name.

Some examples of constant declarations are:

```
final int INT_CONSTANT = 0; // int constant
final double DOUBLE_CONSTANT = 0.0; // double constant
final String STRING_CONST = "Constant string";  // string constant
```

Now that you've got a firm foundation in primitive types, let's see how you can use those types in more sophisticated ways. Please join me in Chapter 4.

Java's Reference Data Types

As I discussed earlier, Java programmers use primitive types as the building blocks to the more complex *reference types*, which are also known as *objects*. Let me explain the difference between reference types and primitives.

Primitive types all share one property. A primitive variable's memory location contains the variable's value. That means for an int variable with a value of zero, its memory location will contain the number zero. A double variable with the value of pi will have 3.14159265358979 in its memory location.

A *reference variable*, however, doesn't contain its object's actual data in its memory location, since object sizes can vary. Instead, the variable contains the reference (also called a *memory address*) of the object it refers to. Let's look at an example to help explain that.

Java uses String objects to hold strings of characters. If I create a String object to hold the character string "a string", and I declare a variable named s to refer to it, the variable's memory location won't contain the character string. Instead, it will contain the memory address of the String object that contains the character string.

Here's a diagram that illustrates what I mean. In this diagram, each blue box represents an area in memory, with its name above it if it has one. The actual String object has no name of its own. Instead, there's a variable named s, and in its memory location is a reference to the String object. So even though we often talk about "the string named s," we really mean "the String object that s refers to."

Text equivalent start.

A box representing the variable s, with a line representing the reference to a String object containing a string.

Text equivalent stop.

When we're programming with primitive and reference variables, we don't treat them differently very often. For now, other than being aware of the distinction, please don't worry about it. I know that if you haven't worked with references or pointers before, the distinction probably isn't very clear. It confused me the first time I ran into it, so don't be concerned if it's fuzzy to you right now. I'll remind you about it when it makes a difference in a program.

One more thing: Both variables and constants can be any data type, either primitive or reference. The only difference between a variable and a constant is that the value of a constant can't change once you've set it, while a variable's value can change at any time.

Now that we know a bit more about Java's types, let's wrap up this lesson by playing around with them a bit. Try writing a small program that declares several variables of different types and then displays their values on the screen. My program is at the link below, but please try to write your own before looking at it.

See my answer

Wrapping Up With Wrapper Classes

Java's *wrapper classes* help us manage primitive types. They provide a class that in many ways can act like a primitive type, yet it's still a class. Some methods require classes as arguments, and wrappers let us use primitive values where objects are normally required because Java lets you substitute primitive types for wrapper objects. Java automatically "wraps" the primitive in a wrapper object when needed.

Each of the primitive types has a wrapper class. For most of them, it's the same name with the first letter capitalized. Float is the wrapper for float, Long is the wrapper for long, and so on. The two exceptions are int and char, whose wrapper classes are Integer and Character.

Each of the wrapper classes except Character has a method that converts, or *parses*, strings into that wrapper's primitive type. For example, if I have a String variable s containing the value "50", then Byte.parseByte(s) will give me back a byte value of 50; Integer.parseInt(s) will give me an int value of 50; Double.parseDouble(s) will give me a double value of 50.0; and so on. If s contains "true" or "false", then Boolean.parseBoolean(s) will give me back a boolean value of true or false.

What the Period Means

Note

The names of these methods and some class constants (like Math.PI, which I mentioned earlier) each have a period in them. The period tells Java that the part of the name after the period "belongs" to the part of the name before the period.

So PI is a constant within the Math class, parseInt() is a method inside the Integer class, and so on. Programmers sometimes call this period the *dot operator*.

Each of the wrapper classes also has a toString() method that converts a value from a primitive type into a String object. Calling Integer.toString(999) will, for example, return the string "999".

Each of the numeric wrapper classes also has two useful constants that let us know the minimum and maximum values for that type. The constants have the names MAX_VALUE and MIN_VALUE. So Byte.MIN_VALUE would tell you the smallest byte value, and Float.MAX_VALUE would tell you the largest float value.

Here are some examples to give you an idea of what we can do with these wrappers:

```
String intString = "99";
String floatString = "99.999";
int intVariable;
double doubleVariable;

// stores 99 into intVariable then displays it
intVariable = Integer.parseInt(intString);
System.out.println(intVariable);

// stores 99.999 into doubleVariable
doubleVariable = Double.parseDouble(floatString);
System.out.println(doubleVariable);

// displays largest int Value
System.out.println(Integer.MAX_VALUE);

// stores then displays largest double value
doubleVariable = Double.MAX_VALUE;
System.out.println(doubleVariable);
```

Try a few of your own combinations, and see what you can find out. Let me know if you have trouble with any of it.

Casting

One last topic before we wrap things up. Java is a *strongly typed* language, which means the type of a name is very important in how we use it. We can't mix most types, and for some that seem like we should be able to, we can't without telling Java that we really mean to. For example, I can't assign a double value to an integer variable without telling Java that's exactly what I want to do. Let me tell you the rules and a little bit about why they are that way, and then we'll get into specifics about how to do what we want to do.

It makes sense that Java won't let us assign boolean or string values to a numeric variable. Java doesn't have a numeric equivalent of true or false, and it has no numeric equivalent for a character string like "abcdefg." But why won't Java let me assign a double value to an int variable? (If you're curious, try it in a program, and see what happens when you try to compile it.) To answer that, we need to discuss the concepts of *wider* and *narrower* numeric types.

A numeric type is wider than another if it can hold a broader range of values. An int is wider than a short, for example. The types in order from narrowest to widest are byte, short, int, long, float, and double. Java will let you assign a narrower type to a wider variable, but it won't allow the reverse. Look at this code to see what I mean:

```
int i = 0;
double d = 0.0;
d = i;  // this will work
i = d;  // this won't work
```

The first three lines will compile just fine, but the fourth won't, even though the value in d is small enough to assign to i. The compiler looks at the types and gives this error: "possible loss of precision." That's because it's possible for a double to hold values that an int can't.

Java allows you to convert a value of one data type to another data type in a process called *casting*. There are two kinds.

- **Implicit casting:** Java will convert values that are safe for us. Programmers call this *implicit casting* because we don't specify the change to Java.

- **Explicit casting:** Java won't convert unsafe values unless we explicitly tell it to make the change. That's *explicit casting* because we tell Java to make the change even if it isn't safe. If the value in the wider type doesn't fit into the narrower type, Java just cuts off the part that doesn't fit. So be very careful with explicit casting.

If you need to, here's how to do explicit casting. Put the type you want to change to in parentheses in front of the type you're converting from. So while the code above won't compile, this code will:

```
int i = 0;
double d = 0.0;
d = i;  // this will work
i = (int)d;  // this will work too
```

Java will just throw away whatever part of the number in d that won't fit into i. That can lead to some unexpected results if it's done incorrectly. Explicit casting can be beneficial at times, such as when you're doing mathematical calculations. But for now, you should know about it so you'll understand any "loss of precision" error messages you get, and you'll know how to deal with them.

One conversion that Java can do, though, is surprising. It's there so we can have an easier time with output operations. Java knows how to convert any of its primitive types to a character string through an operation called *string concatenation*, which is the process of linking multiple strings into one. The concatenation operator is the plus sign whenever one side or the other of the operation is a string. Here are a few examples:

```
String s = "abc";
String t = "DEF";
int i = 99;
double d = 9.999;



String u = s + t;  // puts "abcDEF" into u
u = s + i;         // puts "abc99" into u
u = t + d;         // puts "DEF9.999" into u
```

What that means for output is that I can use one line where I used two in my previous examples. Instead of this:

```
System.out.print("The value in my int variable is ");
System.out.println(myInteger);
```

I can do this:

```
System.out.println("The value in my int variable is " + myInteger);
```

And I get exactly the same results. In general, shorter code is more elegant and allows fewer errors to creep in. So as you get more comfortable programming in Java, string concatenation can really work for you.

Let's move on to Chapter 5 for a summary of this lesson and a preview of Lesson 4.

Summary

Try this game to see how much you remember about the differences between data types.

Text equivalent start.

Instructions: Decide if the word or phrase represents Primitive data types or Reference data types. Then read the second column for the answer.

Word or phrase	Answer
Each of these contains the memory address of the object it refers to.	Reference data types

Word or phrase	Answer
You can use these as a literal or with a named value.	Primitive data types
These include float, double, and boolean.	Primitive data types
Another name for these is "objects."	Reference data types
They're actually built into the language of Java.	Primitive data types

Text equivalent stop.

If you had trouble with the game, don't worry. These data types will become more familiar as you progress in the course.

In this lesson, we covered some basic, foundational topics that we'll build on in future lessons. If you have any questions about it, you are welcome to share them with other students in the Discussion Area.

Next time we'll look at how to design and build new data types (classes) from Java's primitive types, and we'll rework our simple HelloWorld starter program into a more general class that will display different text strings.

I'll see you in Lesson 4. Until then, though, have a look at the Supplementary Material, take the quiz, and work through the assignment. Goodluck!