

JB0150: JAVA OOP: A GENTLE INTRODUCTION TO JAVA DATA TYPES*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNXX and licensed under the
Creative Commons Attribution License 3.0[†]

Abstract

This module introduces Java data types.

1 Table of Contents

- Preface (p. 1)
 - General (p. 1)
 - Prerequisites (p. 2)
 - Viewing tip (p. 2)
 - * Figures (p. 2)
- Discussion (p. 2)
 - Introduction (p. 2)
 - Primitive types (p. 4)
 - * Whole-number types (p. 5)
 - * Floating-point types (p. 6)
 - * The character type (p. 11)
 - * The boolean type (p. 11)
 - User-defined or reference types (p. 12)
 - Sample program (p. 14)
- Miscellaneous (p. 14)

2 Preface

2.1 General

This module is part of a sub-collection of modules designed to help you learn to program computers.
It introduces Java data types.

*Version 1.1: Nov 17, 2012 9:34 am +0000

[†]<http://creativecommons.org/licenses/by/3.0/>

2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools (*as a minimum*) to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) (See <http://www.oracle.com/technetwork/java/javase/downloads/index>.¹)
- Documentation for the Sun/Oracle Java Development Kit (JDK) (See <http://download.oracle.com/javase/7/docs/api/>²)
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures while you are reading about them.

2.3.1 Figures

- Figure 1 (p. 6) . Range of values for whole-number types.
- Figure 2 (p. ??) . Definition of floating point.
- Figure 3 (p. 7) . Different ways to represent 623.57185.
- Figure 4 (p. 8) . Relationships between multiplicative factors and exponentiation.
- Figure 5 (p. 9) . Other ways to represent the same information.
- Figure 6 (p. 9) . Still other ways to represent 623.57185.
- Figure 7 (p. 11) . Range of values for floating-point types.
- Figure 8 (p. 12) . Example of the use of the boolean type.

3 Discussion

3.1 Introduction

Type-sensitive languages

Java and some other modern programming languages make heavy use of a concept that we refer to as *type* , or *data type* .

We refer to those languages as *type-sensitive languages* . Not all languages are type-sensitive languages. In particular, some languages hide the concept of type from the programmer and automatically deal with type issues behind the scenes.

So, what do we mean by type?

One analogy that comes to my mind is international currency. For example, many years ago, I spent a little time in Japan and quite a long time on an island named Okinawa (*Okinawa is now part of Japan*) .

Types of currency

At that time, as now, the type of currency used in the United States was the dollar. The type of currency used in Japan was the yen, and the type of currency used on the island of Okinawa was also the yen. However, even though two of the currencies had the same name, they were different types of currency, as determined by the value relationships among them.

¹<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

²<http://download.oracle.com/javase/7/docs/api/>

The exchange rate

As I recall, at that time, the exchange rate between the Japanese yen and the U.S. dollar was 360 yen for each dollar. The exchange rate between the Okinawan yen and the U.S. dollar was 120 yen for each dollar. This suggests that the exchange rate between the Japanese yen and the Okinawan yen would have been 3 Japanese yen for each Okinawan yen.

Analogous to different types of data

So, why am I telling you this? I am telling you this to illustrate the concept that different types of currency are roughly analogous to different data types in programming.

Purchasing transactions were type sensitive

In particular, because there were three different types of currency involved, the differences in the types had to be taken into account in any purchasing transaction to determine the price in that particular currency. In other words, the purchasing process was sensitive to the type of currency being used for the purchase (*type sensitive*).

Different types of data

Type-sensitive programming languages deal with different types of data. Some data types such as type **int** involve whole numbers only (*no fractional parts are allowed*).

Other data types such as **double** involve numbers with fractional parts.

Some data types conceptually have nothing to do with numeric values, but deal only with the concept of true or false (**boolean**) or with the concept of the letters of the alphabet and the punctuation characters (**char**).

Type specification

For every different type of data used with a particular programming language, there is a specification somewhere that defines two important characteristics of the type:

1. What is the set of all possible data values that can be stored in an instance of the type (*we will learn some other names for instance later*) ?
2. Once you have an instance of the type, what are the operations that you can perform on that instance alone, or in combination with other instances?

What do I mean by an instance of a type?

Think of the type specification as being analogous to the plan or blueprint for a model airplane. Assume that you build three model airplanes from the same set of plans. You will have created three instances of the plans.

We might say that an instance is the physical manifestation of a plan or a type.

Using mixed types

Somewhat secondary to the specifications for the different types, but also extremely important, is a set of rules that define what happens when you perform an operation involving mixed types (*such as making a purchase using some yen currency in combination with some dollar currency*).

The short data type

For example, in addition to the integer type **int**, there is a data type in Java known as **short**. The **short** type is also an integer type.

If you have an instance of the **short** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -32,768 to +32,767.

This constitutes a set of 65,536 different values, including the value zero. No other value can be stored in an instance of the type **short**. For example, you cannot store the value 35,000 in an instance of the type **short** in Java. If you need to store that value, you will need to use some type other than **short**.

Kind of like an odometer

This is somewhat analogous to the odometer in your car (*the thing that records how many miles the car has been driven*). For example, depending on the make and model of car, there is a specified set of values that can appear in the odometer. The value that appears in the odometer depends on how many miles your car has been driven.

It is fairly common for an odometer to be able to store and to display the set of all positive values ranging from zero to 99999. If your odometer is designed to store that set of values and if you drive your car more than 99999 miles, it is likely that the odometer will roll over and start back at zero after you pass the 99999-mile mark. In other words, that particular odometer does not have the ability to store a value of 100,000 miles. Once you pass the 99999-mark, the data stored in the odometer is corrupt.

Now let's return to the Java type named short

Assume that you have two instances of the type `short` in a Java program. What are the operations that you can perform on those instances? For example:

- You can add them together.
- You can subtract one from the other.
- You can multiply one by the other.
- You can divide one by the other.
- You can compare one with the other to determine which is algebraically larger.

There are some other operations that are allowed as well. In fact, there is a well-defined set of operations that you are allowed to perform on those instances. That set of operations is defined in the specification for the type `short`.

What if you want to do something different?

However, if you want to perform an operation that is not allowed by the type specification, then you will have to find another way to accomplish that purpose.

For example, some programming languages allow you to raise whole-number types to a power (*examples: four squared, six cubed, nine to the fourth power, etc.*). However, that operation is not allowed by the Java specification for the type `short`. If you need to do that operation with a data value of the Java `short` type, you must find another way to do it.

Two major categories of type

Java data types can be subdivided into two major categories:

- Primitive types
- User-defined or reference types

These categories are discussed in more detail in the following sections.

3.2 Primitive types

Java is an extensible programming language

What this means is that there is a core component to the language that is always available. Beyond this, individual programmers can extend the language to provide new capabilities. The primitive types discussed in this section are the types that are part of the core language. A later section will discuss user-defined types that become available when a programmer extends the language.

More subdivision

It seems that when teaching programming, I constantly find myself subdividing topics into sub-topics. I am going to subdivide the topic of Primitive Types into four categories:

- Whole-number types
- Floating-point types
- Character types
- Boolean types

Hopefully this categorization will make it possible for me to explain these types in a way that is easier for you to understand.

3.2.1 Whole-number types

The whole-number types, often called *integer* types, are relatively easy to understand. These are types that can be used to represent data without fractional parts.

Applesauce and hamburger

For example, consider purchasing applesauce and hamburger. At the grocery store where I shop, I am allowed to purchase cans of applesauce only in whole-number or integer quantities.

Can purchase integer quantities only

For example, the grocer is happy to sell me one can of applesauce and is even happier to sell me 36 cans of applesauce. However, she would be very unhappy if I were to open a can of applesauce in the store and attempt to purchase 6.3 cans of applesauce.

Counting doesn't require fractional parts

A count of the number of cans of applesauce that I purchase is somewhat analogous to the concept of whole-number data types in Java. Applesauce is not available in fractional parts of cans (*at my grocery store*).

Fractional pounds of hamburger are available

On the other hand, the grocer is perfectly willing to sell me 6.3 pounds of hamburger. This is somewhat analogous to *floating-point data types* in Java.

Accommodating applesauce and hamburger in a program

Therefore, if I were writing a program dealing with quantities of applesauce and hamburger, I might elect to use a whole number type to represent cans of applesauce and to use a floating-point type to represent pounds of hamburger.

Different whole-number types

In Java, there are four different whole-number types:

- byte
- short
- int
- long

(The char type is also a whole number type, but since it is not intended to be used for arithmetic, I discuss it later as a character type.)

The four types differ primarily in terms of the range of values that they can accommodate and the amount of computer memory required to store instances of the types.

Differences in operations?

Although there are some subtle differences among the four whole-number types in terms of the operations that you can perform on them, I will defer a discussion of those differences until a more advanced module.

*(For example some operations require instances of the **byte** and **short** types to be converted to type **int** before the operation takes place.)*

Algebraically signed values

All four of these types can be used to represent algebraically signed values ranging from a specific negative value to a specific positive value.

Range of the byte type

For example, the **byte** type can be used to represent the set of whole numbers ranging from -128 to +127 inclusive. *(This constitutes a set of 256 different values, including the value zero.)*

The **byte** type cannot be used to represent any value outside this range. For example, the **byte** type cannot be used to represent either -129 or +128.

No fractional parts allowed by the byte type

Also, the **byte** type cannot be used to represent fractional values within the allowable range. For example, the byte type cannot be used to represent the value of 63.5 or any other value that has a fractional part.

Like a strange odometer

To form a crude analogy, the byte type is sort of like a strange odometer in a new (*and unusual*) car that shows a mileage value of -128 when you first purchase the car. As you drive the car, the negative values shown on the odometer increment toward zero and then pass zero. Beyond that point they increment up toward the value of +127.

Oops, numeric overflow!

When the value passes (*or attempts to pass*) +127 miles, something bad happens. From that point forward, the value shown on the odometer is not a reliable indicator of the number of miles that the car has been driven.

Ranges for each of the whole-number types

Figure 1 (p. 6) shows the range of values that can be accommodated by each of the four whole-number types supported by the Java programming language:

Range of values for whole-number types.

```
byte
-128 to +127

short
-32768 to +32767

int
-2147483648 to +2147483647

long
-9223372036854775808 to +9223372036854775807
```

Figure 1: Range of values for whole-number types.

Can represent some fairly large values

As you can see, the **int** and **long** types can represent some fairly large values. However, if your task involves calculations such as distances in interstellar space, these ranges probably won't accommodate your needs. This will lead you to consider using the *floating-point* types discussed in the upcoming sections. I will discuss the operations that can be performed on whole-number types more fully in future modules.

3.2.2 Floating-point types

Floating-point types are a little more complicated than whole-number types. I found the definition of floating-point shown in Figure 2 (p. ??) in the *Free On-Line Dictionary of Computing* at this URL ³.

³<http://foldoc.org/floating+point>

A number representation consisting of a mantissa, M, an exponent, E, and an (assumed) radix (or "base") . The number

Figure 2: Definition of floating point.

So what does this really mean?

Assuming a base or radix of 10, I will attempt to explain it using an example.

Consider the following value:

623.57185

I can represent this value in any of the ways shown in Figure 3 (p. 7) (where * indicates multiplication).

Different ways to represent 623.57185.

.62357185*1000
 6.2357185*100
 62.357185*10
 623.57185*1
 6235.7185*0.1
 62357.185*0.01
 623571.85*0.001
 6235718.5*0.0001
 62357185.*0.00001

Figure 3: Different ways to represent 623.57185.

In other words, I can represent the value as a mantissa (62357185) multiplied by a factor where the purpose of the factor is to represent a left or right shift in the position of the decimal point.

Now consider the factor

Each of the factors shown in Figure 3 (p. 7) represents the value of ten raised to some specific power, such as ten squared, ten cubed, ten raised to the fourth power, etc.

Exponentiation

If we allow the following symbol (^) to represent exponentiation (raising to a power) and allow the following symbol (/) to represent division, then we can write the values for the above factors in the ways shown in Figure 4 (p. 8) .

Note in particular the characters following the first equal character (=) on each line, which I will refer to later as the exponents.

Relationships between multiplicative factors and exponentiation.

$$\begin{aligned}1000 &= 10^{+3} = 1*10*10*10 \\100 &= 10^{+2} = 1*10*10 \\10 &= 10^{+1} = 1*10 \\1 &= 10^{+0} = 1 \\0.1 &= 10^{-1} = 1/10 \\0.01 &= 10^{-2} = 1/(10*10) \\0.001 &= 10^{-3} = 1/(10*10*10) \\0.0001 &= 10^{-4} = 1/(10*10*10*10) \\0.00001 &= 10^{-5} = 1/(10*10*10*10*10)\end{aligned}$$

Figure 4: Relationships between multiplicative factors and exponentiation.

In the above notation, the term 10^{+3} means 10 raised to the third power.

The zeroth power

By definition, the value of any value raised to the zeroth power is 1. (*Check this out in your high-school algebra book.*)

The exponent and the factor

Hopefully, at this point you will understand the relationship between the exponent and the factor introduced earlier in Figure 3 (p. 7) .

Different ways to represent the same value

Having reached this point, by using substitution, I can rewrite the original set of representations (p. 7) of the value 623.57185 in the ways shown in Figure 5 (p. 9) .

(It is very important to for you to understand that these are simply different ways to represent the same value.)

Other ways to represent the same information.



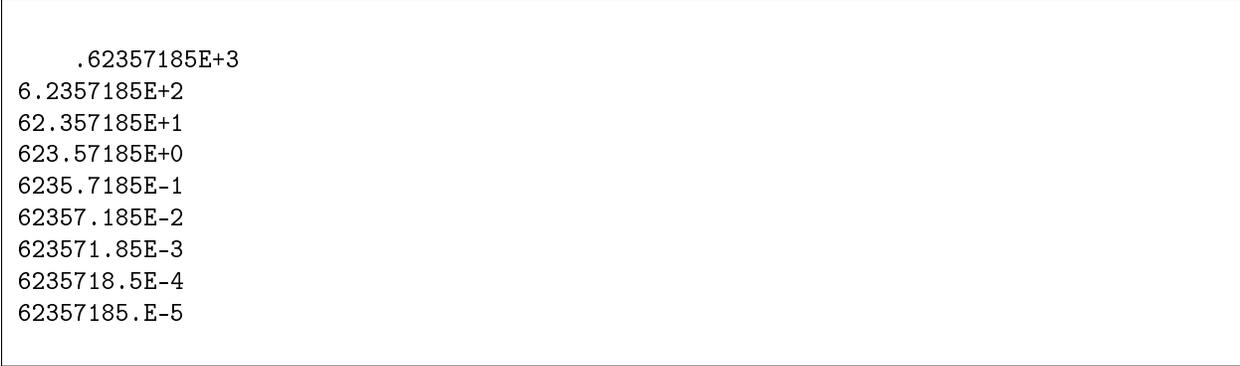
```
.62357185*10^+3  
6.2357185*10^+2  
62.357185*10^+1  
623.57185*10^+0  
6235.7185*10^-1  
62357.185*10^-2  
623571.85*10^-3  
6235718.5*10^-4  
62357185.*10^-5
```

Figure 5: Other ways to represent the same information.

A simple change in notation

Finally, by making a simplifying change in notation where I replace ($*10^$) by (E) I can rewrite the different representations of the value of 623.57185 in the ways shown in Figure 6 (p. 9) .

Still other ways to represent 623.57185.



```
.62357185E+3  
6.2357185E+2  
62.357185E+1  
623.57185E+0  
6235.7185E-1  
62357.185E-2  
623571.85E-3  
6235718.5E-4  
62357185.E-5
```

Figure 6: Still other ways to represent 623.57185.

Getting the true value

Range of values for floating-point types.

<pre>float 1.4E-45 to 3.4028235E38 double 4.9E-324 to 1.7976931348623157E308</pre>

Figure 7: Range of values for floating-point types.

I will discuss the operations that can be performed on floating-point types in a future module.

3.2.3 The character type

Computers deal only in numeric values. They don't know how to deal directly with the letters of the alphabet and punctuation characters. This gives rise to a type named **char**.

Purpose of the char type

The purpose of the character type is to make it possible to represent the letters of the alphabet, the punctuation characters, and the numeric characters internally in the computer. This is accomplished by assigning a numeric value to each character, much as you may have done to create secret codes when you were a child.

A single character type

Java supports a single character type named **char**. The char type uses a standard character representation known as **Unicode** to represent up to 65,535 different characters.

Why so many characters?

The reason for the large number of possible characters is to make it possible to represent the characters making up the alphabets of many different countries and many different spoken languages.

What are the numeric values representing characters?

As long as the characters that you use in your program appear on your keyboard, you usually don't have a need to know the numeric value associated with the different characters. If you are curious, however, the upper-case A is represented by the value 65 in the Unicode character set.

Representing a character symbolically

In Java, you usually represent a character in your program by surrounding it with apostrophes as shown below:

```
'A'
```

The Java programming tools know how to cross reference that specific character symbol against the Unicode table to obtain the corresponding numeric value. *(A discussion of the use of the **char** type to represent characters that don't appear on your keyboard is beyond the scope of this module.)*

I will discuss the operations that can be performed on the **char** type in a future module.

3.2.4 The boolean type

The boolean type is the simplest type supported by Java. It can have only two values:

- true
- false

Generally speaking, about the only operations that can be directly applied to an instance of the **boolean** type are to change it from **true** to **false** , and vice versa. However, the **boolean** type can be included in a large number of somewhat higher-level operations.

The **boolean** type is commonly used in some sort of a test to determine what to do next, such as that shown in Figure 8 (p. 12) .

Example of the use of the boolean type.

```
Perform a test that returns a value of type boolean.
if that value is true,
    do one thing
otherwise (meaning that value is false)
    do a different thing
```

Figure 8: Example of the use of the boolean type.

I will discuss the operations that can be performed on the boolean type in more detail in a future module.

3.3 User-defined or reference types

Extending the language

Java is an *extensible* programming language. By this, I mean that there is a core component to the language that is always available. Beyond the core component, different programmers can extend the language in different ways to meet their individual needs.

Creating new types

One of the ways that individual programmers can extend the language is to create new types. When creating a new type, the programmer must define the set of values that can be stored in an instance of the type as well as the operations that can be performed on instances of the type.

No magic involved

While this might initially seem like magic, once you get to the heart of the matter, it is really pretty straightforward. New types are created by combining instances of primitive types along with instances of other user-defined types. In other words, the process begins with the primitive types explained earlier and builds upward from there.

An example

For example, a **String** type, which can be used to represent a person's last name, is just a grouping of a bunch of instances of the primitive **char** or character type.

A user-defined **Person** type, which could be used to represent both a person's first name and their last name, might simply be a grouping of two instances of the user-defined **String** type.

Differences

The biggest conceptual difference between the **String** type and the **Person** type is that the **String** type is contained in the standard Java library while the **Person** type isn't in that library. However, you could put it in a library of your own design if you choose to do so.

Removing types

You could easily remove the **String** type from your copy of the standard Java library if you choose to do so, although that would probably be a bad idea. However, you cannot remove the primitive **double** type from the core language without making major modifications to the language.

The company telephone book

A programmer responsible for producing the company telephone book might create a new type that can be used to store the first and last names along with the telephone number of an individual. That programmer might choose to give the new type the name **Employee**.

The programmer could create an instance of the **Employee** type to represent each employee in the company, populating each such instance with the name and telephone number for an individual employee.

(At this point, let me sneak a little jargon in and tell you that we will be referring to such instances as objects.)

A comparison operation

The programmer might define one of the allowable operations for the new **Employee** type to be a comparison between two objects of the new type to determine which is greater in an alphabetical sorting sense. This operation could be used to sort the set of objects representing all of the employees into alphabetical order. The set of sorted objects could then be used to print a new telephone book.

A name-change operation

Another allowable operation that the programmer might define would be the ability to change the name stored in an object representing a particular employee. For example when Suzie Smith marries Tom Jones, she might elect to thereafter be known as

- Suzie Smith
- Suzie Jones,
- Suzie Smith-Jones,
- Suzie Jones-Smith, or
- something entirely different.

In this case, there would be a need to modify the object that represents Suzie in order to reflect her newly-elected surname. *(Or perhaps Tom Jones might elect to thereafter be known as Tom Jones-Smith, in which case it would be necessary to modify the object that represents him.)*

An updated telephone book

The person charged with maintaining the database could

- use the name-changing operation to modify the object and change the name,
- make use of the sorting operation to re-sort the set of objects, and
- print and distribute an updated version of the telephone book.

Many user-defined types already exist

Unlike the primitive types which are predefined in the core language, I am unable to give you much in the way of specific information about user-defined types, simply because they don't exist until a user defines them.

I can tell you, however, that when you obtain the Java programming tools from Sun, you not only receive the core language containing the primitive types, you also receive a large library containing several thousand user-defined types that have already been defined. A large documentation package is available from Sun to help you determine the individual characteristics of these user-defined types.

The most important thing

At this stage in your development as a Java programmer, the most important thing for you to know about user-defined types is that they are possible.

You can define new types. Unlike earlier procedural programming languages such as C and Pascal, you are no longer forced to adapt your problem to the available tools. Rather, you now have the opportunity to extend the tools to make them better suited to solve your problem.

The class definition

The specific Java mechanism that makes it possible for you to define new types is a mechanism known as the *class definition*.

In Java, whenever you define a new class, you are at the same time defining a new type. Your new type can be as simple, or as complex and powerful as you want it to be.

An object (*instance*) of your new type can contain a very small amount of data, or it can contain a very large amount of data. The operations that you allow to be performed on an object of your new type can be rudimentary, or they can be very powerful.

It is all up to you

Whenever you define a new class (*type*) you not only have the opportunity to define the data definition and the operations, you also have a responsibility to do so.

Much to learn and much to do

But, you still have much to learn and much to do before you will need to define new types.

There are a lot of fundamental programming concepts that we will need to cover before we seriously embark on a study involving the definition of new types.

For the present then, simply remember that such a capability is available, and if you work to expand your knowledge of Java programming one small step at a time, when we reach the point of defining new types, you will be ready and eager to do so.

3.4 Sample program

I'm not going to provide a sample program in this module. Instead, I will be using what you have learned about Java data types in the sample programs in future modules.

4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: Housekeeping material

- Module name: Jb0150: Java OOP: A Gentle Introduction to Java Data Types
- File: Jb0150.htm
- Published: November 17, 2012

NOTE: Disclaimers: Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-