Language Features, Primitive Types*

R.G. (Dick) Baldwin

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

Abstract

This lesson discusses the ramifications of types short, long, byte, char, and float not being included in the Java subset for the AP Computer Science A exam.

1 Table of Contents

- Preface (p. 1)
  - General (p. 1)
  - Viewing tip (p. 2)
  - Supplemental material (p. 2)
- Discussion (p. 2)
  - Different types of data (p. 2)
  - Type specifications (p. 3)
  - Primitive types (p. 4)
  - Floating-point types (p. 6)
  - The boolean type (p. 8)
  - The char type (p. 8)
- Resources (p. 9)
- Miscellaneous (p. 9)

2 Preface

2.1 General

This tutorial lesson is part of a series of lessons dedicated to the AP Computer Science A Exam [1] .

The purpose of the series is to help you to better understand the *Java Subset* specifications for the *Advanced Placement Computer Science A* exam.

For example, if you access the AP Computer Science Course Description [2] and scroll down to Appendix A titled *AP Computer Science Java Subset* , you will find the following **specification** :

*Version 1.3: Mar 3, 2011 1:27 pm -0600

†http://creativecommons.org/licenses/by/3.0/

[1] http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/4483.html

[2] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf

NOTE: 1. The primitive types int, double, and boolean are part of the AP Java subset.

The other primitive types short, long, byte, char, and float are not in the subset.

In particular, students need not be aware that strings are composed of char values. Introducing char does not increase the expressiveness of the subset.

Students already need to understand string concatenation, String.substring, and String.equals. *(This is a requirement in a different part of the course description* [3] *.)*

Not introducing char avoids complexities with the char/int conversions and confusion between "x" and 'x'.

(Note that I inserted some explanatory text in parentheses and also inserted some line breaks for clarity. Also note that this is the wording that appeared on the website on March 1, 2011. It may change over time.)

May be clear to some, but not to others

While some students and their homeschool parents may know exactly what is meant by this text, others may not. Therefore, my objective will be to elaborate on and attempt to clarify such specifications from a technical viewpoint to help those students and their parents get a better understanding of where they need to concentrate their efforts.

2.2 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.

- Figure 1 (p. ??) . Calculating the number of cans of paint to purchase.
- Figure 2 (p. 6) . Value ranges for integer types.
- Figure 3 (p. ??) . Representing a large range of values.
- Figure 4 (p. 8) . Value range for floating-point types.

2.3 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [4] .

3 Discussion

Java, C++, C#, 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 *strongly typed* or *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.

3.1 Different types of data

Type-sensitive programming languages deal explicitly with different types of data. Some data types involve whole-numbers only *(no fractional parts are allowed)* . We generally refer to these as *integer* types.

Other data types involve numbers with fractional parts. We generally refer to these types as *floating-point* types, because a decimal point can float back and forth, separating the whole-number part from the fractional part. *(After a while, we get lazy and refer to these simply as floating types.)*

An example of integer and floating-point types

[3] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf
[4] http://www.dickbaldwin.com/toc.htm

The scenario in Figure 1 illustrates integer and floating-point types.

The number of tables is an integer type. We don't want to paint 14.6 tables or 15.7 tables. We want to paint exactly 15 tables. *(We don't want to have a fractional part of a table left unpainted.)*

We know that one can of paint will cover 3.6 tables. The number of tables that can be painted with one can of paint is a floating-point value because it contains a whole-number part and a fractional part.

A little arithmetic tells us that 4.167 cans of paint will be required to paint all 15 tables. This is also a floating-point value, because it has a whole-number part and a fractional part.

However, the man at the hardware store is unwilling to sell us 4.167 cans of paint. He requires us to specify the number of cans of paint as an integer value. In this case, we will need to purchase 5 cans of paint in order to have enough paint available to paint all 15 tables, with a little paint left over.

The number of tables is an integer type. We don't want to paint 14.6 tables or 15.7 tables. We want to paint exactly 15 tables. *(We don't want to have a fractional part of a table left unpainted.)*

We know that one can of paint will cover 3.6 tables. The number of tables that can be painted with one can of paint is a floating-point value because it contains a whole-number part and a fractional part.

A little arithmetic tells us that 4.167 cans of paint will be required to paint all 15 tables. This is also a floating-point value, because it has a whole-number part and a fractional part.

However, the man at the hardware store is unwilling to sell us 4.167 cans of paint. He requires us to specify the number of cans of paint as an integer value. In this case, we will need to purchase 5 cans of paint in order to have enough paint available to paint all 15 tables, with a little paint left over.

Consider the problem of determining the number of cans of paint that must be purchased in order to paint all 15 tables in

Figure 1: Calculating the number of cans of paint to purchase.

Other types of data

Although all data in a computer is stored in numeric format, some data types conceptually have nothing to do with numeric values, but deal only with the concept of *true* or *false* or with the concept of the letters of the alphabet and the punctuation characters. I will have more to say about these types later.

3.2 Type specifications

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. The set of all possible data values that can be stored in an instance of the type. *(We will learn more about the concept of an instance shortly.)*
2. The operations that you can perform on that instance alone, or in combination with other instances. *(For example, operations include addition, subtraction, multiplication, division, etc.)*

What do I mean by *instance* ?

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 single set of plans.

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

An example, the *short* data type

There is a data type in Java, C++, and C# known as **short** . If you have an instance of the short type in Java, the set of all possible values that you can store in that instance is the set of all 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 have to use some type other than **short** .

Sort of like an odometer

The **short** type is an integer type. Integer types are 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 99,999.

If your odometer is designed to store that set of values and if you drive your car more than 99,999 miles, it is likely that the odometer will roll over and start back at zero after you pass the 99,999-mile mark. In other words, that particular type of odometer does not have the ability to store a value of 100,000 or more miles. Once you pass the 99,999 mark, the data stored in the odometer is corrupt. It no longer represents the true number of miles for which the car has been driven.

A word or two about operations

Assume that you have two instances of the type **short** in a Java program. Here are some of the operations that you can perform on those instances:

- 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 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, and that set of operations is defined in the specification for the type **short** .

What if you want to do something different?

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 *(class)* types

The remainder of this lesson will deal with primitive types.

3.3 Primitive types

Java, C++, and C# are extensible programming languages.

This means 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 are the types that are part of the core language.

Four categories of primitive types

I am going to subdivide the topic of primitive types into four categories:

- Whole-number *(integer)* types
- Floating-point types
- The boolean type
- The character type

Purchasing applesauce and hamburger

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

For example, the grocer is happy to sell me one jar of applesauce and is even happier to sell me 36 jars of applesauce.

However, she would be very unhappy if I were to open a jar of applesauce in the store and attempt to purchase 6.3 jars of applesauce. A count of the number of jars 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 jars, at least not at the store where I purchase my groceries.

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.

Therefore, if I were writing a program dealing with quantities of applesauce and hamburger, I might elect to use a whole-number type *(such as the type* ***int*** *)* to represent jars of applesauce and to use a floating-point type *(such as the type* ***double*** *)* to represent pounds of hamburger.

Different whole-number types

Four different whole-number types are built into the Java language:

- byte
- short
- int
- long

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.

Although there are some subtle differences among these four types in terms of the operations that you can perform on them, by excluding **byte** , **short** , and **long** from the exam *(keeping only type* ***int*** *)* , the authors of the exam have signaled that they don't require you to know about those differences.

Algebraically signed values

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

Value ranges for the different integer types

The four integer types support the value ranges shown in Figure 2.

Value ranges for integer types.

- byte: -128 to + 127
- short: -32768 to + 32767
- int: -2147483648 to +2147483647
- long: -9223372036854775808 to +9223372036854775807

Figure 2: Value ranges for integer 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 *(or the U.S. national debt)* , these ranges probably won't accommodate your needs. This will lead you to consider using the floating-point types discussed later.

Exclude types *byte* , *short* , and *long*

As you saw earlier , the authors of the exam have excluded types **byte** , **short** , and **long** keeping only type **int** *(of the whole-number types)* in the exam.

If only one whole-number type is to be included in the exam, type **int** is the logical choice to keep. In many ways, it is the default whole-number type in Java. For example, a literal value, such as 1279, is considered to be type **int** unless you write code to force it to be treated as some other type.

3.4 Floating-point types

Floating-point types are a little more complicated than whole-number types. I found the following definition of floating-point in the Free On-Line Dictionary of Computing [5] :

*A number representation consisting of a mantissa, M, an exponent, E, and a radix (or "base"). The number represented is M*R^E where R is the radix.*

What does this really mean?

I'm not going to get into a long discussion about floating-point values at this point. If you are really interested, you will find a detailed discussion in lesson # 905 at http://www.dickbaldwin.com/tocint.htm [6] .

Advantages of floating-point types

One advantage of floating-point types is that they can be used to maintain fractional parts in data values.

Another advantage of floating-point types is that a very large range of values can be represented using a reasonably small amount of computer memory for storage of the values.

For example *(assuming that I counted the number of digits correctly)* Figure 3 shows how to represent a very large value and a very small value as a floating-point type.

can be represented as

6.2357185E+37

The very small value:

[5] http://foldoc.org/floating+point

[6] http://www.dickbaldwin.com/tocint.htm

0.00000000000000000000000000000062357185
can be represented as 6.2357185E-30

can be represented as
6.2357185E+37
The very small value:
0.00000000000000000000000000000062357185
can be represented as 6.2357185E-30

The very large value: 623571850000000000000000000000000000.0 can be represented as 6.2357185E+37 The very small

Figure 3: Representing a large range of values.

When would you use floating-point?

If you happen to be working in an area where you need to keep track of fractional parts *(such as the amount of hamburger in a package)* , have to work with extremely large numbers *(distances between galaxies)* , or have to work with extremely small values *(the size of atomic particles)* , then you will need to use the floating-point types.

Also, if you will be doing computations involving division, floating-point is often easier to work with because division with integer types can be problematic. *(The remainder is discarded in integer division, which can cause arithmetic accuracy problems.)*

Two floating-point types

Java supports two different floating-point types:

- float
- double

These two types differ primarily in terms of the range of values that they can support and the number of significant digits used in the representation of those values.

Figure 4 shows the smallest and largest values that can be accommodated by each of the floating-point types. Values of either type can be either positive or negative.

Value range for floating-point types.

- float: 1.4E-45 to 3.4028235E38
- double: 4.9E-324 to 1.7976931348623157E308

Figure 4: Value range for floating-point types.

Exclude type *float*

As you saw earlier , the authors of the exam have excluded type **float** including only type **double** *(of the two floating-point types)* in the exam. If one of the floating-point types is to be excluded, type **double** is the logical choice to keep. In many ways, it is the default floating-point type in Java. For example, a literal value, such as 3.14159, is considered to be type **double** unless you write code to force it to be treated as type **float** .

3.5 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 applied to an instance of the **boolean** type are to change it from true to false, and vice versa. The boolean type is commonly used in some kind of a test to determine what to do next.

Keep type *boolean*

As you saw earlier , the authors of the exam have included type **boolean** in the exam.

In this case, there was no opportunity to exclude other similar types. The **boolean** type is one of a kind and you will need to know how to use it when taking the exam.

3.6 The char type

As you saw earlier , the authors of the exam elected not to include type **char** in the exam, and they gave several reasons why.

However, just in case you are curious about this type, I will provide some information about it in the following paragraphs.

The character type *char*

Computers deal only in numeric values. They don't know how to deal directly with the letters of the alphabet and punctuation characters.

The purpose of the character type **char** 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. *(For example, in Java an upper-case A character is represented by the numeric value 65, whereas the upper-case B is represented by the numeric value 66.)*

A single character type

Java supports a single character type named **char** . The **char** type uses a standard character representation known as Unicode [7] 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 languages.

Representing a character symbolically

Although each character is represented internally by a numeric value, 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 values associated with the different characters.

In Java, you usually represent a character to the program by surrounding it with apostrophes as follows: '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 lesson.)*

4 Resources

I will publish a list containing links to resources in the following module and will update and add to the list as the remaining modules in this collection are published:

http://cnx.org/content/m37140/latest/ [8]

5 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Java Subset, Primitive Types
- File: Hs01004.htm
- Keywords:
  - Advanced placement
  - AP Computer Science
  - Java subset
  - homeschool

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 income from the Connexions website even if you purchase the PDF version of the module.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX, and have no affiliation with the College Board [9] .

-end-

[7] http://www.unicode.org/
[8] http://cnx.org/content/m37140/latest/
[9] http://apcentral.collegeboard.com/apc/Controller.jpf