AP Computer Science A, Clarification of the Java Subset

By:
Richard Baldwin

AP Computer Science A, Clarification of the Java Subset

By:
Richard Baldwin

Online:
< http://cnx.org/content/col11279/1.4/ >

C O N N E X I O N S

Rice University, Houston, Texas

This selection and arrangement of content as a collection is copyrighted by Richard Baldwin. It is licensed under the Creative Commons Attribution 3.0 license (http://creativecommons.org/licenses/by/3.0/).
Collection structure revised: March 3, 2011
PDF generated: October 29, 2012
For copyright and attribution information for the modules contained in this collection, see p. 30.

Table of Contents

1 Language Features - Introduction 1
2 Language Features - Resources 3
3 Language Features, Primitive Types 5
4 Language Features, Arithmetic Operators 15
Index 29
Attributions 30

Chapter 1

Language Features - Introduction[1]

1.1 Table of Contents

- General (p. 1)
- Resources (p. 2)
- Miscellaneous (p. 2)

1.2 General

This tutorial lesson is the first in a series of lessons dedicated to the AP Computer Science A Exam [2] .

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 [3] and scroll down to Appendix A titled *AP Computer Science Java Subset* , you will find material such as the following:

> NOTE: 13. The main method and command-line arguments are not in the subset. The AP Computer Science Development Committee does not prescribe any particular approach for program invocation. In free-response questions, students are not expected to invoke programs. In case studies, program invocation with main may occur, but the main method will be kept very simple.

(Note: This is the wording that appeared on the website on March 1, 2011. It may change over time.)

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.

In some cases, I will simply elaborate verbally on on the specifications. In other cases, I will also provide program code that illustrates some of what the student is expected to know.

In other cases, I will also provide code that illustrates some of what the student is apparently not expected to know.

In the next lesson, I will begin by elaborating on the following specifications from the above-mentioned course description [4] .

> 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

[1] This content is available online at <http://cnx.org/content/m37133/1.3/>.
[2] http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/4483.html
[3] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf
[4] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf

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

1.3 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/ [5]

1.4 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Language Features - Introduction
- File: Hs01000.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 [6] .

-end-

[5] http://cnx.org/content/m37140/latest/
[6] http://apcentral.collegeboard.com/apc/Controller.jpf

Chapter 2

Language Features - Resources[1]

2.1 Table of Contents

- Preface (p. 3)
- Resources (p. 3)
- Miscellaneous (p. 3)

2.2 Preface

This tutorial lesson is part of a series of lessons dedicated to the AP Computer Science A Exam [2] .

The purpose of this document is to provide a consolidated list of links to online resources that the reader of the other lessons in the series might find useful. I will continue to add links to this document as the other lessons in the series are created and published.

2.3 Resources

- Baldwin's online Java tutorials [3]
- AP Computer Science A Course Home Page [4]
- AP Computer Science Course Description [5]
- Free On-Line Dictionary of Computing [6]
- The Unicode Consortium [7]

2.4 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Language Features - Resources
- File: Hs01002.htm

[1] This content is available online at <http://cnx.org/content/m37140/1.2/>.
[2] http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/4483.html
[3] http://www.dickbaldwin.com/tocint.htm
[4] http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/4483.html
[5] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf
[6] http://foldoc.org/floating+point
[7] http://www.unicode.org/

- 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 [8] .

-end-

[8] http://apcentral.collegeboard.com/apc/Controller.jpf

Chapter 3

Language Features, Primitive Types[1]

3.1 Table of Contents

- Preface (p. 5)
  - General (p. 5)
  - Viewing tip (p. 6)
  - Supplemental material (p. 6)
- Discussion (p. 6)
  - Different types of data (p. 6)
  - Type specifications (p. 7)
  - Primitive types (p. 8)
  - Floating-point types (p. 10)
  - The boolean type (p. 12)
  - The char type (p. 12)
- Resources (p. 13)
- Miscellaneous (p. 13)

3.2 Preface

3.2.1 General

This tutorial lesson is part of a series of lessons dedicated to the AP Computer Science A Exam [2] .

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 [3] and scroll down to Appendix A titled *AP Computer Science Java Subset* , you will find the following **specification** :

> 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.

[1] This content is available online at <http://cnx.org/content/m37130/1.3/>.
[2] http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/4483.html
[3] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf

> Students already need to understand string concatenation, String.substring, and String.equals. *(This is a requirement in a different part of the course description [4] .)*
>
> 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.

3.2.2 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. 10) . Value ranges for integer types.
- Figure 3 (p. ??) . Representing a large range of values.
- Figure 4 (p. 11) . Value range for floating-point types.

3.2.3 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 [5] .

3.3 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.

3.3.1 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

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.

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

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 3.1: 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.

3.3.2 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.

3.3.3 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 3.2: 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.

3.3.4 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 [6] :

*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 [7] .

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:

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

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

0.000000000000000000000000000062357185
can be represented as 6.2357185E-30

can be represented as
6.2357185E+37
The very small value:
0.000000000000000000000000000062357185
can be represented as 6.2357185E-30

The very large value: 6235718500000000000000000000000000000.0 can be represented as 6.2357185E+37 The very small

Figure 3.3: 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 3.4: 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** .

3.3.5 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.

3.3.6 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 [8] 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.)*

[8] http://www.unicode.org/

3.4 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/ [9]

3.5 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 [10] .

-end-

[9] http://cnx.org/content/m37140/latest/
[10] http://apcentral.collegeboard.com/apc/Controller.jpf

Chapter 4

Language Features, Arithmetic Operators[1]

4.1 Table of Contents

- Preface (p. 15)
  - General (p. 15)
  - Viewing tip (p. 16)
    - Figures (p. 16)
    - Listings (p. 16)
  - Supplemental material (p. 16)
- Discussion and sample code (p. 16)
  - Integer division (p. 17)
  - Arithmetic overflow (p. 19)
  - Integer divide by zero (p. 20)
  - Floating divide by zero (p. 21)
  - Mixed-type arithmetic (p. 22)
  - Signed modulus operations (p. 23)
  - Modulus with zero divisor (p. 24)
  - Modulus operator with floating types (p. 25)
  - A totally incorrect modulus result (p. 26)
  - Not a Number (p. 27)
- Run the programs (p. 27)
- Resources (p. 27)
- Miscellaneous (p. 28)

4.2 Preface

4.2.1 General

This tutorial lesson is part of a series of lessons dedicated to the AP Computer Science A Exam [2] .

The purpose of the series is to help you to better understand some of the material that you will find on the website that provides information for the exam.

[1] This content is available online at <http://cnx.org/content/m37146/1.2/>.

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

For example, if you access the AP Computer Science Course Description [3] and scroll down to Appendix A titled *AP Computer Science Java Subset* , you will find the following **specification** :

NOTE: 2. Arithmetic operators: +, -, *, /, % are part of the AP Java subset.

(Note: 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.

In this lesson, I will provide program code that illustrates some of what the student is expected to know about the arithmetic operators.

4.2.2 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 and listings while you are reading about them.

4.2.2.1 Figures

- Figure 1 (p. ??) . Arithmetic Operators.
- Figure 2 (p. 21) . Runtime error.
- Figure 3 (p. 25) . Runtime error.

4.2.2.2 Listings

- Listing 1 (p. 17) . Integer division.
- Listing 2 (p. 19) . Arithmetic overflow.
- Listing 3 (p. 20) . Integer divide by zero.
- Listing 4 (p. 21) . Floating divide by zero.
- Listing 5 (p. 22) . Mixed-type arithmetic.
- Listing 6 (p. 23) . Signed modulus operations.
- Listing 7 (p. 24) . Modulus with zero divisor.
- Listing 8 (p. 25) . Modulus operator with floating types.
- Listing 9 (p. 26) . A totally incorrect modulus result.
- Listing 10 (p. 27) . Not a number (NaN).

4.2.3 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] .

4.3 Discussion and sample code

According to Sun's Summary of Operators website [5] , the Java arithmetic operators are shown in Figure 1.

- Subtraction operator

* Multiplication operator

[3] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf
[4] http://www.dickbaldwin.com/toc.htm
[5] http://download.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html

/ Division operator
% Remainder operator

- Subtraction operator
* Multiplication operator
/ Division operator
% Remainder operator

+ Additive operator (also used for String concatenation) - Subtraction operator * Multiplication operator / Division opera

Figure 4.1: Arithmetic Operators.

Five arithmetic operators

If you compare Figure 1 with the specification (p. 16) , you will see that all of the arithmetic operators are included in the Java subset. Unlike the case for primitive variables discussed in an earlier lesson, none of the arithmetic operators are excluded from the subset. Therefore, the student is expected to understand how to use all of the arithmetic operators in order to successfully complete the exam.

As indicated in Figure 1 (p. **??**) , the + operator is also used for string concatenation, but that is another topic for another lesson.

Are the operators as innocuous as they seem?

At first glance, you might ask why there is a need to elaborate on or to clarify this specification. After all, aren't the arithmetic operators completely innocuous? The fact is that the arithmetic operators may not be as innocuous as they seem. There are several pitfalls involving these operators that can trap the unsuspecting Java programming student.

The subtraction operator is rather innocuous

There is very little to say about the - operator. It performs numeric subtraction as indicated in Figure 1. It is hard to get into trouble doing numeric subtraction.

The other operators are not so innocuous

However, the addition, multiplication, division and remainder operators probably do deserve a little more scrutiny, particularly insofar as to how they behave for different types. *(Note that the remainder operator is also known as the modulus operator.)*

4.3.1 Integer division

Let's begin with a question. Which of the following outputs, if any, is produced by the program shown in Listing 1 (p. 17) ?

- A. Compiler Error
- B. Runtime Error
- C. 33.666666
- D. 34
- E. 33

Listing 4.1: Integer division.

```
    public class Division01{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    int myVar01 = 101;
    int myVar02 = 3;
    System.out.println(myVar01/myVar02);
  }//end doWork()
}//end class definition
```

The first program

Since this the first program code that I have presented in this series of lessons, I will explain some aspects of the code before providing and explaining the answer to the question.

A violation of specification number 18

To begin with, this program, *(and many of the other programs that I will present in this series of lessons)* , violates specification number 18 in the course description [6] . That specification reads partially as follows:

"18. Visibility: In the AP Java subset, all classes are public."

Overkill

While that probably is a worthwhile requirement in large-scale project development, it is overkill for the simple programs that I will present in this series of lessons. *(Note that the class named* ***Worker*** *in Listing 1 (p. 17) is not declared* ***public*** *.)*

If I were to declare the **Worker** class **public** , it would be necessary for me to put it in a separate disk file named **Worker.java** . That would complicate my disk management efforts with regard to this series of lessons, and would also complicate your efforts in those cases where you want to copy, compile, experiment with, and execute the code.

All class definitions in the same disk file

Therefore, in most cases, I will omit the **public** declaration for all classes other than the driver class that contains the **main** method. That will allow me to put all classes belonging to a single program in a single disk file, and to provide a listing of the program in a form that you can easily copy into your IDE.

Having copied the program into your Java IDE, you can save the source code in a single disk file. Having done that, you can compile and execute the program without having to create a separate disk file for each class definition in the program.

Program organization

As you can see from Listing 1 (p. 17) , this program contains two class definitions:

- **Division01** , which is declared **public** .
- **Worker** , which is not declared **public** .

As explained above, all of the source code is contained in a single disk file named **Division01.java** .

The class named **Division01** is what I will frequently refer to as the *driver* class. In particular, it contains the **main** method, where execution begins and ends in a Java application.

A simple main method

In accordance with specification 13 of the course description [7] , the **main** method is very simple. In particular, the **main** method instantiates a new object of the class named **Worker** , and calls a method named **doWork** belonging to that object. All of the work in the program is performed by the method named **doWork** .

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

Back to the question...

Now let's get back to the question that was posed earlier (p. 17) . Which of the outputs listed above are produced by the program code in Listing 1 (p. 17) ?

Explanation

This program illustrates the integer truncation that results when the division operator is applied to operands of the integer types.

The result of simple long division

We all know that when we divide 101 by 3, the result is 33.666666 with the sixes extending out to the limit of our arithmetic accuracy. *(Confirm it using your hand calculator.)*

The result of rounding

If we round the quotient to the next closest integer, the result is 34.

Integer division does not round

However, when division is performed using operands of integer types in Java, the fractional part is simply discarded *(not rounded)* . The result is the whole number result without regard for the fractional part or the remainder. Thus, with integer division, 101/3 produces the integer value 33.

If either operand is a floating type...

If either operand is one of the floating types,

- the integer operand will be converted to the floating type,
- the quotient will be of the floating type, and
- the fractional part of the quotient will be preserved to some degree of accuracy

And the answer to the question is...

The code in Listing 1 (p. 17) displays 33 on the standard output device *(typically the command-line screen)* .

4.3.2 Arithmetic overflow

Let's continue with another question. What output is produced by the program shown in Listing 2 (p. 19) ?

- A. Compiler Error
- B. Runtime Error
- C. 2147483649
- D. -2147483647

Listing 4.2: Arithmetic overflow.

```
    public class Overflow01{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    //Integer.MAX_VALUE = 2147483647
    int myVar01 = Integer.MAX_VALUE;
    int myVar02 = 2;
    System.out.println(myVar01 + myVar02);
  }//end doWork()
}//end class definition
```

Explanation

This program illustrates a dangerous situation involving arithmetic using operands of integer types. This situation involves a condition commonly known as *integer overflow* or *arithmetic overflow* . This problem usually involves either the addition or multiplication of integer operands.

The good news

The good news about doing arithmetic using operands of integer types is that as long as the result is within the allowable value range for the wider of the integer types, the results are exact *(floating arithmetic often produces results that are not exact)* .

The bad news

The bad news about doing arithmetic using operands of integer types is that when the result is not within the allowable value range for the wider of the integer types, the results are garbage, having no usable relationship to the correct result. *(Floating arithmetic has a much higher probability of producing approximately correct results, even though the results may not be exact.)*

For this specific case...

As you will see by the answer to this question given below, when a value of 2 *(see Listing 2 (p. 19))* was added to the largest positive value that can be stored in type **int** , the incorrect result was a very large negative value.

This result is simply not correct. *(If you know how to do binary arithmetic, you can figure out how this happens, but that is a topic for another lesson on another day.)*

No safety net in this case – just garbage

Furthermore, there was no compiler error and no runtime error. The program simply produced an incorrect result with no warning.

You need to be especially careful when writing programs that perform arithmetic using operands of integer types. Otherwise, your programs may produce incorrect results without you even knowing that it is happening.

And the correct answer to the question is...

D. -2147483647

4.3.3 Integer divide by zero

What output is produced by the program in Listing 3 (p. 20) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. 11

Listing 4.3: Integer divide by zero.

```
    public class Division03{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    int myVar01 = 11;
    int myVar02 = 0;
    System.out.println(myVar01/myVar02);
  }//end doWork()
}//end class definition
```

Dividing by zero

This program attempts to divide the **int** value of 11 by the **int** value of zero.

Integer division by zero is not allowed

This code produces the runtime error shown in Figure 2 (p. 21) , which terminates the program.

Two ways to deal with this kind of problem

One way is to test the right operand before each divide operation to ensure that it isn't equal to zero, and to take appropriate action if it is.

A second *(probably preferred)* way is to use exception handling and to surround the divide operation with a **try** block, followed by a **catch** block for the type **java.lang.ArithmeticException** . The code in the catch block can be designed to deal with the problem if it occurs. *(Exception handling will be discussed in a future lesson.)*

And the answer to the question is...

The code in Listing 3 (p. 20) produces the runtime error shown in Figure 2 (p. 21) .

Runtime error.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Worker.doWork(Division03.java:13)
    at Division03.main(Division03.java:5)
```

Figure 4.2: Runtime error.

4.3.4 Floating divide by zero

What output is produced by the program shown in Listing 4 (p. 21) ?

- A. Compiler Error
- B. Runtime Error
- C. -Infinity
- D. 11

Listing 4.4: Floating divide by zero.

```
    public class Division02{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
```

```
    double myVar01 = -11;
    double myVar02 = 0;
    System.out.println(myVar01/myVar02);
  }//end doWork()
}//end class definition
```

Floating divide by zero

The program in Listing 4 (p. 21) attempts to divide the double value of -11 by the double value of zero.

No runtime error with floating divide by zero

In the case of floating types, an attempt to divide by zero does not produce a runtime error. Rather, it returns a value that the **println** method interprets and displays as either *Infinity* or *-Infinity* .

What is the actual value?

The actual value returned by the division operation in this program is the value of a static final variable in the **Double** class named **NEGATIVE_INFINITY** . *(There is also a value for **POSITIVE_INFINITY** .)*

Is this a better approach?

As you learned earlier, if you attempt to divide by zero with integer arithmetic, an exception will be thrown. Is this a better approach than throwing an exception? I will let you be the judge of that. The important thing is that you know what will happen in both cases and be prepared to deal with the consequences.

In either case, you could test the right operand before the divide to ensure that it isn't equal to zero, and then take appropriate action if it is zero.

Cannot use exception handling in this case

For floating divide by zero, you cannot handle the problem by using *try-catch* because an error or exception won't be thrown. However, you can test the result following the divide to see if it is equal to either of the infinity values mentioned above (p. 22) . Note, however, that you will need to test against the constant values shown above (p. 22) and not against the output (p. 22) produced by the **println** method.

And the correct answer to the question is...

C. -Infinity

4.3.5 Mixed-type arithmetic

What output is produced by the program shown in Listing 5 (p. 22) ?

- A. Compiler Error
- B. Runtime Error
- C. 4.294967294E9
- D. 4294967294

Listing 4.5: Mixed-type arithmetic.

```
    public class Mixed01{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    //Integer.MAX_VALUE = 2147483647
    int myIntVar = Integer.MAX_VALUE;
    System.out.println(2.0 * myIntVar);
```

```
  }//end doWork()
}//end class definition
```

This program illustrates the use of arithmetic operators with operands of different types.

Declare and initialize an int

The method named **doWork** declares a local variable of type **int** named **myIntVar** and initializes it with the largest positive value that can be stored in type **int** .

Evaluate an arithmetic expression

An arithmetic expression involving **myIntVar** is evaluated and the result is passed as a parameter to the **println** method where it is displayed on the computer screen.

Multiply by a literal double value

The arithmetic expression uses the multiplication operator (*) to multiply the integer value stored in **myIntVar** by 2.0 *(this literal operand is type* ***double*** *by default)* .

Automatic conversion to wider type

When arithmetic is performed using operands of different types, the type of the operand of the narrower type is automatically converted to the type of the operand of the wider type, and the arithmetic is performed on the basis of the wider type.

(By narrower and wider, we mean the width of the range of values that can be accommodated by the type in question.)

Result is of the wider type

The type of the result is the same as the wider type; **double** in this case.

In this case...

Because the left operand of the multiplication operation in Listing 5 (p. 22) is type **double** , the **int** value is converted to type **double** and the arithmetic is performed as type **double** . This produces a result of type **double** , causing the floating value 4.294967294E9 to be displayed on the computer screen.

And the correct answer is...

C. 4.294967294E9

4.3.6 Signed modulus operations

What output is produced by the program in Listing 6 (p. 23) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. -2

Listing 4.6: Signed modulus operations.

```
    public class Modulus01{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    int myVar01 = -11;
    int myVar02 = 3;
    System.out.println(myVar01%myVar02);
  }//end doWork()
}//end class definition
```

What is a modulus operation?

In elementary terms, we like to say that the modulus operation returns the remainder that results from a divide operation. In general terms, that is true.

Some interesting behavior

However, the modulus operation has some interesting behaviors that are illustrated in this and the next several questions. This program returns the modulus of -11 and 3, with -11 being the left operand.

What is the algebraic sign of the result?

Here is a rule:

The result of the modulus operation takes the sign of the left operand, regardless of the sign of the quotient and regardless of the sign of the right operand.

In this program, that produced a result of -2. Changing the sign of the right operand would not have changed the sign of the result.

Exercise care using the sign of the modulus result

Thus, you may need to exercise care as to how you interpret the sign of the result when you perform a modulus operation having a negative left operand.

And the correct answer is...

D. -2

4.3.7 Modulus with zero divisor

What output is produced by the program shown in Listing 7?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 11

Listing 4.7: Modulus with zero divisor.

```
    public class Modulus02{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    int myVar01 = -11;
    int myVar02 = 0;
    System.out.println(myVar01%myVar02);
  }//end doWork()
}//end class definition
```

Integer modulus involves integer divide

The modulus operation with integer operands involves an integer divide. Therefore, it is subject to the same kind of problem as an ordinary integer divide when the right operand has a value of zero.

Program produces a runtime error

In this case, the program produced the runtime error shown in Figure 3 (p. 25) , which terminated the program.

Dealing with the problem

As with integer divide, you can either test the right operand for a zero value before performing the modulus operation, or you can deal with the problem after the fact using try-catch.

And the answer is...
The code in Listing 7 (p. 24) produces the runtime error shown in Figure 3 (p. 25) .

Runtime error.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Worker.doWork(Modulus02.java:13)
    at Modulus02.main(Modulus02.java:5)
```

Figure 4.3: Runtime error.

4.3.8 Modulus operator with floating types

What output is produced by the program shown in Listing 8 (p. 25) ?

- A. Compiler Error
- B. Runtime Error
- C. -0.010999999999999996
- D. 0.010999999999999996

Listing 4.8: Modulus operator with floating types.

```
    public class Modulus03{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    double myVar01 = -0.11;
    double myVar02 = 0.033;
    System.out.println(myVar01%myVar02);
  }//end doWork()
}//end class definition
```

Modulus operator can be used with floating types
In this case, the program returns the remainder that would be produced by dividing a double value of -0.11 by a double value of 0.033 and terminating the divide operation at the beginning of the fractional part of the quotient.

Say that again
Stated differently, the result of the modulus operation for floating types is the remainder that results after

- subtracting the right operand from the left operand an integral number of times, and
- terminating the repetitive subtraction process when the result of the subtraction is less than the right operand

(Did you know that division is nothing more than repetitive subtraction and multiplication is nothing more than repetitive addition?)

Modulus result is not exact

According to my hand calculator, taking into account the fact that the left operand is negative, this operation should produce a modulus result of -0.011. As you can see from the answer that follows, the result produced by the application of the modulus operation to floating types is not exact.

And the answer to the question is...

C. -0.010999999999999996

4.4 A totally incorrect modulus result

What output is produced by the program shown in Listing 9 (p. 26) ?

- A. Compiler Error
- B. Runtime Error
- C. 0.0
- D. 1.5499999999999996

Listing 4.9: A totally incorrect modulus result.

```
    public class Modulus04{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    double myVar01 = 15.5;
    double myVar02 = 1.55;
    System.out.println(myVar01%myVar02);
  }//end doWork()
}//end class definition
```

A totally incorrect result

Unfortunately, due to floating arithmetic inaccuracy, the modulus operation in this program produces an entirely incorrect result. The result should be 0.0, and that is the result produced by my hand calculator.

Terminates one step too early

However, this program terminates the repetitive subtraction process one step too early and produces an incorrect remainder.

Be careful

This program is included here to emphasize the need to be very careful how you interpret the result of performing modulus operations on floating operands.

And the answer to the question is...

D. 1.5499999999999996

4.4.1 Not a Number

What output is produced by the program shown in Listing 10 (p. 27) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. NaN

Listing 4.10: Not a number (NaN).

```
    public class Modulus05{
  public static void main(String args[]){
    new Worker().doWork();
  }//end main()
}//end class definition

class Worker{
  public void doWork(){
    double myVar01 = 15.5;
    double myVar02 = 0.0;
    System.out.println(myVar01%myVar02);
  }//end doWork()
}//end class definition
```

Floating modulus operation involves floating divide

Because the modulus operation for floating operands involves a floating divide, you might expect the result to be *Infinity* when the right operand value is 0.0.

Not true!

The modulus operation with floating operands and 0.0 as the right operand produces *NaN* , which stands for *Not a Number* .

What is the actual value of *Not a Number* ?

A symbolic constant that is accessible as **Double.NaN** specifies the value that is returned in this case. Be careful what you try to do with it. It has some peculiar behavior of its own.

And the correct answer is...

D. NaN

4.5 Run the programs

I encourage you to run the programs that I have presented in this lesson to confirm that you get the same results. Copy the code for each program into your IDE. Then compile and run each program. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.6 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]

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

4.7 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Language Features, Arithmetic Operators
- File: Hs01006.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-

[9] http://apcentral.collegeboard.com/apc/Controller.jpf

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

A Advanced placement, § 1(1), § 2(3), § 3(5), § 4(15)
AP Computer Science, § 1(1), § 2(3), § 3(5), § 4(15)
Arithmetic Operators, § 1(1), § 2(3), § 3(5), § 4(15)

D data types, § 3(5)

H homeschool, § 1(1), § 2(3), § 3(5), § 4(15)

J Java subset, § 1(1), § 2(3), § 3(5), § 4(15)

L Language features, § 1(1), § 2(3), § 3(5), § 4(15)

P primitive types, § 3(5)

Attributions

Collection: *AP Computer Science A, Clarification of the Java Subset*
Edited by: Richard Baldwin
URL: http://cnx.org/content/col11279/1.4/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Language Features - Introduction"
By: Richard Baldwin
URL: http://cnx.org/content/m37133/1.3/
Pages: 1-2
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Language Features - Resources"
By: Richard Baldwin
URL: http://cnx.org/content/m37140/1.2/
Pages: 3-4
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Language Features, Primitive Types"
By: Richard Baldwin
URL: http://cnx.org/content/m37130/1.3/
Pages: 5-13
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Language Features, Arithmetic Operators"
By: Richard Baldwin
URL: http://cnx.org/content/m37146/1.2/
Pages: 15-28
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

AP Computer Science A, Clarification of the Java Subset
The purpose of the collection is to help you to better understand the Java Subset specifications for the Advanced Placement Computer Science A exam.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.