Chapter 2: Java Fundamentals

CS 2070

January 21, 2018

Review

Fundamentals

main

Standard Input and Output

Variables

Primitive Types

Variable Declaration

Arithmetic

Constants

Review

Review

- What is bytecode?
- What is just-in-time compiling?
- What is the name of the IDE that we are using in this class?
- What do the following acronym stand for: IDE, JVM, JDK, nd JRE.
- What is the file extension of Java source code?
- What is the file extension of Java class files?
- What's the name of the program that compiles Java programs?
- What's the name of the program that executes Java programs through the JVM?

Review

- What is bytecode?
  - It's Java's low-level programming language.
- What is just-in-time compiling?
  - It's the process which converts bytecode to machine code.
- What is the name of the IDE that we are using in this class?
  - Netbeans
- What do the following acronym stand for: IDE, JVM, JDK, and JRE.
  - Integrated Development Environment, Java Virtual Machine, Java Development Kit, and Java Runtime Environment

Review

- What is the file extension of Java source code? java
- What is the file extension of Java class files? class
- What's the name of the program that compiles Java programs? JDK or "javac"
- What's the name of the program that executes Java programs through the JVM? JRE or "java"

Fundamentals

Parts of a Java Program

- A Java source code file contains one or more Java classes.
- If more than one class is in a source code file, only one of them may be public.
- The public class and the filename of the source code file must match.
  - A class named Simple must be in a file named Simple.java
  - The IDE will take care of this for you.

Compiling and Executing a Java program

```
javac HelloWorld.java
java HelloWorld
```

- The first command "javac" will convert the source code into bytecode and generate the class file.
- The second command "java" will take the class file and produce the execution.
- Once again, the IDE takes care of this for us.

main

The main Method

```
public static void main(String[] args) {
    // Your Java program begins here.
    // If it behaves correctly, it should also end here.
}
```

The main Method: static and non-static

- Java doesn't have functions like C++.
- In Java, the equivalent to a function is a "method".
- All methods are associated with classes.
- They come in two flavors:
  - **static** methods are associated directly with classes.
  - **non-static** methods are associated with objects.
- Because there is no starting object in Java programs, the **main method must be static**.

main in detail

- public: This method is accessible to any method in the Java ecosystem.
- static: This method is associated with the class and not any one object.
- void: This method returns nothing.
- main: This is the name of the method.
- (String[] args): This method takes as input an array of String objects. This is used to pass data into your program from the start of execution.

C++ and Java: Similarities

- Single line comments and Multi-line comments are identical.
- Curly braces define scope of variables and program execution.
  - There are subtle differences in variable scope rules between the two languages.
  - These differences are not important if you remember to always name your variables appropriately.
- All statements require a semicolon at the end.

C++ and Java: Differences

- C++ encourages you to create header files and implementation files for your class and to use guards to prevent you from accidentally including the same file more than once.
- Java requires that you pack your class into a single file. Java is smart enough to know when a class has been imported more than once. There is no syntax for guards.
- C++ classes are statements, thus they require a semicolon at the end.
- Java classes are not statements, thus they do not require a semicolon at the end.

Standard Input and Output

Java API

- Java has many built-in classes for the programmer to use. This list of classes numbers in the thousands.
- This collection is called the **Java Applications Programming Interface**.
- Shorter, this collection is called the Java API.
- A few classes are automatically imported into every program and do not require import lines.
- Such classes include **System** and **Math**.

System

Inside the **System** class, there are two objects which are available to all programs:

- **in**: This is the standard input. It is typically the keyboard.
- **out**: This is the standard output. It is typically the monitor.

Printing output to the monitor

We use **System.out** to print things to the screen:

- System.out.println("Hello, world!");
  - Prints "Hello, world!" to the screen with a end-of-line character at the end so that the next content printed will be on a new line.
- System.out.print("Hello, world!");
  - Prints "Hello, world!" to the screen without an end-of-line character. The next content printed will appear on the same line.

What will this print?

Work this out on paper. If you are unsure, write a small Java program to verify your answer. Be sure to put this content inside the **main** method.

```
System.out.print("A ");
System.out.println("B ");
System.out.print("C ");
System.out.println("D ");
```

Java Escape Sequences

All Java Escape Sequences begin with a backslash character. “\”. You may use them inside of strings to change how the string behaves.

- n: newline
- t: tab
- b: Moves the cursor to the left one position. (By default, the cursor moves to the right after each charcter is displayed.)
- r: Moves the cursor to tbe beginning of the current line.
- \: Causes a backslash to be printed.
- ': Causes a single quote to be printed.
- ": Causes a double quote to be printed.

Java Escape Sequence Example

```
// Print tabs between apples, oranges, and grapes
System.out.println("apples\toranges\tgrapes");

// Print new lines between apples, oranges, and grapes
System.out.println("apples\noranges\ngrapes");
```

Variables

Variables

- In programming, variables hold data inside of a labeled container.
- A literal is data that appears directly in the source code of a program. It doesn't have a name.
- Examples of literals are 0, 1, 42, 3.14, false, or "dog".
- It's considered a best practice to assign literal numerical values to variables rather than using them in statements.
  - Seeing a number in code without being assigned to a variable is called a **magic number**. This is bad.
  - Exceptions to this rule are when an expression needs a 0 or a 1. This is okay.

Area of a Circle

The area of a circle with a radius of 2.5 units can be approximated with either equation.

$\pi radius^2$

-or-

3.14159×2.5^2

The difference between these two equations is that in the first, all of the parts are labeled, but in the second, you have to remember why each number is where it is. Be kind and don't use magic numbers.

Variable Definitions

- In Java (this is not true for every language) you must write a variable definition (also called a variable declaration).
- The Java compiler will provide an error if it encounters a variable in an expression which has not been defined.
- There are many different sizes and types of variables and we will cover these in more detail in this course.

Example of Variable output

```
int count = 5; // Example of Summation
int value = count + 1;
// Example of Concatenation
System.out.println("count is equal to "+count+"."); // 5
System.out.println("value is equal to "+value+"."); // 6
// Also prints 6
System.out.println("count+1 is equal to "+(count+1)+".");
```

The meaning of "+" operator in Java will change depending on the context. If both operands are numerical, then the two numbers will be added. If one or both of the operands is a String, Java will attempt to convert any non-String operands to a string and **concatenate** the String objects. If you wish to do addition inside of an output statement (not recommended), it must be wrapped in parentheses.

Rules for Naming Variables

- The first character must be on of the following:
  - upper or lower case letter
  - the underscore
  - the dollar sign
- Characters after the first can be any of the above and may include the digits 0 through 9.
- Variable names must not be reserved words.
- Variable names are case sensitive. “itemsOrdered” is different from “itemsordered”.

Soft Rules for Naming Variables

- "$$$", while legal, is a poor choice for a variable name.
- Begin your variables with a lower case letter.
- Begin your classes with an upper case letter (like String).
- Use nouns.
- Do not use jokes.
- Do not use abbreviations.
- Call things what they are.
- Your code should be self-documenting.

Primitive Types

Primitive Types

There are 8 primitive types in Java.

- Integer Types: byte, short, **int**, long
- Floating Point Types: float, **double**
- Boolean types: **boolean**
- Character types: **char**

Typically, you should use **int** for your integers and **double** for your floats.

Integer Types

There are four integer types.

- byte (8 bits) -128 to 127
- short (16 bits) -32768 to 32768.
- int (32 bits) -2.1B to 2.1B
- long (64 bits) -9e18 to 9e18

Most of the Java API uses int by default, so sticking with int won't cause a loss of precision.

Floating Point Types

- float (32 bits) 7 digits of accuracy
- double (64 bits) 15 digits of accuracy

To avoid loss of accuracy in floating point calculations, use the **double**.

Floating Point Types

By default, Java treats all floating point literals as **double** (thus you have a second reason to use **double**).

```
float number;
number = 3.14; // Error!
```

This will cause a compiler error because 3.14 is a **double** and you are assigning the value to **float**.

```
number = 3.14f; // Better.
```

Adding "f" or "F" to the end of a floating point number forces the number to be treated as a **float**. (This is identical in Java and C#.)

Boolean type

- The **boolean** type stores **true** or **false** and only **true** or **false**.
- Unlike C++, Java has a strict notion of true and false and are not related to numbers at all.
- You can only use boolean variables and expressions in contexts which require boolean expressions.
  - This is good because you will never accidentally use an integer in place of a condition.
  - The expression "if (x = 2) { ... }" will compile in C++ and give you an error in Java.

Character types

- The **char** type will store a single character.
- Like C++, we use single quotes for single characters and double quotes for strings.
- Unlike C++, the word "String" is always capitalized (because it is a class).
- Escape sequences use two characters yet count as one.

Code.

```
char firstLetter = 'A';
char newline = '\n';
String hello = "Hello, world!";
```

How big is a character?

- Currently Java uses Unicode UTF-16 encoding, which specifies that characters are 16 bits long.
- This allows for a **char** to maintain up to 65,536 different glyphs.
- The first 256 characters in the binary representation of the **char** are identical to the ASCII table.

Variable Declaration

Variable Declaration

Variable declaration works in Java the same as it does in C++.

- Variables must be declared.
- After they are declared, they must defined before they are used.
  - The defined value or expression must match the type of the variable.
  - Reading the value from a keyboard counts as defining a variable.
  - Note: You cannot define variables with undefined variables.
- The variable may then be printed to the screen.

What's wrong with the following code?

```
double pi = 3.14159;
double radius;
double area = pi * radius * radius;

System.out.println("The area is: "+area);
```

What's wrong with the following code?

```
double pi = 3.14159;
double radius;
double area = pi * radius * radius;

System.out.println("The area is: "+area);
```

The radius of the circle was never defined.

Variable Declaration: Take Note

- Variables only hold one value at a time.
  - Overwriting a variable with a new value is sometimes called "clobbering" (especially if you didn't mean to do it).
- Local variables are not assigned a default value and the compiler will complain if you forget this.
- Local variables must be assigned a valid type in order to be used.

Arithmetic

Arithmetic

- Java has your classic arithmetic operators.
  - + - * / %
- They also behave identical to how they behave in C++.
  - The division operator performs integer division when used with two integers.
  - 5 / 2 is 2
  - 11 / 4 is 2
  - 1 / 0 is going to be an error (either compiler if detected early and runtime if detected late)

Order of Operations

Here are the order of operations for classic arithmetic operators. Regardless of the order in which these operators are typed in an expression, they will be evaluated in this order.

- Anything inside of parentheses.
- unary positive and negative
- binary multiplication, division, and modulus
- binary addition and subtraction

Knowing this, you might be thinking that you can use parentheses to override anything. This thinking leads to lots of useless parentheses in code, which makes code harder to read.

Combined Operators

Like C++, Java has combined operators for most operators. These allow expressions to perform an assignment and another operation in the same expression.

- Addition Assignment (+=)
- Subtraction Assignment (-=)
- Multiplication Assignment (*=)
- Division Assignment (/=)
- Modulo Assignment (%=)

In all cases, the mathematical operator comes first, then the equal sign.

Constants

Why Constants?

- Constant variables allow us to create variables which we designate as unchangeable.
  - There are a maximum of 5 players on the field of a hockey team. Will this maximum ever change? Maybe, but not in the middle of a game.
- If we accidentally write an assignment statement to clobber an unchangeable variable, the compiler will generate an error. **This prevents us from making mistakes.**
- If we collectively decide in the future to change a constant variable, then there's a single point of change in our code. **This makes code easier to maintain.**
- We always know the value of a constant when we see it. **There is no mystery and this makes code easier to read.**

Why Constants?

- Constants are awesome.
- Some languages, like Rust, use constant variables by default and changeable variables with a special keyword (the opposite of C++ and Java).
- Some languages, like Haskell, require the programmer to use constant variables for everything. Nothing is allowed to be changed.

Constants in Java

- Here's the first major change from C++ to Java. The keyword for a constant in Java is **final**.
- By convention, all constants are created with ALL UPPERCASE LETTERS and have words separated using underscores.
- You can decide to declare a constant and assign a value to it later, but then it can never be reassigned again.

Code.

```
final double PI = 3.14159;
final double TN_SALES_TAX;

TN_SALES_TAX = 0.07;
```

Strings in Java

Strings in Java

Here's the second major change from C++. Study the following C++ code.

```
string message;
```

This (in C++) will call the string class' default constructor and create an empty string. Now study this code in Java.

```
String message;
```

This creates a String variable called message, but it's uninitialized. **Java will not call default constructors by default.**

Strings in Java

We could create a new String variable a few ways.

```
String message = new String("Hello!");
```

Or this.

```
String message = "";
```

Reminder: all classes in Java begin with an uppercase letter.

Strings in Java

Primitive variables in Java will hold a value of a variable. Variables based on classes do not.

```
int number = 25;
String message = "Hello!";
```

In this example, **number** is a variable which holds 25, but **message** is a placeholder reference that points to another memory location somewhere in memory that contains the characters “Hello!” In this example, **message** is a **reference**.

Does Java have pointers?

When I said, "Java doesn't have pointers!" that was partially true.

- Java has something similar to pointers called **references**. They behave the same as C++'s pointers.
- All object variables are **references** and should contain the address in memory which has the true value of the object.
- References can also be null. This is like holding a ticking time bomb.
- Dereferencing a reference which is null will trigger a **NullReferenceExceception** and your program will crash.
- Java doesn't have pointer syntax to memorize. At least there's that.

Jeez. This is all confusing.

I know what you are thinking: How can I quickly determine what is a primitive and what is an object reference?

- If it's a byte, short, int, long, float, double, char, or boolean, then it's a primitive variable.
  - You need to memorize these eight types as the collection of primitive types.
- Everything else is a reference variable and behaves like a C++ pointer.

Back to Strings

We can explore features of the String class by reading the documentation for String on the Oracle website. Here's a few of those methods.

```
String message = "Hello!";
System.out.println(message); // Prints "Hello!"
System.out.println(message.length()); // Prints "6"
System.out.println(message.charAt(0)); // Prints "H"
System.out.println(message.charAt(5)); // Prints "!"
```

Take Note: Strings

- There are many methods in the String class, but none of them will change the String object.
- Strings are immutable. You cannot directly modify a String object.
- You can call a method which will return a new String object and assign that back to the object which you are using.

Take Note: Strings

For example, here's how we can make a string all upper case.

```
String message = "Hello!";
System.out.println(message); // Prints "Hello!"
message.toUpperCase();
System.out.println(message); // Prints "Hello!" No change.
message = message.toUpperCase();
System.out.println(message); // Prints "HELLO!"
```

The first call to method "toUpperCase()" generates a completely new String object, but because String objects are immutable, this does nothing. In the second call to "toUpperCase()", we also use the assignment statement to clobber the existing String object. We didn't directly change the original memory. We overwrote the existing reference.

Input and Output

Input and Output

All programs from Microsoft Word to Facebook to Overwatch follow this general approach to processing data:

1. Get some data from an input device (such as the keyboard) or from a file.
2. Process that data using a combination of algorithm and secondary storage files.
3. Send the processed result to an output device (such as the monitor) or write the results to a file.

Reading from the Keyboard

- As we mentioned earlier, **System.in** points to your computer's keyboard.
- We still have to create an object which uses the keyboard and this object is called **Scanner**.

Code.

```
Scanner keyboard = new Scanner(System.in);
```

This will produce a compiler error because Java doesn't automatically import the Scanner class.

Importing the Scanner class

- The Scanner class is found in the **java.util.Scanner** class library.
  - Import statements are similar to include statements in C++.
  - We must put the statement "import java.util.Scanner;" near the top of our code.
- NetBeans has a tool called "Fix Imports" which will automatically write this line for you.
  - On Windows, this is "Ctrl+Shift+I".
  - On Mac, this is "Cmd+Shift+I".
- I will be recording a video on how to use this feature.

Using the Scanner class.

Once we have the class imported, we can use it. Here's an example.

```
System.out.println("This program reports your age in five y
Scanner keyboard = new Scanner(System.in);
System.out.print("What is your age? ");
int age = keyboard.nextInt();
int agePlusFive = age + 5;
System.out.println("In five years, you will be " +
    agePlusFive + " years old");
```

Email me your questions about this lecture.

churchj@apsu.edu