CSE 1325 - Object-Oriented Programming

Introduction to Java

Alex Dillhoff

University of Texas at Arlington

Analyzing `FirstProgram`

We begin the journey into Java by looking at the simple program written in the introduction.

Analyzing FirstProgram

```java
public class FirstProgram {
    public static void main(String[] args) {
        String msg = "Welcome to CSE1325!";
        System.out.println(msg);

        for (int i = 0; i < msg.length(); i++) {
            System.out.print("=");
        }

        System.out.println();
    }
}
```

Analyzing FirstProgram

```
public class FirstProgram {
```

This first line declares a **class**.

Everything in Java is defined within a class. This idea allows developers to encapsulate variables and methods belonging to a particular idea.

Analyzing FirstProgram

```
public class FirstProgram {
```

The keyword `public` is an **access modifier**.

Access modifiers control which parts of a program are accessible and usable from other parts.

Without the context of a larger program, this doesn't have much meaning.

Analyzing FirstProgram

```
public static void main(String[] args) {
```

The second line may look familiar coming from C.

It is declaring the main function and accepts an array of arguments.

Technically, this is called a **method** in Java.

Analyzing FirstProgram

```
public static void main(String[] args) {
```

The `static` keyword indicates that the method can be accessed directly, without needing to instantiate an object of the class `FirstProgram`.

Analyzing `FirstProgram`

```
public static void main(String[] args) {
```

The argument `String[] args` indicates that an array of command line arguments are passed to the method.

Since `args` is an object of the `String` class, we can utilize all of the methods associated with it (more on that later).

Analyzing `FirstProgram`

```
String msg = "Welcome to CSE1325!";
```

This line instantiates an **object** of class `String` and initializes the value.

Any of the member methods defined in the `String` class can be used (i.e. `msg.length()`).

Analyzing `FirstProgram`

```
System.out.println(msg);
```

`System.out` is Java's interface to the standard output stream.

The `println` class method prints a message, including a newline character at the end.

Analyzing FirstProgram

```java
for (int i = 0; i < msg.length(); i++) {
    System.out.print("=");
}
```

Loop syntax is similar to C/C++. The two things of note in this block are `msg.length()` and `System.out.print()`.

Analyzing FirstProgram

```
for (int i = 0; i < msg.length(); i++) {
    System.out.print("=");
}
```

`msg.length()` calls the `length()` method defined in the `String` class. This returns the length of the string.

Analyzing `FirstProgram`

```
for (int i = 0; i < msg.length(); i++) {
    System.out.print("=");
}
```

`System.out.print()` prints a message to the standard output stream newline character.

Comments in Java

The syntax for commenting in Java is similar to that of C/C++.

```
// This is a single-line comment

/*
 * This is a
 * multi-line
 * comment
 */
```

Comments in Java

There is a third type of comment which is used for automatic documentation generation.

```
/**
 * A simple program.
 * @version 1.0
 * @author Alex Dillhoff
 */
 ...
```

Comments in Java

We will cover documentation generation in greater detail later in the semester.

Data Types

Just like C and C++, Java is a **strongly typed language**.

That is, every variable is declared with a type.

Data Types

There are 8 **primitive** types in Java:

- 4 integer types
- 2 floating-point types
- the character type
- the `boolean` type

Unlike C/C++, the sizes for the types are not implementation specific. They are always the same, no matter what machine you run the code on.

Integer Types

The 4 integer types are

1. `int` (4 bytes)
2. `short` (2 bytes)
3. `long` (8 bytes)
4. `byte` (1 byte)

Integer Types

Certain integer numbers can be indicated by using a suffix or prefix.

For example, a `long` integer can be indicated as `1000L`.

Hexadecimal values are specific by the prefix `0x` or `0X`.

Integer Types

Binary numbers can be written with the previx `0b` or `0B`.

For readability, underscores can be added to **all** numbers. For example,

```
0b1001_1010;   // 154 in binary
32_123_456;    // 32,123,456
```

Floating-Point Types

The 2 floating-point types are

1. `float` (4 bytes)
2. `double` (8 bytes)

If not specified with the suffix `f` or `F`, a floating point number will be considered a `double` (e.g. `1.28`).

The `char` Type

Java uses a unicode character set. This permits the use of characters in languages with much larger typesets.

These characters can be written by their unicode values.

Reference: `https://www.rapidtables.com/code/text/unicode-characters.html`

The char Type

Using unicode characters in a string is similar to how it was done with ASCII and C.

```
String msg = "End with a newline.\u000A";
```

The `char` Type

Using the `char` type explicitly should be avoided unless absolutely necessary.

The `String` class provides many useful methods and operators for strings and characters.

The `boolean` Type

`boolean` values are the **only** ones usable for logical statements.

In C/C++, a nonzero value is interpreted as **true** and a zero value is interpreted as **false**.

The motivation behind this restriction in Java is that it is more secure. A statement such as `if (x = 0) ...` would not compile.

Variables

Variables in Java largely follow the same rules and syntax as those in C.

Traditionally, all variables must be declared with a type. Starting with Java 10, it is possible to declare a variable whose type is inferred by the initialization.

```
var dist = 3.24; // a double
var msg = "Wait."; // a String
```

Constants

Constants are variables whose values cannot be modified after initialization.

The `final` modifier is used to define a constant.

This will ensure that the variable cannot be modified within the method it is defined.

Constants

To create a constant that can be used through an entire class, define a variable with the modifiers `static final`.

Constants

Example: Local constant

```
public static void main(String[] args) {
    // useable within the scope of main
    final double pi = 3.14;
}
```

Constants

Example: Class constant

```
public class FirstProgram {
    // useable through out the class
    private static final double pi = 3.14;
}
```

Enumerated Types

Enumerated types are useful in defining a restrictive set of values for a particular property.

For example, you may want to encode that a stop light can only take on the colors red, yellow, or green.

```
enum LightColors {
    RED,
    YELLOW,
    GREEN
};
```

Enumerated Types

Using these in your code is as simple as calling the `enum` name followed by the particular value.

```
LightColors color = LightColors.GREEN;

if (color == LightColors.GREEN) { ... }
```

Operators

The operators available in C/C++ are also available in Java and using them will be very familiar.

Operator precedence will also be similar as well. The exact operator precedence can be reviewed at `https://introcs.cs.princeton.edu/java/11precedence/`

Operators

There is a difference between integer and floating-point division when dividing by 0.

Integer division by 0 will raise an **exception** (more on that later).

Floating-point division by 0 will yield an infinite value or `NaN` (Not a Number).

Operators

Example: `OperatorExample.java`

A Math Library

Any useful language will provide a library with common math functions and constants. Java is no exception.

There is no need to import any package to use this library.

Reference
`https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Math.html`

A Math Library

Example: `MathExample.java`

Numeric Conversions

In general, no information is lost when converting to a primitive type with higher precision.

For example, a `byte` can be converted to a `short` without loss of precision.

However, converting a `long` to `float` would lose precision.

Strings

For those new to Java, `Strings` will be their first experience with the convenience of object-oriented design.

However, there are a few key points to remember when working with `Strings`.

Strings

A `String` can be declared and initialized just as you would any other variable.

```
String str1 = "Hello";
```

This string it **immutable**, meaning it cannot be changed!

Strings

Coming from another language in which strings are treated as character arrays, this can seem bizarre.

There is a benefit to this design: **efficiency**.

In Java, `Strings` are stored in a shared location.

Strings

Consider the following statements.

```
String str1 = "CSE1325";
String str2 = "CSE1325";
```

In the shared space, there is only one literal string `"CSE1325"`.

Strings

The motivation behind this design decision is that it is more common to compare strings than it is to build them.

To remedy this immutable property, Java provides mutable classes for building strings. We will explore these later.

Strings

Review the following example to show some commonly used `String` methods.

Example: `StringExample.java`

Strings

See the Java API documentation on `Strings` for a complete list of methods.

Reference `https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/String.html`

Input and Output

Reading and writing input to the shell is done by interfacing with the input, output, and error streams.

We have already seen examples of printing to `stdout` via `System.out.print`, `System.out.println`, and `System.out.printf`.

Reading Input

Let's start by looking at how Java handles user input from the shell.

This is facilitated with the `Scanner` class.

Start by constructing a `Scanner` object.

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

Reading Input

We can read multiple types of data by utilizing the methods provided by `Scanner`.

Example: `ScannerExample.java`

Formatting Output

Output can be formatted similar to how it is done in C/C++ using `printf`.

Aside from `System.out.printf`, Java offers the `Formatter` class.

Reference
`https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Formatter.html`

File I/O

The `Scanner` class can be used to read information from files.

To write to a file, the Java API provides the `PrintWriter` class.

Example: `FileExample.java`

Arrays in Java

Just like other languages, arrays can be made for any type. This includes user-defined class objects.

Declaring an array is as easy as

```
double a[]; // familiar syntax
double[] b; // allowable in Java
```

The above are merely declarations, they have no storage.

Arrays in Java

To define an array, use the `new` operator.

```
double[] arr = new double[100];
```

Arrays in Java

The number of elements can be dynamic. For example:

```
int n = 100;
double[] arr = new double[n];
```

Arrays in Java

See the following example demonstrating the basics of arrays.

Example: `ArrayExample.java`

Sorting Arrays

Sorting an array in Java is done easily using a static method from the `Arrays` class.

```
int[] arr = {10, 5, 3, -5, 7, 0, -3};
Arrays.sort(arr);
```

Sorting is done via an optimized version of QuickSort.

Sorting Arrays

Example: `ArraySortExample.java`

Sorting Arrays

Much more control can be had when sorting arrays.

This is especially useful when custom objects are used and it is not clear what property to use to sort arrays.

We will look at this in more detail later on.

Arrays in Java

Reference

`https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Arrays.html`

Multidimensional Arrays

Using multidimensional arrays in Java will be familiar for those coming from C.

They are declared by adding a set of brackets for each desired dimension:

```
// 2D array
int[][] arr2d;
// 3D array
int[][][] arr3d;
```

Multidimensional Arrays

Again, Arrays in general can be defined using variables.

```
double[][][] img = new double[channels][height][width];
```

Ragged Arrays

Multidimensional arrays in Java are not contiguous as they are in C/C++.

They behave more like an array of pointers would in C.

For example, the rows of a 2D array in Java each ”point” to another 1D array.

Ragged Arrays

This also means that you can construct multidimensional arrays whose rows contain arrays of different length.

```
int[][] arr = new int[n][];
arr[0] = new int[20];
arr[1] = new int[30];
...
```

Multidimensional Arrays

Review the provided example for more common usages.

Example: `MultiDimensionalArrayExample.java`

Methods (Functions)

Java is an object-oriented programming language, which means that solutions are implemented as `class`es.

Functions as we knew them in C/C++ are called **methods**.

These methods describe the behaviors and actions of a `class`.

Methods (Functions)

The syntax for creating a custom method will be familiar coming from C.

Additional method modifiers will be explored when we discuss object and classes in detail.

Methods (Functions)

Just like in other langues, a method (function) is defined by a return type and a parameter list.

Of course, it is not necessary to have any parameters if your method does not need it.

Methods (Functions)

Consider the following method

```
public int addNumbers(int a, int b) {
    return a + b;
}
```

The access modifier `public` is not important for now. We will look at how access modifiers are used in the next lecture.

Methods (Functions)

Just like in C, the return type `int` specifies that the method will return an `int`.

Also familiar, the parameter list specifies the number and type of each argument that is passed to the method when it is called.

Methods (Functions)

Calling the method within the class it is defined requires no special syntax. For example:

```
int result = addNumbers(20, 30);
```

Methods (Functions)

Like C, Java uses **call by value** for argument passing.

That means that the object's value is copied when passed to a method.

What does this imply for modifying variables inside of a method?

Methods (Functions)

For primitive types like `double`, it means that a change inside a method does not persist outside of that scope.

```java
int x = 0;
addTwo(x); // adds 2 to `x`
System.out.print(x); // `x` is still 0
```

Methods (Functions)

What about **objects**? Does this mean we cannot change object properties within a method?

Methods (Functions)

What about **objects**? Does this mean we cannot change object properties within a method?

Object instances have a reference to the object. When they are passed, the reference is copied.

Methods (Functions)

This is not **pass-by-reference** as seen in C++. This is similar to the address value being copied for a pointer in C.

Example: `ObjectReferenceExample.java`