CS101, Mock
Syntax, Program Structure, Data Types

In this section, we'll look in more detail about the format, structure, and functions involved in creating Java programs.

General Format of a Simple Java Program

In general, your programs will have the following structure:

```
import …

class ClassName {
          public variable declarations;
          private variable declarations;

          public method declarations;
          private method declarations;
}
```

The import statement at the top tells the Java compiler what other pre-compiled packages you want to use. Rather than rewrite commonly used procedures from scratch, you can use pre-built **packages**. The import statement specifies which packages you wish to use. Inside the class, you will declare variables that belong to that class. We'll focus on two categories of variables, public and private, where public is information we'll make available to the rest of the program, and private is only available within the class. Then we'll define methods that make up the class.

Here is a list of the terms that you will commonly encounter:

- **Program** : A general term describing a set of one or more Java classes that can be compiled and executed
- **Variable** : This is a name for a value we can work on. This value will be stored somewhere in memory, or perhaps in a register. Variables can refer to simple values like numbers, or to entire objects.
- **Method** : This is a function. It groups together statements of code to perform some type of task.
-
- **Class** : This describes the object that contains variables and methods.
- **Object** : This is used interchangeably with class. More specifically, an object is an actual instance of an object created by the new statement.
- **Identifier** : This is the name of an entity in Java, which could be a class, variable, method, etc.

- **Keyword** : This is a reserved word with special meaning to Java and can't be used as an identifier. For example, the word "class"
- **Statement** : This is a single line of code that does a particular task. A statement must be terminated by a semicolon.
- **Parameters** : These are values that are passed to a method that the method will operate on.

So far we've written our first program that could output a line of text. Let's expand from the first program and first learn more output options.

To output text, we used "System.out.println" and the message we wanted within quotation marks. But what if we wanted to print out a double quotation mark? The computer would get confused because it would think the quotation mark you want to print really ends the message to be printed:

```
	class OutputTest {
		public static void main(String[] args) {
			System.out.println("A famous politician once said,
"If we do not succeed, then we run the risk of failure."  ");
		}
	}
```

What will happen when this program is compiled? The compiler will complain about missing some expected character. Text enclosed in double quotes is referred to as **strings**. If we want to print a double quote itself (or some other special characters) then we need to use an **escape character**. In Java, the escape character is \. The program that works is:

```
	class OutputTest {
		public static void main(String[] args) {
			System.out.println("A famous politician once said,
\"If we do not succeed, then we run the risk of failure.\"  ");
		}
	}
```

Some other escape characters:

\n	- newline, Moves the cursor to the next line like endl
\t	- horizontal tab
\r	- carriage return, but does not advance the line
\\	- print the escape character itself

Quiz : Output of the following?

```
	class OutputTest {
		public static void main(String[] args) {
			System.out.println("He said, \n\"Who's responsible
for the riots? \tThe rioters.\"");
		}
	}
```

Let's expand our program a little bit more to include some **identifiers**. An identifier is made up of letters, numbers, and underscores, but must begin with a letter or an underscore.

Beware: Java is case sensitive. This means that Value, VALUE, value, and vaLue are four separate identifiers. In fact, we can construct 32 distinct identifiers from these five letters by varying the capitalization. Which of these identifiers are valid?

_FoosBall   F00sBall   %FoosBall   9FoosBall%   12391   _*_   _FF99

Note: When picking identifiers try to select meaningful names!
Here is a short program that uses some variables as identifiers:

```
class OutputTest {
        public static void main(String[] args) {
                char period = '.';                              // Single quotes
                String name = "Cotty, Manny";                   // Double quotes
                String foods = "cheese and pasta";
                int someNum = 0xF;                              // 0x indicates hex

                System.out.println(name + " loves to eat " + foods);
                System.out.println(someNum + " times as much as you" + period);
        }
}
```

Let's walk through the program:

Lines 3 through 6 instruct the compiler to assign variables of a particular type. The format is to first indicate the data type identifier, in this case **char**, **String**, or **int**.

- char indicates that the value to be stored is to hold a single ASCII character.
- String (note the uppercase S) indicates that the value to be stored can be composed of many characters. Note that if we want to modify the contents of a string, there is another type called StringBuffer that lets us do this efficiently. We can change Strings to something else, but this ends up making a new copy of the String instead of changing the original String.
- int indicates that we want to store an integer value, e.g. using the 2's complement representation.

In line 3, a period is stored in the variable named *period*. In line 4, the string made up of the characters 'C', 'o', 't', 't', 'y', ',' , ' ', 'M', 'a', 'n', 'n', 'y' is stored in *name*. By convention, most Java programmers use all uppercase for identifiers that won't change,

and lowercase mixed with uppercase for ones that may change. Since these strings won't change, we could have named this identifier NAME instead of name.
In line 6, we defined one numeric value, *someNum* to 0xF. This sets *someNum* to fifteen. We have a number of ways of defining numbers:

Decimal: Use the normal number, e.g. 9	= decimal 9
Octal: Use a leading 0, e.g. 011	= decimal 9
Hex: Use a leading 0x, e.g. 0xF	= decimal 15

When the compiler processes the variable declarations, it assigns a memory location to each one. It is intended that the data we store in each memory location is of the same type that we defined it in the program. For example, we defined someNum to be of type int. This means we should only store integers into the memory location allocated for someNum. We shouldn't be storing floating point numbers, strings, or characters.

Also note that it is customary to define all variables used within a function at the top of the function (in this case, the function is main).

Finally, we have output statements that print the contents of the variables. Note that when we use println, we can print out variables of different types and **concatenate** the output using the + symbol. The + symbol will also serve as addition as we will see later! So be aware that a symbol may do different things in a different context.

The final output when run is:

```
Cotty, Manny loves to eat cheese and pasta
15 times as much as you.
```

Words and Symbols with Special Meanings

Certain words have predefined meanings within the Java language; these are called *reserved words* or *keywords*. For example, the names of data types are reserved words. In the sample program there are reserved words: **char**, **int**, **void, main**. We aren't allowed to reserved words as names for your identifiers.

Data Types

A data type is a set of values and a set of operations on these values. In the preceding program, we used the data type identifiers **int**, **char**, and **String**.

In Java there are four integral types that can be used to refer to an integer value (whole numbers with no fractional parts). These types are **byte**, **short**, **int**, and **long** and are intended to represent integers of different sizes. These are just integers stored using the 2's complement format that by now you should know and love. The set of values for each of these integral data types is the range of numbers from the smallest value that can be represented through the largest value that can be represented. The operations on these values are the standard arithmetic operations allowed on integer values.

the sizes for the integer types are:

> **byte** - one byte, holds a number from –128 to 127
> **short** – two bytes, holds numbers up to 32767
> **int** – four bytes, holds numbers from –2,147,483,648 to 2,147,483,647 (2^{31}),
> **long** – eight bytes , holds numbers up to around 10^{18}

You might always be tempted to use type long for all numbers because it can hold the largest range. While this is possible, it would not be efficient if the values your program is processing are all small. In this case, you'll be wasting lots of bits in allocating eight bytes of space when you might really only be using one byte.

Data type **char**, which can also be used to store bytes, has a primary use for describing one alphanumeric character. Although arithmetic operations are defined on alphanumeric characters because they are type **char** (also an integral type), such operations would not make any sense to us at this point. However, there is a collating sequence defined on each character set, so we can ask if one character comes before another character (the ASCII code). Fortunately, the uppercase letters, the lowercase letters, and the digits are in order in all character sets. The relationship between these groups varies, however. We discuss manipulating **char** data in more detail later.

We used two variables of data type **String**, *name* and *foods*. String is actually not an integral data type; it is an object. We'll say more about the difference later. For now, you can use String to hold a sequence of characters. Note that string data is denoted with double quotes, not a single quote as for char. "I" refers to the string with the letter I in it, while 'I' refers to the character with the letter I. The two are different, they are different types and are stored in a dramatically different way!

Finally, there are separate data types for fractional or floating point numbers. These are numbers that are stored using the IEEE 754 format we discussed previously in class.
There are two types of floating point storage:

> float - 4 bytes, using IEEE 754. Values up to 10^{38} possible
> double – 8 bytes, using IEEE 754. Values up to 10^{308} possible

Arithmetic Expressions

Variables and constants of integral and floating point types can be combined into expressions using arithmetic operators. The operations between constants or variables of these types are addition (**+**), subtraction (**-**), multiplication (*****), and division (**/**). If the operands of the division operation are integral, the result is the integral quotient. If the operands are floating point types, the result is a floating point type with the division carried out to as many decimal places as the type allows. There is an additional operator for integral types, the modulus operator (**%**). This operator returns the remainder from integer division.

In addition to the standard arithmetic operators, Java provides an *increment* operator and a *decrement* operator. The increment operator **++** adds one to its operand; the decrement operator **--** subtracts one from its operand.

Examples:

```
class OutputTest {
      public static void main(String[] args) {
            int x=1;

            x = x + 55;
            System.out.println(x);
      }
}
```

Same with:

x = 5 * 10 * 2;
x = 14 % 5;
x = 10 /2 ;
x++;
x--;
x = 11 / 2;
x = 1 / 2;
x = 100000000 * 100000000;   (may get overflow warning)

Note : Truncation, not rounded to nearest integer

Precedence Rules

The precedence rules of arithmetic apply to arithmetic expressions in a program. That is, the order of execution of an expression that contains more than one operation is determined by the precedence rules of arithmetic. These rules state that:

1. parentheses have the highest precedence
2. multiplication, division, and modulus have the next highest precedence
3. addition and subtraction have the lowest precedence.

Because parentheses have the highest precedence, they can be used to change the order in which operations are executed. When operators have the same precedence, order is left to right.

Examples:

x = 1 + 2 + 3 / 6; output?
x = (1 + 2 + 3) / 6;
x = 2*3 + 4 * 5;
x = 2 / 4 * 4 / 2;
x = 4 / 2 * 2 / 4;
x = 10 % 2 + 1;

Converting Numeric Types

If an integral and a floating point variable or constant are mixed in an operation, the integral value is changed temporarily to its equivalent floating point representation before the operation is executed. This automatic conversion of an integral value to a floating point value is called *type coercion*. Type coercion also occurs when a floating point value is assigned to an integral variable. Coercion from an integer to a floating point is exact. Although the two values are represented differently in memory, both representations are exact. However, when a floating point value is coerced into an integral value, loss of information occurs unless the floating point value is a whole number. That is, 1.0 can be coerced into 1, but what about 1.5? Is it coerced into 1 or 2? In Java when a floating point value is coerced into an integral value, the floating point value is **truncated**. Thus, the floating point value 1.5 is coerced into 1.

Type changes can be made explicit by placing the new type in parentheses in front of the value:

intValue = int(10.66);

produce the value 10 in intValue

Here are some same typecasts:

(double) intValue; (int) doubleValue;
(long) intValue; (int) longValue;
(long) floatValue;

Examples with float:

```
class OutputTest {
      public static void main(String[] args) {
            float x=1;

            x = 11 / 2;
            System.out.println(x);
      }
}
```

You might think this would produce 5.5. But instead it produces 5.0. Why?

How about the following:

```
x = (float) 11 / 2;          // Since 11 is a float, 2 is also turned into a float
x = 11 / (float) 2;          // Similar to above
x =  2 / 4 * 4 / 2;
x = 2 / 4.0 * 4 / 2;         // 4.0 treated as a double
                             // Compiler may complain about double to float
```

The bottom line here is to be careful if you are mixing integers with floating point values in arithmetic expressions. Especially if performing division, you might end up with zero when you really want a floating point fractional answer. The solution is to coerce one of the integers into a float or double so the entire calculation is made using floating point.

Let's put together what we know so far with an example program. Here is the problem:

You are running a marathon (26.2 miles) and would like to know what your finishing time will be if you run a particular pace. Most runners calculate pace in terms of minutes per mile. So for example, let's say you can run at 7 minutes and 30 seconds per mile. Write a program that calculates the finishing time and outputs the answer in hours, minutes, and seconds.

Input:

```
Distance : 26.2
PaceMinutes:  7
PaceSeconds: 30
```

Output:

```
3 hours, 16 minutes, 30 seconds
```

Here is one algorithm to solve this problem:

1. Express pace in terms of seconds per mile, call this SecsPerMile
2. Multiply SecsPerMile * 26.2 to get the total number of seconds to finish. Call this result TotalSeconds.
3. There are 60 seconds per minute and 60 minutes per hour, for a total of 60*60 = 3600 seconds per hour. If we divide TotalSeconds by 3600 and throw away the remainder, this is how many hours it takes to finish.
4. TotalSeconds mod 3600 gives us the number of seconds leftover after the hours have been accounted for. If we divide this value by 60, it gives us the number of minutes.
5. TotalSeconds mod 3600 gives us the number of seconds leftover after the hours have been accounted for. If we mod this value by 60, it gives us the number of seconds leftover. (We could also divide by 60, but that doesn't change the result).
6. Output the values we calculated!

Code:

```
class RacePace {
      public static void main(String[] args) {
        double distance = 26.2;
        int paceMinutes = 7;
        int paceSeconds = 30;

        long secsPerMile, totalSeconds;

        secsPerMile = (paceMinutes * 60) + paceSeconds;
        totalSeconds = (long) (distance * secsPerMile);
        System.out.print("You will finish in: ");
        System.out.print(totalSeconds / 3600 + " hours, ");
        System.out.print((totalSeconds % 3600) / 60 + " minutes, ");
        System.out.println((totalSeconds % 3600) % 60 + " seconds.");
      }
}
```

A few things to note about this program:

```
totalSeconds = (long) (distance * secsPerMile);
```

Since distance is a double and secsPerMile is an int, the whole thing is typecast to a long, which is the type of totalSeconds.

System.out.print is the same as System.out.println, except System.out.print does not add a newline to the end of the output. This means that everything gets printed onto the same line until the very final statement, which prints a newline with the println statement.

The output is;

```
You will finish in: 3 hours, 16 minutes, 30 seconds.
```

If we wanted to calculate the finish time for different distances and different paces, we'll need to change the values in the program and recompile it. This can be inconvenient – it would be nice to have the user input any values he or she desires. In the next few lectures we'll see how to input values from the user using the book's input package and also create some graphical windows to display data.