CSCE A201
Syntax, Program Structure, Data Types, Variables, Arithmetic Operations, Input

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 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 a class 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.

- **Arguments** : These are values that are passed to a method that the method will operate on. The variables holding the arguments in the method are **parameters**.

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:

```
      public 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:

```
      public class OutputTest {
            public static void main(String[] args) {
                  System.out.println("Dan Quayle 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
\' - single quote

Quiz : Output of the following?

```
      public 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!

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 use 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. The table below lists the primitive data types built into Java.

Type Name	Kind of Value	Memory Used	Range of Values
`byte`	Integer	1 byte	–128 to 127
`short`	Integer	2 bytes	–32,768 to 32,767
`int`	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
`long`	Integer	8 bytes	–9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
`float`	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
`double`	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
`char`	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
`boolean`		1 bit	True or false

To define a variable we use the name of the data type followed by an identifier. For example, to make the variable **numStudents** that can store an integer we would use:

```
int numStudents;
```

Note that the variable **numStudents** can ONLY store integers. It can't store a floating point number like 1.5. If we needed to store a floating point number we'd have to define it using one of the floating point types, such as:

```
double numStudents;
```

Optionally (but recommended), we can initialize variables to values. For example, to declare the variable numStudents to be 10 we can write:

```
int numStudents = 10;
```

Note that you only need to declare a variable once; thereafter you can use it over again within the variable's scope. We'll discuss later but for now you only need to make a variable once.

There is another data type, **String**, that we will also use. This data type can store sequences of characters. It's not a primitive data type (we'll see the distinction later) but it is built into Java. Here is an example of a string named food that stores the value Pizza:

```
String food = "Pizza";
```

Note the uppercase S in String.

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:

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

		x = x + 55;
		System.out.println("The value of x is " + 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

Also note that the "+" in the System.out.println doesn't add anything together, but is used to **concatenate** or append the value of x to the string "The value of x is".

To complicate matters a bit more, if ++ and -- are used within an expression, we get different results. ++x is a **pre-increment** while x++ is a **post-increment** operation. In a pre-increment operation, the variable is incremented before evaluated. In a post-increment operation, the variable is not incremented until after evaluation. Consider the following example:

```
public class OutputTest
{
	public static void main(String[] args)
	{
		int x=1;
		int y=1;
		x = (y++)*2;
		System.out.println(x + " " + y);
		x = 1;
		y = 1;
		x = (++y)*2;
		System.out.println(x + " " + y);
	}
}
```

The output from this program is:

2 2
4 2

In the post-increment evaluation of (y++)*2, we assign the current value of y multiplied by 2 into x (which is 1*2 = 2) and then increment y to 2.

In the pre-increment evaluation of (++y)*2, we first increment y to 2, then compute 2*2=4 and assign that into x.

Shorthand Assignment Operators

A useful shorthand in Java for assignment statements is to combine the assignment along with a mathematical operator. In general, we can use a math symbol in combination with an = to apply the operation to the same variable on the left, and store the result back into that variable.

The following are equivalent:

x = x+ y	←→	x += y
x = x* y	←→	x *= y
x = x - y	←→	x -= y
x = x/ y	←→	x /= y
x = x% y	←→	x %= y

Generally these are used with constants; e.g.:

```
x += 5;
```

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 sample 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.

Example:

To maintain one's body weight, a human that is *A* years old, weighs *K* kilograms and is *H* centimers tall needs to consume approximately the following number of Calories per day:

Males: $10\times K + 6.25\times H - 5\times A + 5$
Females: $10\times K + 6.25\times H - 5\times A - 161$

(One Calorie is also referred to as one kilo-calorie or kcal. A kilocalorie is what food manufacturers refer to as a single "Calorie" on food labels.) This formula is derived from the Mifflin-St Jeor equations and is an approximation based on a person that engages in no physical activity - i.e. the person basically stays in bed all day.

One package of twix (i.e. two cookie bars, because nobody eats just one) is 568 Calories according to the M&M/Mars website. If you go on the twix-only diet all day long, then the number of twix packages you would need to eat is TotalCaloriesNeeded / 568.

Write a program that has variables for:

- A person's age in years
- A person's weight in pounds
- A person's height in feet and inches (two variables)

Thinking about the algorithm, our program has to convert from the English units of pounds, feet, and inches, into the metric units of kilograms and centimeters.

So we should make additional variables for:

- The person's weight in kilograms. One kilogram is 2.2 pounds.
- The person's height in centimeters. Compute this by converting the person's height in inches (12 inches per foot) and then from inches to centimeters (2.54 centimeters per inch).

Here is our pseudocode:

```
Set age, weight, feet, inches to the values we are interested in
kilograms = weight / 2.2
centimeters = ((feet * 12) + inches) * 2.54
calories needed = 10 * kilograms + 6.25 * centimeters – 5*age + 5 (for males)
twix bars to eat = calories needed / 568
```

Here is working code:

```
public class TwixBars
{
      public static void main(String[] args)
      {
            int age = 60;
            int weight = 210;
            int feet = 5;
            int inches = 6;

            double kilograms = weight / 2.2;
            double centimeters = ((feet * 12) + inches) * 2.54;

            double calories =  10 * kilograms + 6.25 *
                               centimeters - 5 * age + 5;
            double numTwix = calories / 568;

            System.out.println("You need to eat " + numTwix +
                  " twix bars per day to maintain your weight.");
      }
}
```

Named Constants

In the prior example, let's say that the program is only going to be used for twix bars. In this case, we might always divide by 568. That number by itself is kind of mysterious because it's not obvious that is the number of calories in a twix bar. Assuming this value is never going to change, it is a good programming practice to define the value as a **constant** rather than as a number or variable.

As the name implies, something defined as a constant may never be changed. To create a constant, use the following syntax outside the method, but inside the class:

public static final NAME = VALUE;

For example:

```
public class TwixBars
{
      public static final int CALORIES_PER_TWIX = 568;
      public static void main(String[] args)
      {
            int age = 60;
            int weight = 210;
            int feet = 5;
            int inches = 6;

            double kilograms = weight / 2.2;
            double centimeters = ((feet * 12) + inches) * 2.54;
```

```
            double calories =  10 * kilograms + 6.25 *
                               centimeters - 5 * age + 5;
            double numTwix = calories / CALORIES_PER_TWIX;

            System.out.println("You need to eat " + numTwix +
                  " twix bars per day to maintain your weight.");
      }
}
```

It is not required, but normal practice for programmers to spell named constants using all uppercase letters.

What is this syntax saying? The word "public" says that the constant can be accessed anywhere in the program. The word "static" means that there will be only one copy of this particular value. The word "final" means that this is the final value, i.e. that this is a constant that may not be changed later.

The use of named constants can make your program easier to read and modify than if the actual value were scattered throughout the program. By using the constant, if the calories ever changes, there is only one place that needs to be changed. When applicable, a constant is also a bit more efficient than using a variable in terms of speed of execution.

Keyboard Input

So far we have "hard coded" all values we want directly in our code. For example, in the twix calorie calculator we input the number of calories, height, weight, and age directly as variables in our program. It would be nice to be able to run our program, have it ask the user to enter their age, weight, and height, then calculate the number of twix bars to eat. This makes the program much more general and functional with easily-changed values instead of having to change the values in the program, recompile it, and then re-run the program.

To input data from the keyboard we use the **Scanner** class. Here is an example:

```
import java.util.Scanner;

class ScannerTest
{
      public static void main(String[] argv)
      {
            int i;
            float f;
            double d;

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

```
            System.out.println("Enter an integer. ");
            i = keyboard.nextInt();
            System.out.println("You entered: " + i);

            System.out.println("Enter a double. ");
            d = keyboard.nextDouble();
            System.out.println("You entered: " + d);

            System.out.println("Enter a float. ");
            f = keyboard.nextFloat();
            System.out.println("You entered: " + f);

            // Now sum up all three things entered as a double
            d = d + (double) f + (double) i;
            System.out.println("The sum of all three is: " + d);
      }
}
```

We start with the "`import java.util.Scanner`" which gives us access to the code library containing the Scanner class, which allows us to input data from the keyboard. This line should be near the top of the file.

Inside the main method the line "Scanner keyboard = new Scanner(System.in);" creates a Scanner object that takes System.in as a parameter, which refers to input from the keyboard. This is basically defining a variable with a data type of "Scanner" and the name of the variable is "keyboard". We could have used a different name.

With the keyboard variable we can now use the nextInt(), nextDouble(), or nextFloat() **methods** to return numbers of those particular data types typed from the keyboard. A method is a name for a group of code that performs some specific task – in this case, it reads in whatever is typed "next" as a double, integer, float, etc. This method is defined for you as part of the Java language. Later we will see that we can write our own methods.

Here is the output for a sample run:

```
Enter an integer.
5
You entered: 5
Enter a double.
2.412
You entered: 2.412
Enter a float.
3.14
You entered: 3.14   // Note roundoff errors below!
The sum of all three is: 10.552000104904174
```

Inputting Strings

Reading input from the keyboard for strings is almost the same as inputting numbers, except there is a pitfall when reading an entire line vs. reading a single number or word.

To read in a single word (surrounded by whitespace, i.e. spaces, tabs, carriage returns, etc.) use the next() method:

```
public static void main(String[] argv)
{
      int i;
      String s;
      Scanner keyboard = new Scanner(System.in);

      System.out.println("Enter an integer. ");
      i = keyboard.nextInt();
      System.out.println("You entered: " + i);

      System.out.println("Enter a word. ");
      s = keyboard.next();
      System.out.println("Your word was: " + s);
}
```

This program reads in an integer, prints it out, then reads in a word and prints it out.

For example:

```
Enter an integer.
5
You entered: 5
Enter a word.
coffee
Your word was: coffee
```

If you enter more than one word note that the next() method ONLY reads the next word:

```
Enter an integer.
5
You entered: 5
Enter a word.
need more coffee
Your word was: need
```

To read a whole line we can use the method nextLine():

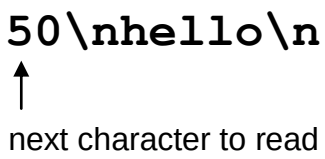
```
public static void main(String[] argv)
{
      int i;
      String s;
      Scanner keyboard = new Scanner(System.in);
```

```
        System.out.println("Enter an integer. ");
        i = keyboard.nextInt();
        System.out.println("You entered: " + i);

        System.out.println("Enter a line of text. ");
        s = keyboard.nextLine();
        System.out.println("Your line was: " + s);
    }
```

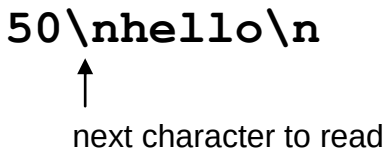
You would probably expect it to behave like the first program, but the program immediately ends without prompting you to enter a line of text. The reason is because the nextLine() method behaves a little differently than the other methods.

Java interprets its input as a stream of characters. When you press the enter key, a newline character is inserted into the stream. Let's say that you are giving the input of "50" for the integer and "hello" for the text. Java sees this input as:



The \n is a single character for newline.

When we execute: i = keyboard.nextInt();
Java reads the 50 and advances its input buffer to reference the next character in the stream, the \n:

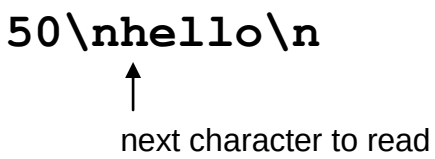


If we execute: s = keyboard.next(); (or nextInt() or nextDouble, etc.)

then Java will **skip any whitespace** characters looking for a word that is surrounded by whitespace. You can test this by entering leading spaces, newlines, tabs, etc. They will all be skipped and Java will return back only the word you enter.

However, if we execute: s = keyboard.nextLine();

then Java will stop until it hits a newline character, and consume it. It just so happens that the next whitespace is the \n so we end up with:



And the string s contains a blank string.

The solution? Use an extra nextLine() after a nextInt(), next(), nextDouble(), etc. to skip the whitespace character if you want to later read in an entire line:

```
public static void main(String[] argv)
{
      int i;
      String s;

      Scanner keyboard = new Scanner(System.in);

      System.out.println("Enter an integer. ");
      i = keyboard.nextInt();
      System.out.println("You entered: " + i);

      keyboard.nextLine();   // Skip \n character

      System.out.println("Enter a line of text. ");
      s = keyboard.nextLine();
      System.out.println("Your line was: " + s);
}
```

Formatted Output

So far we have been using System.out.println to output all of our messages. This method outputs whatever is inside the parentheses and then adds a newline (future output goes to the next line).

If you don't want output to go to the next line then use System.out.print instead of System.out.println. For example:

```
System.out.print("The Seawolf mascot's name is ");
System.out.println("Spirit");
```

Outputs:

```
The Seawolf mascot's name is Spirit
```

You can use combinations print() and println() to control how you output different variables.

You have probably noticed that when you output floating point numbers you are often getting more decimal places than you really want. In an earlier example, the output was 10.552000104904174 but you might want to only output it as 10.55. An easy way to do this is to use the printf() method which allows you to format your output.

Here is an example that shows a few format specifiers, one to print an integer, an integer in a field of 5 spaces, and a floating point number with 2 digits after the decimal point:

```
        int num1, num2;
        double num3;
        num1 = 1;
        num2 = 9;
        num3 = 5.58831;
        System.out.printf("An integer:%d\nAn integer in 5 spaces:%5d\nA
floating point number with 2 decimal places:%.2f\n",num1,num2,num3);
```

Output:

```
        An integer:1
        An integer in 5 spaces:    9
        A floating point number with 2 decimal places:5.59
```

The % symbol is used in the printf statement as placeholders to substitute values that are specified after the string.

In this case "%d" is a placeholder for the first argument to come after the string which is the variable num1. "%d" stands for an integer, so it will output as an integer the value in num1.

"%5d" is the next placeholder and it means to output an integer, but do so in a field of 5 characters. You can use this if you want to make columns of numbers line up nicely, for example, to make a table. In this case it matches up with the second argument, which is the value stored in num2.

"%.2f" is the next placeholder and it means to output a floating point number with 2 places after the decimal point. It matches up with the third argument, which is the value stored in num3. Note that the number is rounded up when it is printed out.

There are more format specifiers the next most common one is %s which is a placeholder for a String – see the book for details.