

# Module 1 Day 5

Command Line Applications: Inputs and Outputs  
Methods  
Command Line

You Made It! Happy Friday!

# Module 1 Day 5

Can you?

1. ... use `System.in/System.out/Console.ReadLine()` to perform console I/O in a program
2. ... parse input from the input stream to primitive data types
3. ... check for string equality
4. ... string apart using known split character
5. ... explain the process of a command line application (Take input, calculate data, give output)
6. ... run your command line apps in your IDE

# Methods

# Methods

- Methods are **related** statements that complete a specific task or set of tasks.
- Methods can be called from different places in the code.
- When methods are called, they can be provided inputs or arguments to provide data for their parameters.
- Methods can also return a value to its caller. That return value is determined by the called method return type.

# Methods: General Syntax

Here is the general syntax:

```
<access Modifier> <return type> <name of the method> (... arguments...) {  
  
    // method code.  
  
}
```

- The return type can be any of the **data types** (boolean, int, float, etc.) we have seen so far\*\*  
-or-
- The return type can be “**void**”, which means the method performs a task and returns nothing.

# Methods: Example

Here is a specific example of a typed method:

```
public class MyClass {  
    public int addTwoNumbers(int a, int b) {  
        return a+b;  
    }  
}
```

The method `addTwoNumbers` is a method of the `MyClass` class.

The method has a return value of `int`, so there needs to be a return statement that returns an integer.

The method expects 2 parameters as input. More specifically, it expects 2 integers

# Methods: Example

... and this is an example of a void method:

```
public class MyClass {  
  
    public void addTwoNumbers(int a, int b) {  
        System.out.println(a+b);  
    }  
}
```

This method is void; has no return statement. It only performs the task of sending the value of parameter ***a*** plus parameter ***b*** to the console output stream.



# Methods: Calling A Method

Methods can be called from other methods!

```
public class MyClass {  
  
    public void callingMethod (String args[]) {  
  
        int result = addTwoNumbers(3,4);  
        System.out.println(result);  
  
        String userName = fullName("Rich", "Seeds");  
        System.out.println(userName);  
    }  
  
    public int addTwoNumbers(int a, int b) {  
        return a+b;  
    }  
  
    public String fullName(String first, String last) {  
        return last + ", " + first;  
    }  
}
```

Here, in MyClass() **callingMethod()** first calls the method **addTwoNumbers(int, int)**, saving its return to the variable **result**.

Then, **callingMethod()** calls **fullName(String, String)**, providing all needed parameters and saving the result to the variable **userName**.

# Methods: Calling A Method

Once a method has been defined, it can be called from somewhere else.

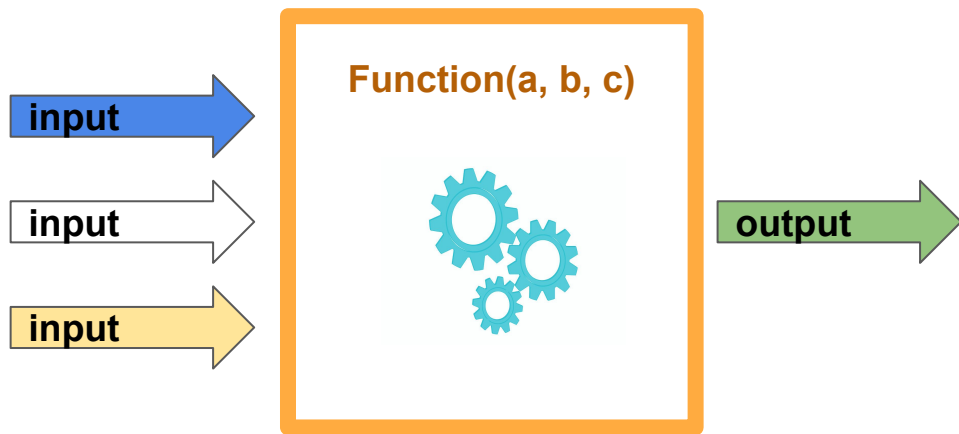
```
public class MyClass {  
  
    public int addTwoNumbers(int a, int b) {  
        return a+b;  
    }  
  
    public void callingMethod (String args[]) {  
  
        int result = addTwoNumbers(3,4);  
        System.out.println(result);  
        // result will be equal to 7.  
    }  
}
```

**addTwoNumbers** takes 2 inputs, integer a and an integer b. These are known as **parameters**.

When we call **addTwoNumbers**, we must provide the exact inputs specified (in this case 2 integers).

# Methods: In Summary

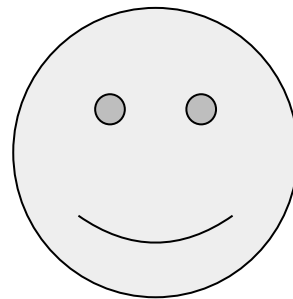
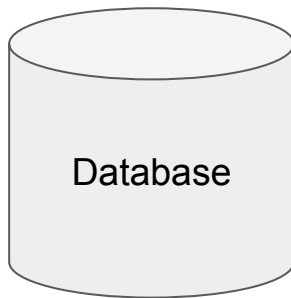
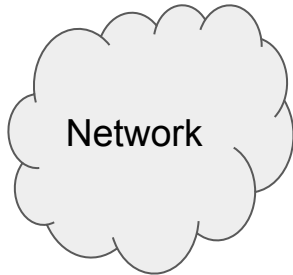
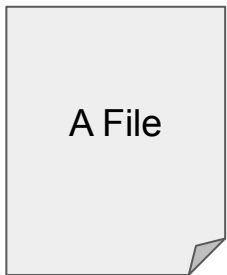
Methods are Java's versions of functions. Think of them as a processes that can take inputs (or none at all) and use it to generate output (or do work).



# Command Line Input / Output

# Getting Input from the Command Line

- All programming languages must have the ability to read data (input) in order to interact with the user and the systems, or world, around them
- Examples of input: a file, data being transmitted from a network, a datastore, or **data typed in by the user!**



# Using the Scanner Object

```
import java.util.Scanner;
```

To use the scanner object, we must import in the correct class.

```
public class InputReader {
```

Create an object of type scanner

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

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

The input is read and stored into a String called name.

```
        System.out.print("Please enter your name: ");
```

```
        String name = userInput.nextLine();
```

```
        System.out.print("Please enter your height: ");
```

The input is read and stored into a String called heightInput.

```
        String heightInput = userInput.nextLine();
```

```
        int height = Integer.parseInt(heightInput);
```

heightInput is converted into an int using the **Integer Wrapper Class**.

```
        System.out.println("Your name is: " + name + ".");
```

```
        System.out.println("Your height is: " + height + " cm's.");
```

```
    }
```

```
}
```

# Reading In Multiple Items

```
import java.util.Scanner;

public class InputReader {

    public static void main(String[] args) {

        Scanner userInput = new Scanner(System.in);
        System.out.print("Please enter several objects: ");
        String lineInput = userInput.nextLine();

        String [] inputArray = lineInput.split(" ");

        for (int i=0; i < inputArray.length; i++) {
            System.out.println(inputArray[i]);
        }
    }
}
```

This is one possible way to handle input for more than one item.

- When prompted a user enters each item separated by a space.
- The split method separates out each time using the spaces, and puts all of the items into an array!

# Processing the Multiple Items Read

```
1 import java.util.Scanner;
2
3 public class InputReader {
4
5     public static void main(String[] args) {
6
7         Scanner userInput = new Scanner(System.in);
8         System.out.print("Please enter several objects: ");
9         String lineInput = userInput.nextLine();
10
11         String [] inputArray = lineInput.split(" ");
12
13         for (int i=0; i < inputArray.length; i++) {
14             System.out.println(inputArray[i]);
15         }
16     }
17 }
```

Console x

<terminated> InputReader [Java Application] C:\Program Files\Java\jre1.8.0\_211\bin\javaw.exe (Sep

Please enter several objects: Ford GM Chrysler Toyota Honda Nissan BMW

Ford

GM

Chrysler

Toyota

Honda

Nissan

BMW

The user entered each car brand separated by a space

The whole input is “split” and repackaged as an array



# Wrapper Classes

- Up until now, we have seen most of the primitive data types, to name a few: **int**, **boolean**, **char**, **long**, **float**...
- You have also seen some non-primitive types: **Strings** and **Arrays**
- You might have noticed that non-primitive types seem to have extra functionality that can be invoked with the dot operator, for example: **(myArray.length)**.
- All the primitive data types have more powerful non-primitive equivalents, these are called **wrapper classes**. You have seen an example of this.

```
int height = Integer.parseInt(heightInput);
```

\* albeit this example uses a static method of the wrapper class (more on this at a later date)

# Wrapper Classes

Unboxing ← → Boxing

Primitive	Wrapper	Example of Use
int	Integer	Integer myNumber = 3;
double	Double	Double myDouble = 3.1;

Declaring a variable using the Wrapper class gives you a bit more flexibility. For example, the wrapper classes let you run certain utility methods by using the dot operator.

```
Integer myNumber = 3;  
String myStringNumber = myNumber.toString();
```

In the above example we have used a Wrapper class, **Integer**, and then a method of that class, **.toString()**, to convert the value to a String. In general, if you know type conversions are needed, wrapper classes are a good idea.