A series on variable types
This episode starring
Instance Variables

First: Review input

```
import java.util.Scanner; // Or import java.util.*;
                          // First way is preferred
public class ReadConsole
{
   public static void main(String[] args)
   {  Scanner cin = new Scanner(System.in); //instantiate

      System.out.print("Enter an integer: ");
      int a = cin.nextInt();
      System.out.print("Enter an integer: ");
      int b = cin.nextInt();

      System.out.println(a + " * " + b + " = " + a * b);
   }
}
```

Scanner methods

These are for **ints** (integers). There are also Scanner methods available for floats, etc, which we'll see later on in the quarter.

`nextInt()`

Does something with the int

`hasNextInt()`

Checks to see if there is an int

`nextLine()`

Replaces the int in the keyboard buffer with a newline character so the program won't use the int again

Variables in Depth

- **After Today**'s class on "**instance variables**" you should have the coding wherewithal to set up the **counters** and be able to **collect** the number of moves made, how many times the robot moved in any particular direction, and then **print** out the whole shebang at the end of the program.
- You will use instance variables to complete your maze navigation homework.

Variable Types

Variable Type	How it is defined	Scope (how long it lasts)	Uses
Local Variable	<type> Name = value; In function or method code block.	Until the closing bracket for the code block it is in.	Temporary use in a method, or loop.
Parameters	<type> Name in method declaration. Value is passed when method is called	For the duration of the Method. Copies value inside of variable.	Passing information to a Method.
Instance Variables (Non-static Fields)	<type> Name during class definition	For the lifetime of an object (an instance of the class)	Values that help define an instance of the class and persist through multiple service calls
Class Variables (Static Fields)	Static <type> Name during class definition	For the life time of the class (can be accessed through class definition)	Values that are the same for EVERY instance of the class

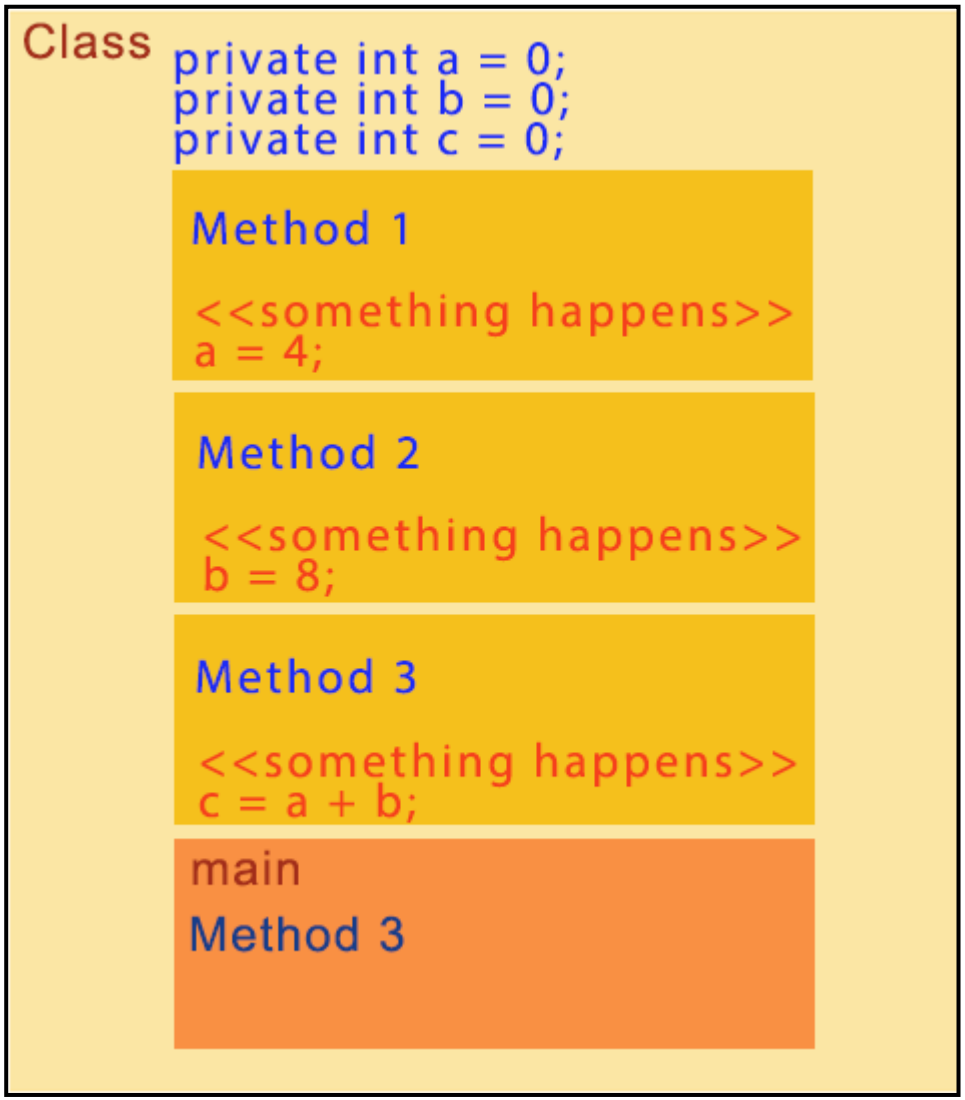
Instance Variables

- Local (temporary) variables disappear at the end of the code block in which they are declared.
- Parameter variables disappear at the end of the service (function/method) to which they are provided.
- But, our objects want to REMEMBER, so, we use INSTANCE VARIABLES!, which can retain values for the lifetime of the object. (This is how we make class/object attributes.)

Instance Variables

We declare the variables *inside* the class, but *outside* all the methods. This way all methods can use them.

We also usually declare them as **private** so only objects instantiated from the class that declared them can use and/or alter them, protecting them from outside interference (hence, *private*).



Access Modifers

Access Type	How it is defined	What is means	Uses
Public	Keyword public	Everyone can see	To allow non-class members access (often methods/services)
Protected	Keyword protected	Only package and subclass members can see	To give open access within a package, but enforce ownership outside the package.
Private	Keyword private	Only class members can see; private methods are not inherited by subclasses	To enforce ownership of a value or method to the class (often instance variables)
Package Private	No extra keyword; this is the default setting	Similar to protected	Similar to protected

Where to put code

With Class Scope (eg., in a class, before all the methods)	**If it is naturally a trait, or behavior, of the class. Things defining a robot, or a robot's behavior, go in the robot class.**
Within a class's method (eg., in one of the methods defined as part of a class)	**If it is a clear-cut behavior that is repeated, and it operating on data from that class only. Robots move, but they don't add walls to a city.**
In another class	**If there is a natural separation of data and behavior. Robots and Things are very different, so they have different classes.**
In the method of a different class	**If the behavior is clear-cut, and repeated, and perhaps operates on many different classes contained by the new class. A robot game method may control one or more robots.**

***NOTE: These are all building blocks. A complex class may be made of many simpler classes, and a complex method may use many simpler methods.**

Overloading a Constructor

Useful when you create a new subclass with more attributes than the general super class.

Create a new type of Robot that also starts with a limited number of actions it's allowed to do when the program is run, say for example, 20 total actions. You can add another argument to its constructor parameters to hold this number.

- You would create a new class that extends the Robot class
- You could create an instance variable to hold data, like the total number of actions
- You could create a new argument in the constructor's parameter to enter this additional data
- You could assign this entered data inside the constructor to the instance variable to be used elsewhere in the program for whatever purposes you need

Example on Next Page →

Constructor Overload Example

```
class RechargableRobot extends Robot {
  private int numOfActions; // number of actions executed instance-variable
  private int maxNumOfActions; // max number of actions instance-variable

  RechargableRobot( City c, int st, int ave, Direction dir, int num, int numActions) {
     super(c, st, ave, dir, num); // ← Notice NO additional argument here for the superclass
     this.maxNumOfActions = numActions; // set the max number of actions
  }
  // Additional Methods go here
}

public class ExampleNewArgument extends Object {
   public static void main(String[] args) {
      City toronto = new City();
      RechargableRobot Jo = new RechargableRobot(toronto, 3, 0, Direction.EAST, 0, 10);
      new Thing(toronto, 3, 2);
      // Additional Code goes here
   }
}
```

Creating Named Constants with **final**

- Many programs have data that does not need to be changed.
- Littering programs with **literal values** can make the program hard to read and maintain.
- Replacing literal values with **constants** remedies this problem.
- **Constants** allow the programmer to use a **name** rather than a value throughout the program.
- **Constants** also give a **singular point** for changing those values when needed.

Creating Named Constants with **final**

- **Constants** keep the program **organized** and easier to maintain.
- **Constants** are **identifiers** that can hold only a **single value**.
- **Constants** are declared using the keyword `final`.
- **Constants** need not be initialized when declared; however, they must be initialized before they are used or a compiler error will be generated.

Creating Named Constants with **final**

- Once initialized with a value, constants cannot be changed programmatically (can not be changed later on in the code).
- By convention, constants are all upper case and words are separated by the underscore '_' character.

```
final float CAL_SALES_TAX = 0.75;
```

Both the **Java** and **Becker** API libraries have several constants built in. For example:

Java has **math.PI** (where PI = 3.14159265)

Becker has **direction.NORTH** (and **EAST, SOUTH, WEST**) where the directions represents specific degrees on a compass (0,90,180,270).

Constant Example

```
class RechargableRobot extends Robot {
  private int numOfActions; // number of actions executed instance-variable
  private int final MAX_ACTIONS=10; // max number of actions instance-variable

  RechargableRobot( City c, int st, int ave, Direction dir, int num) {
     super(c, st, ave, dir, num);
     this.numOfActions = 0;
  }
  // Additional Methods go here
}

public class ExampleNewArgument extends Object {
   public static void main(String[] args) {
      City toronto = new City();
      RechargableRobot Jo = new RechargableRobot(toronto, 3, 0, Direction.EAST, 0, 10);
      new Thing(toronto, 3, 2);
      // Additional Code goes here
   }
}
```