BIT115: Introduction to Programming
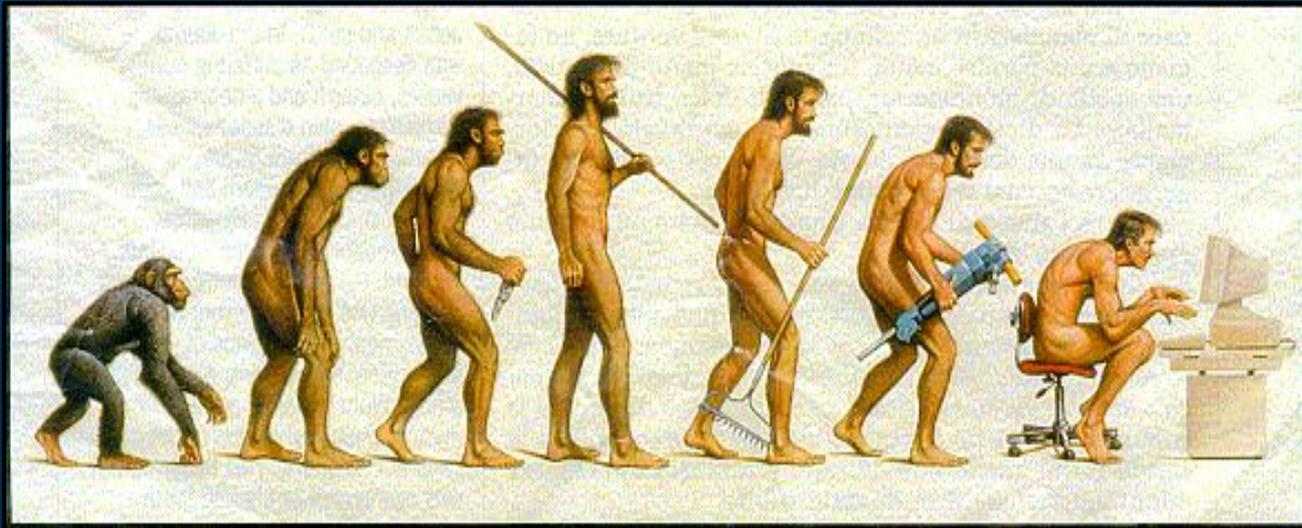
Lecture 12

Instructor: Craig Duckett

Instance Variables

Assignment Dates (By Due Date)

- **Assignment 1** (LECTURE 5)
  Section 1: Wednesday, October 11th
  Section 3: Thursday, October 12th

- **Assignment 2** (LECTURE 9)
  Section 1: Wednesday, October 25th
  Section 3: Thursday, October 26th

- **Assignment 1 Revision** (LECTURE 11)
  Section 1: Monday, November 6th
  Section 3: Thursday, November 2nd

- **Assignment 2 Revision** (LECTURE 13)
  Section 1: Monday, November 13th
  Section 3: Thursday, November 9h   The Fickle
                                               Finger of Fate

- **Assignment 3** (LECTURE 15)
  Section 1: Monday, November 20th

- *Section 3*: Thursday, November 16th

- **Assignment 3 Revision** (LECTURE 18)
  Section 1: Wednesday, November 29th
  Section 3: Thursday, November 30th

- **Assignment 4** (LECTURE 21) **NO REVISION AVAILABLE!**
  Section 1: Monday, December 11th
  Section 3: Tuesday, December 12th

Instance Variables

Instance variables are declared <u>inside</u> the class, but <u>outside</u> the methods or constructor

Class

Constructor

Instance Variable

Method

Method

Method

Main

Instance Variables

Temporary/Local Variables *versus* Instance Variables

All the **local variables** we've looked at so far will **disappear** at the end of the service (method call). This means that they're ***temporary***.

We want memory that <u>won't</u> go away between service calls. We want the robot to *remember* something, to be able to keep a <u>running tally</u> that can be called up *outside* the scope of the method, not just get some temporary working space. An **instance variable** is what we need.

A **temporary variable** can <u>only</u> be used *inside* the <u>method</u> where it is created.

An **instance variable** may be used **anywhere** in the **class** containing it, by **any** of its methods. **Remember**: *inside* the class, but *outside* the **methods** and **constructor**.

We might also manage this by declaring it **private**, which is foreshadowing the beginnings of **encapsulation** (although we aren't quite *there* yet), although we don't have to for what we'll be doing with it.

These are Temporary Variables

Method

```
public void moveMultiple(int numberOfIntersections)
{
        int counter = 0;
        while( counter < numberOfIntersections)
        {
            if(this.frontIsClear())
            {
              this.move();
               counter = counter + 1;

            }
         }
    }
```
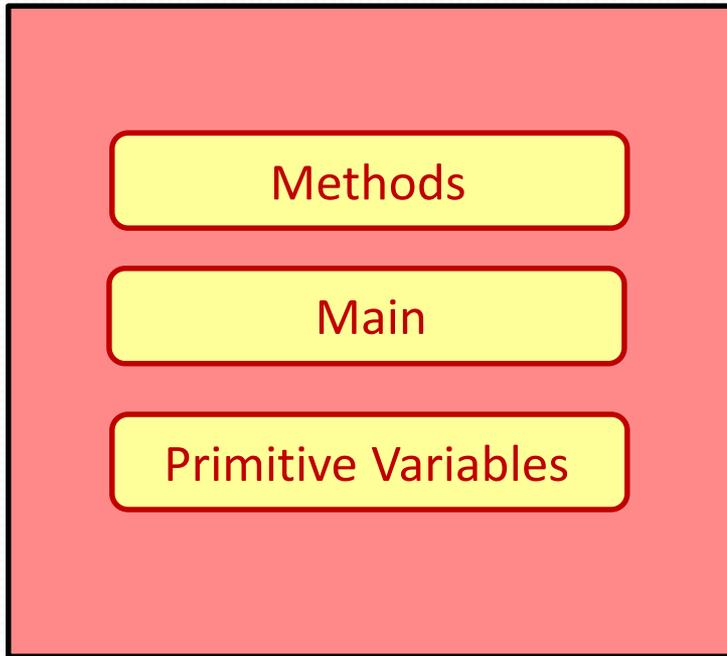

numberOfIntersections

counter

Main
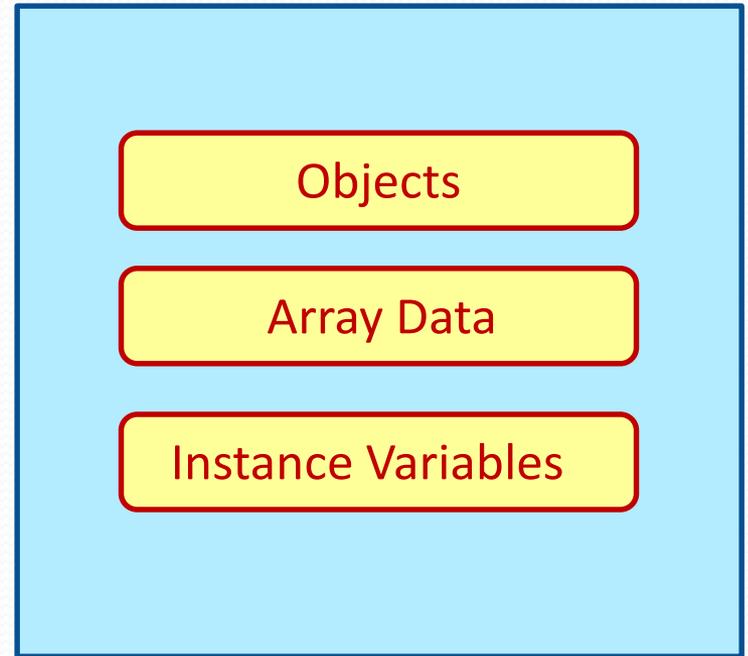
```
System.out.println("How many intersections forward would you like the robot to go?");

        if( keyboard.hasNextInt() )
        {
            int numMoves = keyboard.nextInt();
            System.out.println ("You entered a " + numMoves + ".");
            rob.moveMultiple(numMoves);
        }
```
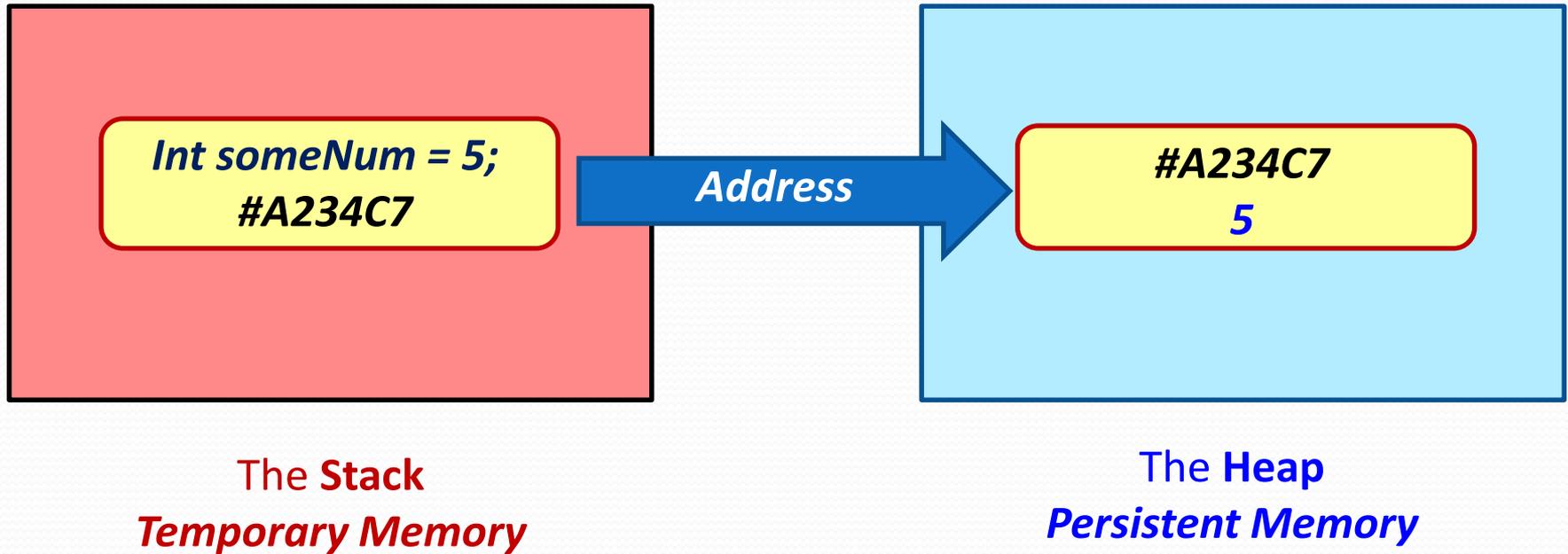
Temporary Variables

Suppose we want to recall a number from one method to use in a different method? With local (or temporary) variables we can't do that since the variable space is no longer available once we leave the method

This does not work as planned because **int x** and **int y** are not available to **Method3**
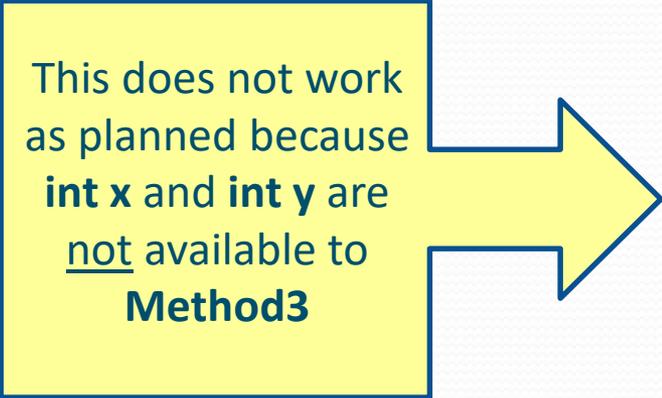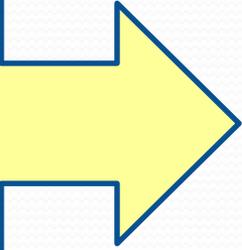
So how do we get around this limitation of local variables?

Class

Method 1
int a = 0;
<<something happens>>
a = 4;

Method 2
int b = 0;
<<something happens>>
b = 8;

Method 3
int c = 0;
<<something happens>>
c = a + b;

main
Method 3

We Do It With **Instance Variables**

We declare the variables <u>inside</u> the class but <u>outside</u> the methods, that way <u>all</u> the methods can access and use them

We also 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*).

```
Class
    private int a = 0;
    private int b = 0;
    private int c = 0;

    Method 1

    <<something happens>>
    a = 4;

    Method 2

    <<something happens>>
    b = 8;

    Method 3

    <<something happens>>
    c = a + b;

    main
    Method 3
```
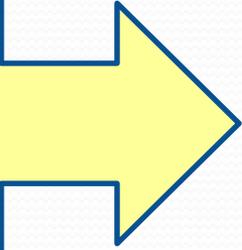
We Do It With **Instance Variables**

We declare the variables <u>inside</u> the class but <u>outside</u> the methods, that way <u>all</u> the methods can access and use them

See:

InstVar_Demo_1.java
InstVar_Demo_2.java

Class
```
private int a = 0; 4
private int b = 0; 8
private int c = 0; 12
```

Method 1

```
<<something happens>>
a = 4;
```

Method 2

```
<<something happens>>
b = 8;
```

Method 3

```
<<something happens>>
c = a + b;
```

main

Method 3

Adding Another Argument to Constructor

FYI!

Useful when you create a new robot Object with other functionality besides just its placement in the City

Let's say you want to 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 (although you'll *still* have to write the code to actually *do something* with this number)!

So, how would you do this?

- You would create a new **class** that **extends** the **Robot** class
- You could create an **instance variable** to hold data
- You could create a **new argument** in the **constructor's parameter** to enter this additional data
- You could assign this entered data <u>inside the constructor</u> to the instance variable to be used elsewhere in the program for whatever purposes you need.
- In <u>main</u>, you would have your new robot object <u>do something</u> with this new number (either *subtract* from it or *add* to it)

```java
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, 20);
    new Thing(toronto, 3, 2);
    // Additional Code goes here
  }
}
```

Some Helpful Hints for Assignment 3

- **HINT #1**: This assignment is using **RobotSE** which contains additional methods that can be used in the code for this assignment. For example: **turnRight()**, **turnaround()**, **isfacingEast()**, **isFacingSouth()**, **isFacingWest(), isFacingNorth()**, etc. See the **Becker Library** for reference.

- **HINT #2**: You will need to declare and initialize five (5) instance variables!
  These will be used to keep a running tally of the following:
  - Total Number of Moves Made
  - Total Number of Moves East
  - Total Number of Moves South
  - Total Number of Moves West
  - Total Number of Moves North

- **HINT #3:** You should create a method (e.g., **movesCounted()**) that will *first* keep a running tally of all the moves mentioned above and *then* **move()**. EXAMPLE: if the robot is facing east, count it.

- **HINT #4:** You should create a method (e.g., printEverything()) that will print out the tally of all the moves made <u>at the end</u> of the **NavigateMaze()** method.

- **HINT #5:** Instead of using **move()** in the **NavigateMaze()** method, use **movesCounted()** which will not only *move*, but *tally* the move to the *instance variables*.

- **HINT #6:** All you need to add to the NavigateMaze() method to successfully navigate the Maze is fourteen (14) lines of code—and this includes the squiggles! If you need to use more lines with your logic, that's okay too (it doesn't *only* have to be 14). Also, the robot will only put down a thing if there is a thing in the backpack to put down, otherwise will still move without putting down a thing.

- **HINT #7:** Make sure and add backpack items to the robot object, don, down in main! Changing the 0 to at least 100 would be nice ☺ (I will be testing your code with 100 items in backpack, and 33, and 10, and 0).

```java
class MazeBot extends RobotSE // <-- NOTE This is RobotSE class, not Robot class
{
    public MazeBot(City theCity, int str, int ave, Direction dir, int numThings)
    {
        super(theCity, str, ave, dir, numThings);
    }

    // Five (5) Instance variables are declared and initialized here

    public void movesCounted()
    {
        //First tally the move and direction faced, then actually move
    }

    public void printEverything()
    {
        // Print everything (moves and directions faced) tallied to the instance variables above
    }

    private boolean isAtEndSpot()
    {
        return (this.getAvenue() == 9 && this.getStreet() == 10);
    }

    public void NavigateMaze()
    {
        while( !this.isAtEndSpot() )
        {
            // All you really need is 14 lines of code to navigate Maze, including squiggles,
            // making sure only to put down a thing if there is a thing in the backpack, otherwise
            // just move without putting down a thing! Of course, if you need to use more than
            // 14 lines of code that's okay too (for instance, 18 or 20 or 24 lines, but not 200!)
        }

        // Print everything here
    }
}

// Finally, down in main, change the number of things in backpack to at least 100
```