BIT 115 Lecture 2

Today

- Notes on software
- Tracing Code
- Types of Errors
- Debug tables
- In Class Exercises

Java and JGrasp

- Office hours are directly after class today.
- Always download the java files, save them in your BIT115 folder (the same one that holds your becker.jar file), and then edit that copy.
  - In some browsers you can ‘right click’ and then ‘save as’ or ‘download as’. In other browsers the file downloads automatically. On some school computers the MS Edge browser will not let you download all file types; try using Chrome.
  - Make sure you create a local copy of the file, and pay attention to where you put it. Make sure to use the same file name – if you download more than one copy you may have a (1) or a (2) appended to the file name – remove those numbers.

Work and Assessments

- Save a copy of all of your work
  - Subfolders of your BIT115 folder do well for this.
  - Make sure you have a copy even if you work on something with a classmate
- Exams will be pencil and paper
  - Please bring pencils to class.
  - Practice writing code from memory

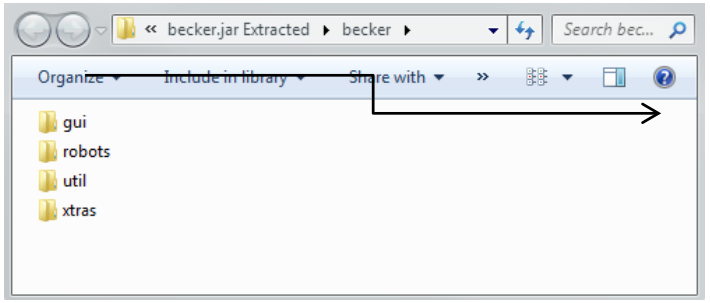
Dissecting the Code:

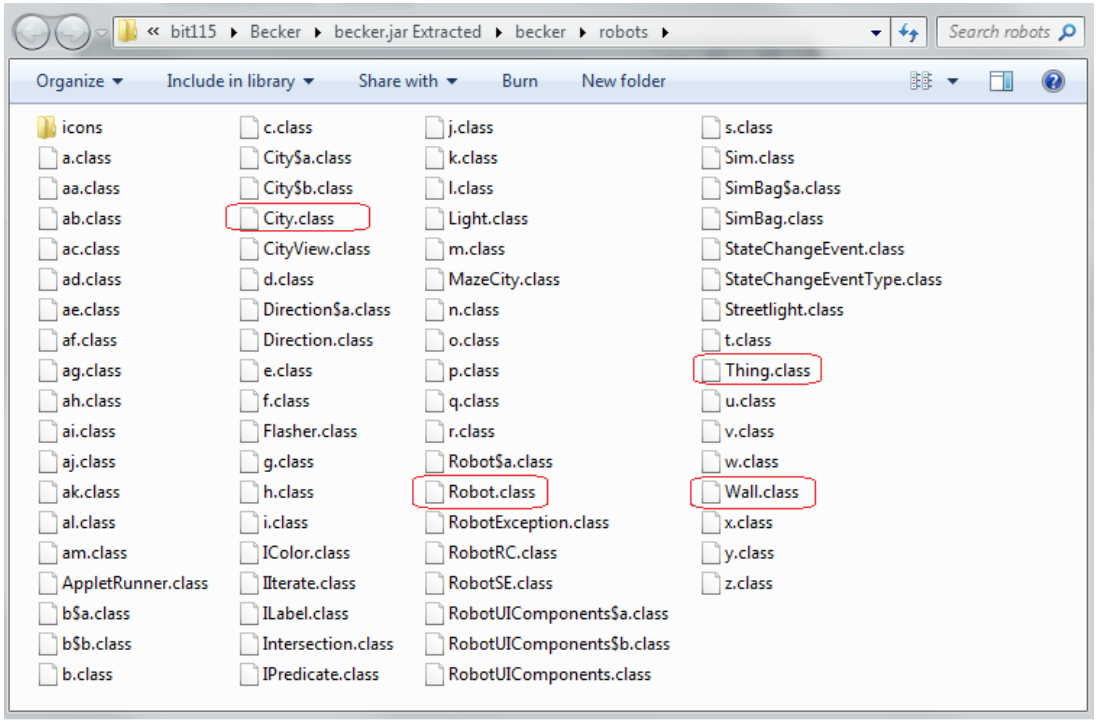
What It Means and What It Does (Quiz2.java)

```
1   import becker.robots.*;
2
3
4    public class Quiz2 extends Object
5    {
6       public static void main(String[] args)
7      {
8          City toronto = new City();
9
10         Robot Jo = new Robot(toronto, 0, 3, Direction.NORTH, 0);
11
12         new Thing(toronto, 2, 2);
13         new Wall(toronto, 3, 3, Direction.EAST);
14         new Wall(toronto, 3, 3, Direction.NORTH);
15         new Wall(toronto, 3,3, Direction.SOUTH);
16
17         Jo.turnLeft();
18
19      }
20   }
```

`import becker.robots.*;`

This imports from the **becker.jar** collection all the **classes** contained in the **robots** file so they would be functionally available in Java (JDK) and any code written using it. These classes include the **Robot.class**, the **Thing.class**, the **City.class**, the **Wall.class**, and over a hundred more.





NOTE: Java loads *some* classes by default into the compiler, while others it does not. Later on we will have to import additional Java classes in our code to get our programs to work

public class Quiz2 extends Object

---

public 'public' so other classes can use it.

class a 'class' is a blueprint for making 'objects' that can do something

Quiz2 'Quiz2' is the name of the class. It can be any name. The program uses this name when it runs to know what 'objects' and 'methods' to use in the program.

extends Object Java, like other object-orientated languages, supports class inheritance. Inheritance allows one class to "inherit" the properties of another class. For example, all Java objects are inherited from the java.lang.Object class which is a fundamental part of Java and the Java libraries. Our **Quiz2** class is inheriting the properties of Java's **Object** class, and so **extends** its actions and attributes in the program hierarchy.

http://www.tutorialspoint.com/java/java_object_classes.htm

Note: main is a method() inside a Class

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

The **method** is **public** because it is accessible to the JVM (Java Virtual Machine) to begin execution of the program, and can be accessed by any other class or method.

The **static** keyword signifies the fact that this method can be invoked without creating an instance of that class (an object). Main is called before any objects are made, hence *static*.

void signifies that this method does not **return** anything. In other words no new or altered data is sent out from it for use by the rest of the program. Nothing is returned, so void.

Just like in the C "procedural" language and other languages, this is the main method in a Java program. When you run a Java program, execution begins in the **main** method.

String is the data **type** that could be passed to the method, although no data type needs to be passed. An example of a data type is an **int** or **float**. The **[]** represents unknown quantity.

args is the name of the parameter. We can pass any number of arguments to this method.

```
City toronto = new City();
```

Names the object ← Creates the new City object

We want to make a **new** instance of the **City** class (City.class) found in the **becker > robots** directory (*inside* **becker.jar**) and call this new object *Toronto*

The **City** class contains all the attributes and actions necessary to set up a city when a named city **object** (like **toronto**) is created, include **shape**, **size**, **color**, **streets**, **avenues**, etc.

By itself, the City class can't do anything. It's just a collections of *ideas* and *concepts*. You need an **object** made from the City class (like **toronto** here) to actually do something with a city.

NOTE:
In Java, '=' is an assignment operator, and points right to left; "==" is an equals sign.

EXAMPLE:
x = 1 assigns 1 to variable space x and x == 1 means x equals 1.

```
1   import becker.robots.*;
2
3
4    public class Quiz2 extends Object
5    {
6       public static void main(String[] args)
7      {
8          City toronto = new City();
9
10         Robot Jo = new Robot(toronto, 0, 3, Direction.NORTH, 0);
11
12         new Thing(toronto, 2, 2);
13         new Wall(toronto, 3, 3, Direction.EAST);
14         new Wall(toronto, 3, 3, Direction.NORTH);
15         new Wall(toronto, 3,3, Direction.SOUTH);
16
17         Jo.turnLeft();
18
19      }
20   }
```

```
Robot Jo = new Robot(toronto, 0, 3, Direction.NORTH, 0);
```

Names the Robot ← Creates the new Robot using five parameters

- Our *instance* of the Robot *object* is named **Jo**
- Jo is placed in a *City* called **Toronto**
- Jo is starting out on **Street 0**
- Jo is starting out on **Avenue 3**
- Jo is starting out facing **North**
- Jo is starting out with **0** (zero) *Things* in its backpack

Now, another way to "construct" this is with only four parameters, by leaving on the number of Things in the backpack

```
Robot Jo = new Robot(toronto, 0, 3, Direction.NORTH);
```

If you are not going to be picking up or putting down Things in your program, then you can "construct" your Robot without this fifth 'backpack' parameter.

Later on, when we start creating our own types of Robots and methods, some of the ICES will be set up in such a way that parts of the code will use five parameters and other parts of the code will use four parameters and this will cause an error. *We'll go over this in greater detail when the time comes.*

```
new Thing(toronto, 2, 2);
new Wall(toronto, 3, 3, Direction.EAST);
new Wall(toronto, 3, 3, Direction.NORTH);
new Wall(toronto, 3, 3, Direction.SOUTH);
```

These will work by default. For stuff that just sits there, we don't have to actually give them unique names (e.g., BrickWall) but we can't talk about them in code, however a Robot can pick up/put down an unnamed object. Why would you want to give a wall a unique name?

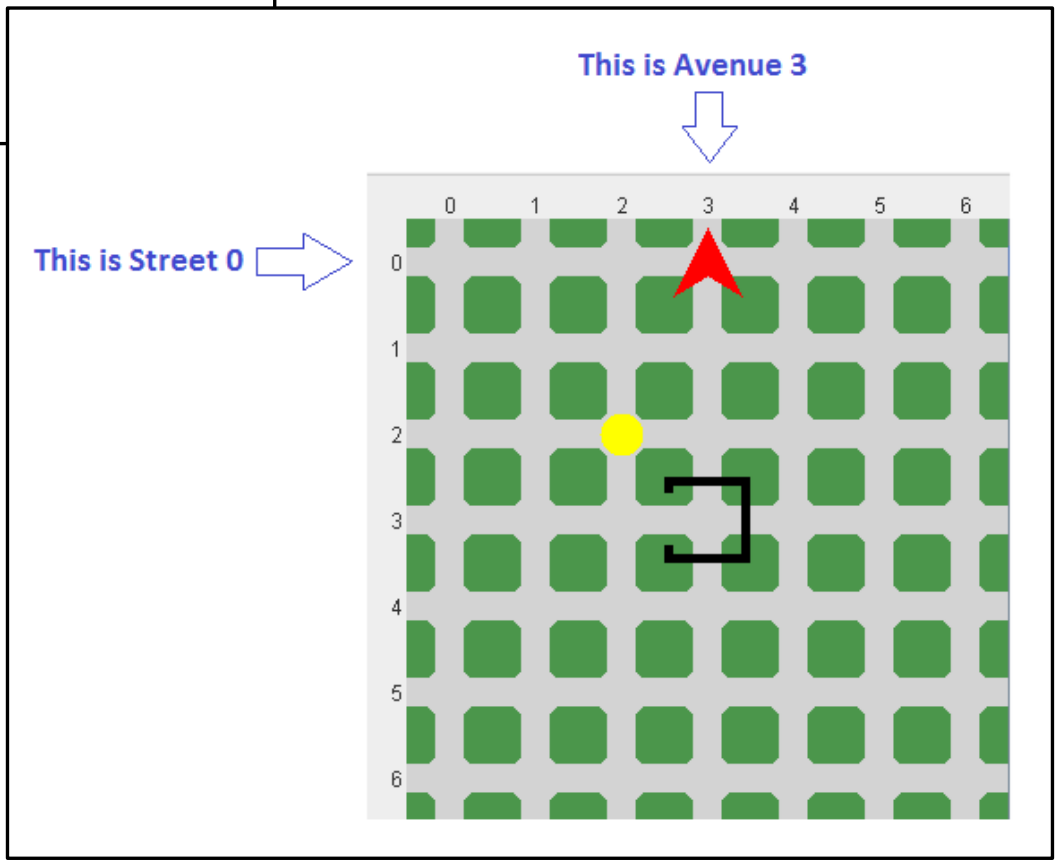
If you look at the becker library you will discover that Wall actually extends Thing, that is to say Wall *has* ***inherited*** *all the initial properties of* Thing

```
Jo.turnLeft();
```

turnLeft() is one of the methods of the **Robot** class, along with **move**(), **pickThing**(), **putThing**(), **frontIsClear**(), **countThingsInPackback**(), and several more. Because Robot can turn left, Jo can turn left.

ect Settings Tools Window Help

```
import becker.robots.*;

public class Quiz2 extends Object
{
   public static void main(String[] args)
   {
      City toronto = new City();

      Robot Jo = new Robot(toronto, 0, 3, Direction.NORTH, 0);

      new Thing(toronto, 2, 2);
      new Wall(toronto, 3, 3, Direction.EAST);
      new Wall(toronto, 3, 3, Direction.NORTH);
      new Wall(toronto, 3,3, Direction.SOUTH);

      Jo.turnLeft();

   }
}
```



Something to Remember:

In parameter order, street comes <u>first</u> and avenue comes <u>second</u>.

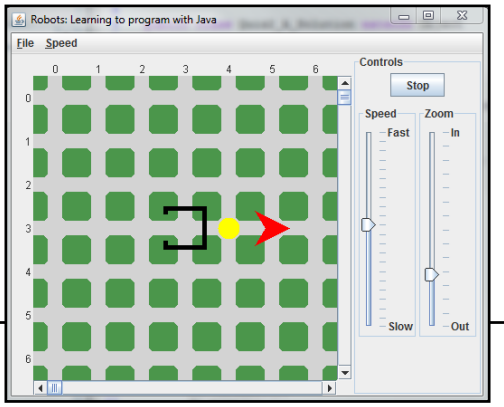
Streets go **West-East** (left-right)

Avenues go **North-South** (up-down)

A neat trick to remember the difference is to recall that the 'A' and 'V' in Avenue point up and down.

Chapter 1.4.5 – Tracing Code

```
1   import becker.robots.*;
2
3
4   public class Quiz2_A_Solution extends Object
5   {
6      public static void main(String[] args)
7      {
8         City toronto = new City();
9
10        Robot Jo = new Robot(toronto, 3, 3, Direction.NORTH, 0);
11
12        new Thing(toronto, 2, 2);
13        new Wall(toronto, 3, 3, Direction.EAST);
14        new Wall(toronto, 3, 3, Direction.NORTH);
15        new Wall(toronto, 3,3, Direction.SOUTH);
16
17        Jo.turnLeft();
18        Jo.move();
19        Jo.turnLeft();
20        Jo.turnLeft();
21        Jo.turnLeft();
22        Jo.move();
23        Jo.pickThing();
24        Jo.turnLeft();
25        Jo.turnLeft();
26        Jo.turnLeft();
27        Jo.move();
28        Jo.move();
29        Jo.turnLeft();
30        Jo.turnLeft();
31        Jo.turnLeft();
32        Jo.move();
33        Jo.putThing();
34        Jo.turnLeft();
35        Jo.move();
36     }
37  }
38
39
```



PROGRAM TRACE TABLE

Program Source File: Quiz2_A_Solution.java

Line #	Program Statement	Robot's Street #	Robot's Ave #	Robot's Direction	Robot's Backpack Contents	Thing's Location
8	City toronto = new City();	-	-	-	-	-
10	Robot Jo = new Robot(toronto, 3, 3, Direction.NORTH, 0);	3	3	NORTH	0	(2,2)
12	new Thing(toronto, 2, 2);	3	3	NORTH	0	(2,2)
13	new Wall(toronto, 3, 3, Direction.EAST);	3	3	NORTH	0	(2,2)
14	new Wall(toronto, 3, 3, Direction.NORTH);	3	3	NORTH	0	(2,2)
15	new Wall(toronto, 3,3, Direction.SOUTH);	3	3	NORTH	0	(2,2)
17	Jo.turnLeft();	3	3	WEST	0	(2,2)
18	Jo.move();	3	2	WEST	0	(2,2)
19	Jo.turnLeft();	3	2	SOUTH	0	(2,2)
20	Jo.turnLeft();	3	2	EAST	0	(2,2)
21	Jo.turnLeft();	3	2	NORTH	0	(2,2)
22	Jo.move();	2	2	NORTH	0	(2,2)
23	Jo.pickThing();	2	2	NORTH	1	-
24	Jo.turnLeft();	2	2	WEST	1	-
25	Jo.turnLeft();	2	2	SOUTH	1	-
26	Jo.turnLeft();	2	2	EAST	1	-
27	Jo.move();	2	3	EAST	1	-
28	Jo.move();	2	4	EAST	1	-
29	Jo.turnLeft();	2	4	NORTH	1	-
30	Jo.turnLeft();	2	4	WEST	1	-
31	Jo.turnLeft();	2	4	SOUTH	1	-
32	Jo.move();	3	4	SOUTH	1	-
33	Jo.putThing();	3	4	SOUTH	0	(3,4)
34	Jo.turnLeft();	3	4	EAST	0	(3,4)
35	Jo.move();	3	4	EAST	0	(3,4)

```
 1    import becker.robots.*;
 2
 3
 4    public class Quiz2_A_Solution extends Object
 5    {
 6       public static void main(String[] args)
 7       {
 8          City toronto = new City();
 9
10          Robot Jo = new Robot(toronto, 3, 3, Direction.NORTH, 0);
11
12         new Thing(toronto, 2, 2);
13          new Wall(toronto, 3, 3, Direction.EAST);
14          new Wall(toronto, 3, 3, Direction.NORTH);
15          new Wall(toronto, 3,3, Direction.SOUTH);
16
17          Jo.turnLeft();
18          Jo.move();
19          Jo.turnLeft();
20          Jo.turnLeft();
21          Jo.turnLeft();
22          Jo.move();
23          Jo.pickThing();
24          Jo.turnLeft();
25          Jo.turnLeft();
26          Jo.turnLeft();
27          Jo.move();
28          Jo.move();
29          Jo.turnLeft();
30          Jo.turnLeft();
31          Jo.turnLeft();
32          Jo.move();
33          Jo.putThing();
34          Jo.turnLeft();
35          Jo.move();
36       }
37    }
```

PROGRAM TRACE TABLE

Program Source File: Quiz2_A_Solution.java

Line #	Program Statement	Robot's Street #	Robot's Ave #	Robot's Direction	Robot's Backpack Contents	Thing's Location
8	`City toronto = new City();`	-	-	-	-	-
10	`Robot Jo = new Robot(toronto, 3, 3, Direction.NORTH, 0);`	3	3	NORTH	0	(2,2)
12	`new Thing(toronto, 2, 2);`	3	3	NORTH	0	(2,2)
13	`new Wall(toronto, 3, 3, Direction.EAST);`	3	3	NORTH	0	(2,2)
14	`new Wall(toronto, 3, 3, Direction.NORTH);`	3	3	NORTH	0	(2,2)
15	`new Wall(toronto, 3,3, Direction.SOUTH);`	3	3	NORTH	0	(2,2)
17	`Jo.turnLeft();`	3	3	WEST	0	(2,2)
18	`Jo.move();`	3	2	WEST	0	(2,2)
19	`Jo.turnLeft();`	3	2	SOUTH	0	(2,2)
20	`Jo.turnLeft();`	3	2	EAST	0	(2,2)
21	`Jo.turnLeft();`	3	2	NORTH	0	(2,2)
22	`Jo.move();`	2	2	NORTH	0	(2,2)
23	`Jo.pickThing();`	2	2	NORTH	1	-
24	`Jo.turnLeft();`	2	2	WEST	1	-
25	`Jo.turnLeft();`	2	2	SOUTH	1	-
26	`Jo.turnLeft();`	2	2	EAST	1	-
27	`Jo.move();`	2	3	EAST	1	-
28	`Jo.move();`	2	4	EAST	1	-
29	`Jo.turnLeft();`	2	4	NORTH	1	-
30	`Jo.turnLeft();`	2	4	WEST	1	-
31	`Jo.turnLeft();`	2	4	SOUTH	1	-
32	`Jo.move();`	3	4	SOUTH	1	-
33	`Jo.putThing();`	3	4	SOUTH	0	(3,4)
34	`Jo.turnLeft();`	3	4	EAST	0	(3,4)
35	`Jo.move();`	3	4	EAST	0	(3,4)

Chapter 1.5: Types of Errors

3 Different General Categories of Errors:

- ✓**Compile-Time Errors (Syntax Errors)**
- ✓**Run-Time Errors (Application Errors)**
- ✓**Intent (Logical) Errors**

Compile-Time Errors (Syntax Errors)

Anything that goes wrong when you compile the file is a compile-time error!
Point out that the Output Panel is often small, but it can be resized, and you should get used to figuring out which line the error pertains to.

Java is case sensitive
- **Java** is different from **java**, is different from **JaVa**
- This means you must type in names, etc, EXACTLY the same

File name MUST be the same as the class name

You need the **import becker.robots.*;** at the top of each robot file

Forgetting to Compile *BEFORE* Running
- You need to compile the file EVERY SINGLE TIME you change it
- You should also wait UNTIL the compilation finishes BEFORE trying to run it

Strategy For Fixing Compile-Time Errors:
Follow the debugging strategy (listed in your text) to find and correct the syntax errors in the **FindErrors.java** program later this evening when you do your **In-Class Exercises**.
1. Compile the program to get a list of errors;
2. Fix the most obvious errors, beginning with the first error reported
3. Compile the program again to get a revised list of the remaining errors.

Run-Time Errors (Crashing Errors)

Anything that causes the program to crash while it's running. If you encounter a run-time error when you are compiling and running your Java programs, then this would be a jGrasp problem, or a Windows Operating System problem, and NOT a Java Problem. The Java language does a lot to protect you from this type of error, so you shouldn't see many of these.

Intent Errors (Logical Errors)

The program compiles and runs without crashing, but it doesn't do what you want it to.

Example:

- **Robot** takes an extra **leftTurn**, and runs off the screen
- **Robot** doesn't pick up a **Thing** when it's supposed to.
- **Robot** collides with a **Wall** (the book calls this a run-time error, but it technically isn't)

As you continue learning to code you'll make fewer "typo" type mistakes, and end up with more **Intent** errors then **Compile-Time** errors.

You need to come up with a personal strategy to figure out where the Intent problems are, and how to fix them. One Option: Use a **Program Debug Table** (show where this is on web site)