

BIT 115 Lecture 2

Today

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

Java and JGrasp

- I will stay around today to help with laptops and other questions
- Installation problems with Win8 may be solved by download the classic shell:
 - http://download.cnet.com/Classic-Shell/3000-2072_4-75553853.html
- 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.

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 and some quizzes will be pencil and paper
 - Please bring pencils to class.
 - Practice writing code from memory

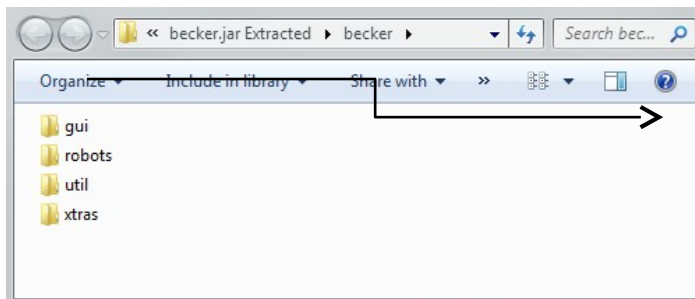
Dissecting the Code:

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

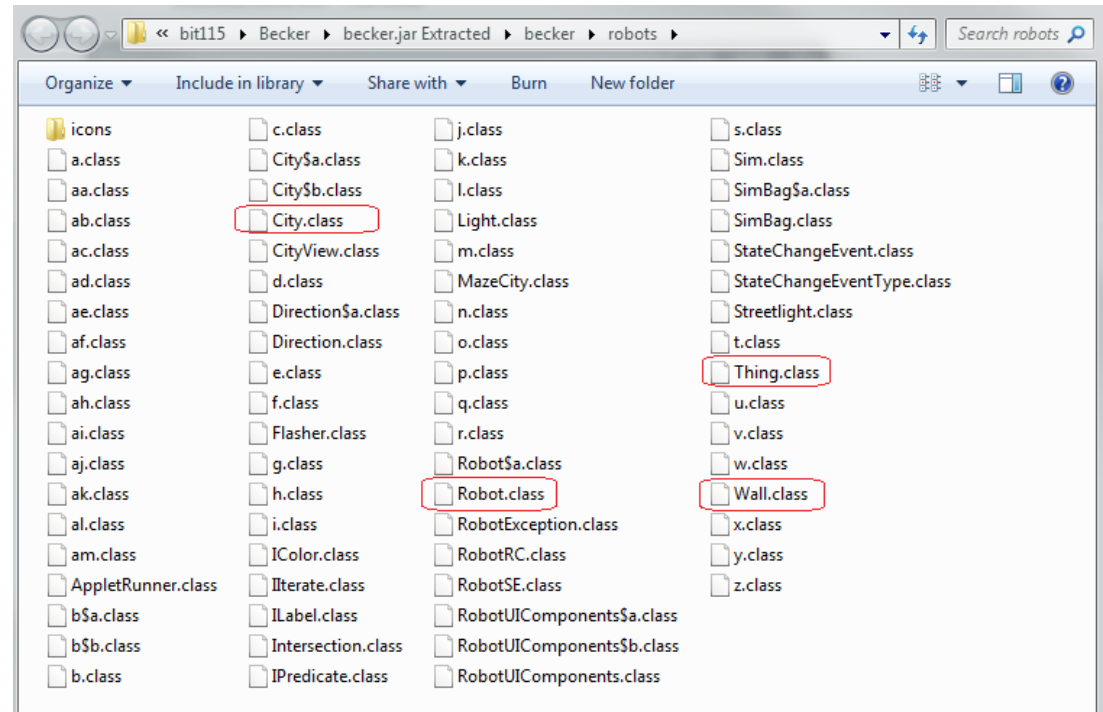
```
1  import becker.robots.*;
2
3
4  public class Quiz2 extends Object
5  {
6      public static void main(String[] args)
7      {
8          City toronto = new City();
9
10         Robot Jo = new Robot(toronto, 0, 3, Direction.NORTH, 0);
11
12         new Thing(toronto, 2, 2);
13         new Wall(toronto,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

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 first and avenue comes second.

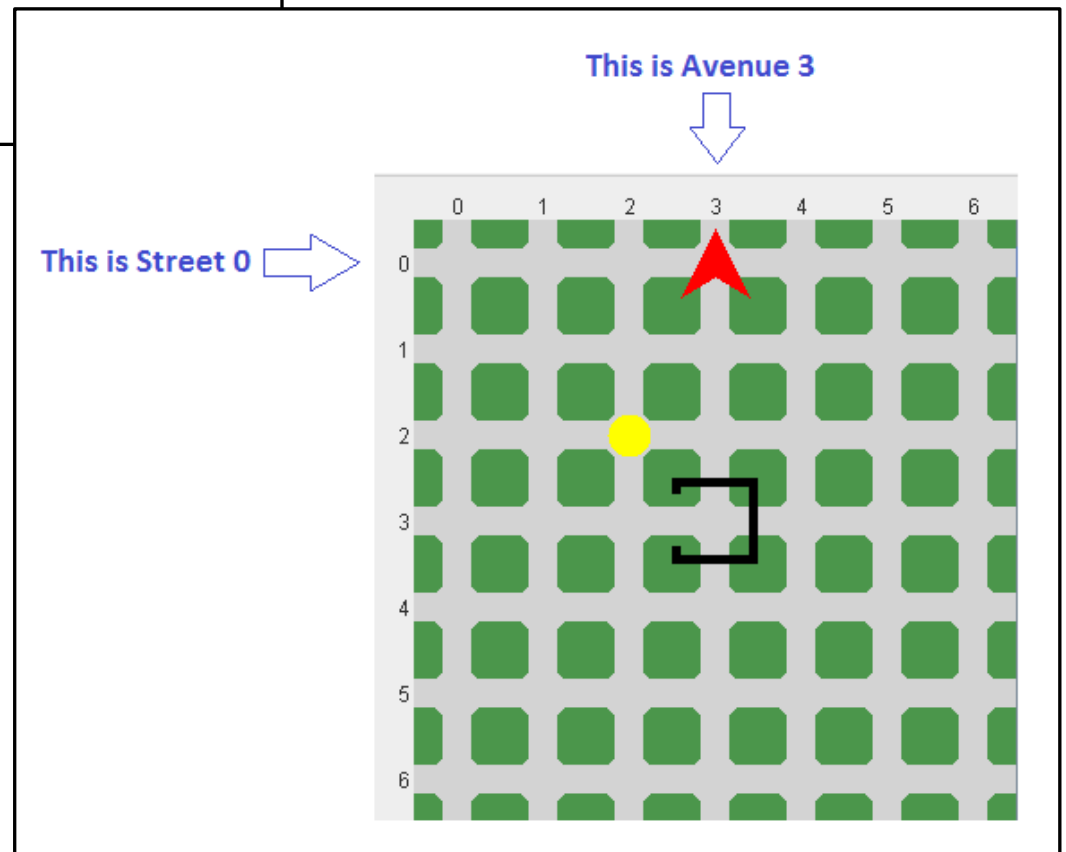
Streets go **West-East** (left-right)



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.

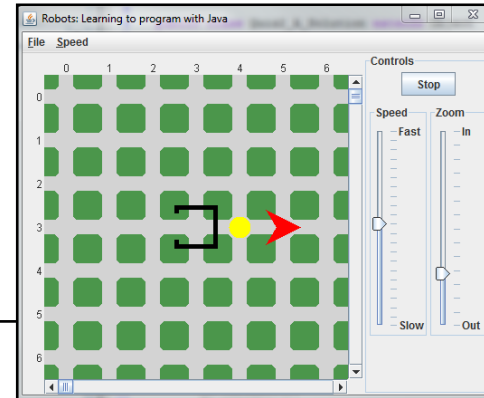


Chapter 1.4.5 – Tracing Code

```

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	<code>City toronto = new City();</code>	-	-	-	-	-
10	<code>Robot Jo = new Robot(toronto, 3, 3, Direction.NORTH, 0);</code>	3	3	NORTH	0	(2,2)
12	<code>new Thing(toronto, 2, 2);</code>	3	3	NORTH	0	(2,2)
13	<code>new Wall(toronto, 3, 3, Direction.EAST);</code>	3	3	NORTH	0	(2,2)
14	<code>new Wall(toronto, 3, 3, Direction.NORTH);</code>	3	3	NORTH	0	(2,2)
15	<code>new Wall(toronto, 3,3, Direction.SOUTH);</code>	3	3	NORTH	0	(2,2)
17	<code>Jo.turnLeft();</code>	3	3	WEST	0	(2,2)
18	<code>Jo.move();</code>	3	2	WEST	0	(2,2)
19	<code>Jo.turnLeft();</code>	3	2	SOUTH	0	(2,2)
20	<code>Jo.turnLeft();</code>	3	2	EAST	0	(2,2)
21	<code>Jo.turnLeft();</code>	3	2	NORTH	0	(2,2)
22	<code>Jo.move();</code>	2	2	NORTH	0	(2,2)
23	<code>Jo.pickThing();</code>	2	2	NORTH	1	-
24	<code>Jo.turnLeft();</code>	2	2	WEST	1	-
25	<code>Jo.turnLeft();</code>	2	2	SOUTH	1	-
26	<code>Jo.turnLeft();</code>	2	2	EAST	1	-
27	<code>Jo.move();</code>	2	3	EAST	1	-
28	<code>Jo.move();</code>	2	4	EAST	1	-
29	<code>Jo.turnLeft();</code>	2	4	NORTH	1	-
30	<code>Jo.turnLeft();</code>	2	4	WEST	1	-
31	<code>Jo.turnLeft();</code>	2	4	SOUTH	1	-
32	<code>Jo.move();</code>	3	4	SOUTH	1	-
33	<code>Jo.putThing();</code>	3	4	SOUTH	0	(3,4)
34	<code>Jo.turnLeft();</code>	3	4	EAST	0	(3,4)
35	<code>Jo.move();</code>	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 than **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)

Some Important Terms To Remember

A **class** is a set of data and methods that can work together to accomplish a task. It can contain or manipulate data, but does so according to a conceptual pattern rather than a specific implementation. A *single instance* of a class is an object.

An **object** receives all of the characteristics of a class, including all of its default data and any actions that can be performed by its functions. The object is for use with specific data or to accomplish particular tasks.

A **method** refers to a function that is encased in a class. It is code that acts on the data in a give class.

A **parameter** is a variable that is passed into a function that instructs it how to act or gives it information to process. Parameters are also sometimes called *arguments*.

A **field** or **property** is a default set of data stored in a class. A class can have multiple properties and can be changed dynamically through the methods of the class. These are sometimes called **attributes**.

Inheritance is one of the keys that make OOP tick. Simply put, classes can inherit methods and fields from other classes by extending them and each class can be extended by multiple classes. This means that you can start with a base (or parent) class that contains shared characteristics among several classes. That base class can then be extended by other classes (children) that are similar but are meant for slightly different purposes. Any changes in the parent class will automatically cascade to its children.

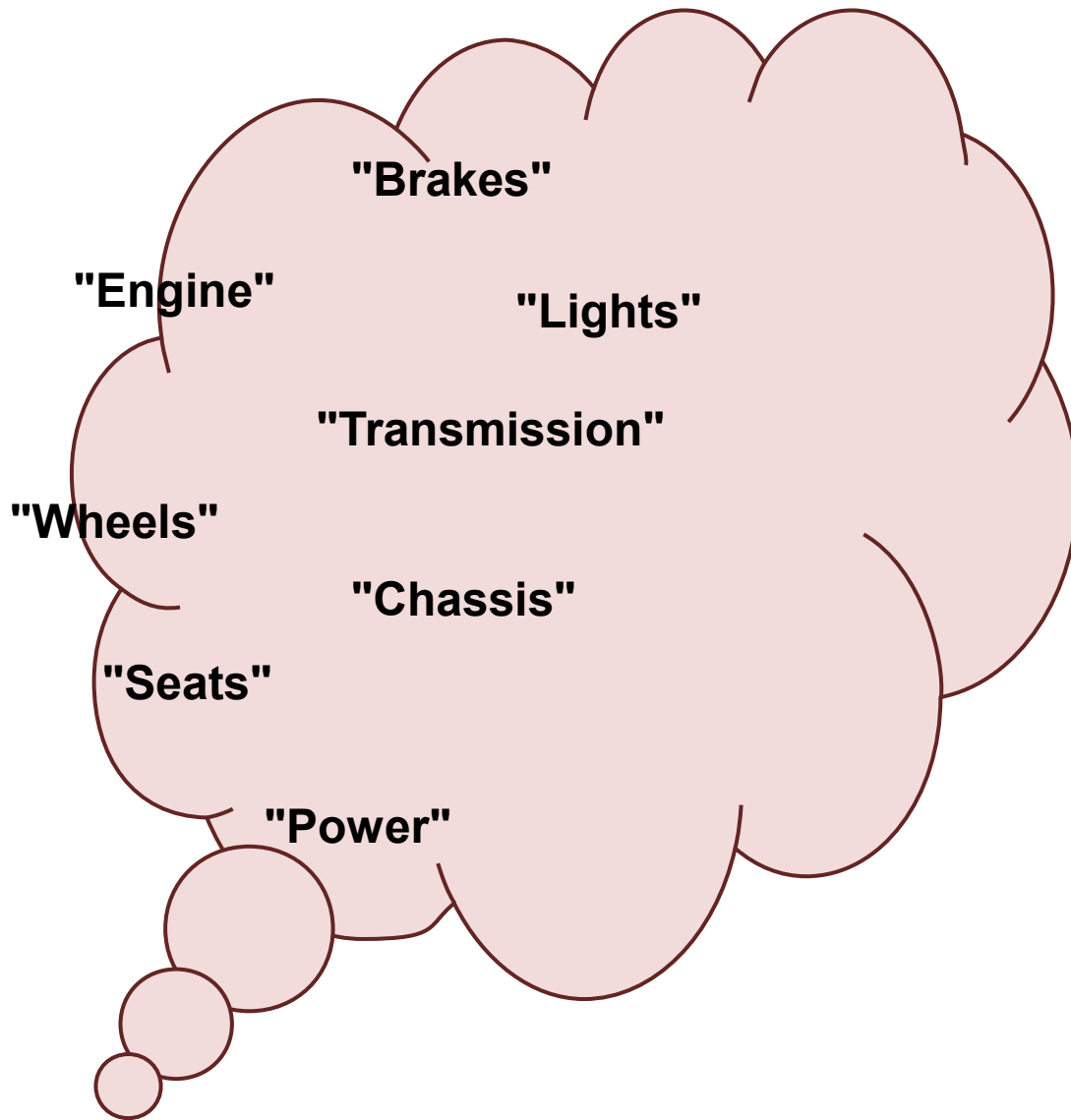
Class (The “Idea” of Features and Functions)

A **class** is a collection of functions that can work together to accomplish a task. It can contain or manipulate data, but it usually does so according to a pattern rather than a specific implementation. An *instance* of a class is considered an object. Until an object is *instantiated* from a class, the class can't actually do anything.

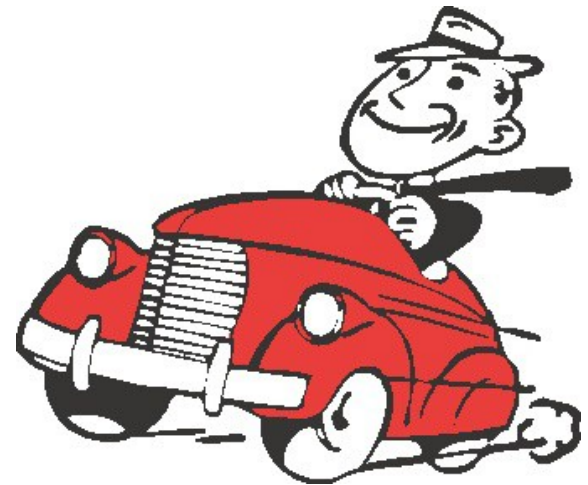
Example

The **Robot** class. The Robot class contains all the *features* and *functions* that a robot might have once it is created (or *instantiated*) as an *object*. Until then, all the Robot features and functions are just the *idea* how a robot might work, but being an idea the class doesn't do any work itself..

Just like we might have an idea how a car should look and operate—body, engine, four tires, steering wheel, gas and brake pedals, etc—an idea of a car is not the same of an actual working car. Here the *idea* of a car is a *class*, and an *actual working* car is an *object*.



"Car" Class



"Car" Object

Object (An Instance of the “Idea” Made Actionable)

An **object** receives all of the characteristics of a class, including all of its default data and any actions that can be performed by its functions. The object is for use with specific data or to accomplish particular tasks.

Example

When we create an **object** from the **Robot** class we do so by creating a *named instance* of the Robot class that will possess all the features and functions that were outlined as belonging to the Robot class as an *idea*. Once the object is *instantiated*, what was once the idea of a Robot now becomes an actual Robot that possesses specific ‘Robot’ *properties* (its shape, size, color, speed) and that can perform specific ‘Robot’ *actions* (like move, turn left, put down a thing, pick up a thing, etc).

Method (An “Action” That Might Be Performed by the Object)

A **method** simply refers to a function that is encased in a class. It usually entails an *action*.

Example

After we have created a named *object* from the Robot *class* that possesses all the *properties* (or *attributes*) belonging to robots, the robot can now perform some kind of *action* or *actions* (like move, turn left, put down a thing, pick up a thing, etc). These actions are the **methods** belonging to the Robot classes, and which an instance of the Robot class can perform. A robot can move because the Robot class has a method called `move()` which will allow the robot to move one space forward each time it is called upon to do so.

Although we might want the robot to *mix a martini*, the robot can not because the Robot class does not contain a `mixAMartini()` method. Later on in this course we will learn how we might make a new kind of class that could contain a `mixAMartini()` method, but the Robot class we are using now does not have just such a method.

Parameter (“Information” or “Instruction” That Might Be Given)

A **parameter** is a variable that is passed into a function that instructs it how to act or gives it information to process. Parameters are also sometimes called *arguments*.

Example

When we have a robot object call upon a method in order to perform some action, that method may or may not take a parameter which is an additional piece of information or an instruction that might extend its functionality.

Currently most of the methods belonging to the Robot class do not take any parameters (although in a moment we’ll look at a Robot method that *does* take a parameter). For example, the **move()** method when called upon will allow the robot to move one space forward. You’ll notice that the **move()** method has two “empty” parentheses which represents that it doesn’t take a parameter. Had the programmer of the Robot class designed the move() method differently, then it might have taken a parameter to represent how many places the robot should move. For example move(1) might make the Robot move one space forward, move(2) might make the Robot move two spaces forward, move(3) might make the robot move three spaces forward, and so on. Since the programmer did not design the move() method this way, using the move() method like this will not work. This was only a theoretical example.

Field or Property (The “Attributes” of Class and Its Objects)

A **field** or **property** is a default set of data stored in a class. A class can have multiple properties and can be changed dynamically through the methods of the class. These are sometimes called **attributes**.

Example

When we create a named instance of a robot from the Robot class, this robot object comes with a set of predefined properties and attributes. For example, by default the new robot object is so many pixels high and so many pixels wide, its shape is an arrowhead, its color is red, and its location and placement and direction in the City is defined by the parameters that were passed to it (which we will discuss in the next *Dissecting the Code* section).

For example, one of the properties of the Robot class is that a robot will move one space forward in .5 seconds. This speed, .5 seconds, is a pre-defined property of the Robot class and any instances of objects made using the Robot class.

Now these pre-defined properties might be changed dynamically while the program is running if just such a feature was developed programmatically to do so. In the case of the robot’s default speed of .5 seconds for every move, there is a `setSpeed()` method which can be used to change the robot’s initially defined speed. For example `setSpeed(4)` will move the Robot twice as fast (the default speed is 2).

Inheritance (Extending a Class's Features to a New Class)

Inheritance is one of the keys that make OOP tick. Simply put, classes can inherit methods and fields from other classes by extending them and each class can be extended by multiple classes. This means that you can start with a base (or parent) class that contains shared characteristics among several classes. That base class can then be extended by other classes (children) that are similar but are meant for slightly different purposes. Any changes in the parent class will automatically cascade to its children.

Example

We will learn about **inheritance** and how to put it to good use next week. A brief example what inheritance is can be explained by looking at our Robot class. Right now the Robot class has a `turnLeft()` method but it doesn't have a `turnRight()` method. If you created a Robot object and wanted it to turn right then you would have to call upon it to turn left three times!

Now we might create a new type of **Robot** class (perhaps called **MrRoboto**) that would extend all the *features* and *functions* of the Robot class (like `move()`, `turnLeft()`, etc) and also create its own new set of methods (like `turnAround()`, `turnRight()`, etc). In this example we are using the *original Robot* class to create our *new MrRoboto* class, and as such the **MrRoboto** class is *inheriting* all the original features and functions of the **Robot** class. This is a great tool of **Object Oriented Programming** (or **OOP**) because this means the programmer doesn't have to code all the Robot class features and functions from *scratch* to use in the new **MrRoboto** class. All the programmer has to do is code the new methods (like `turnAround()` or `turnRight()`) since the **MrRoboto** class inherited all the original methods (like `move()`, `turnLeft()`, etc).