BIT 115 Lecture 2A

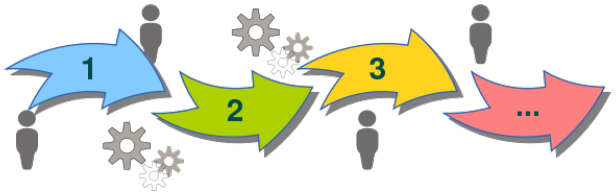
Today

- What is a Variable?
- What is a Method?
- Object Oriented Java
- In Class Exercises

Programming:
Data & Processes

Programs hold on to data

- Collected statistics
- Measurements
- Record of state



Programs process data

- Summarize statistics
- Respond to measurements
- Change state

Programming:
Data & Processes

Programs hold on to data

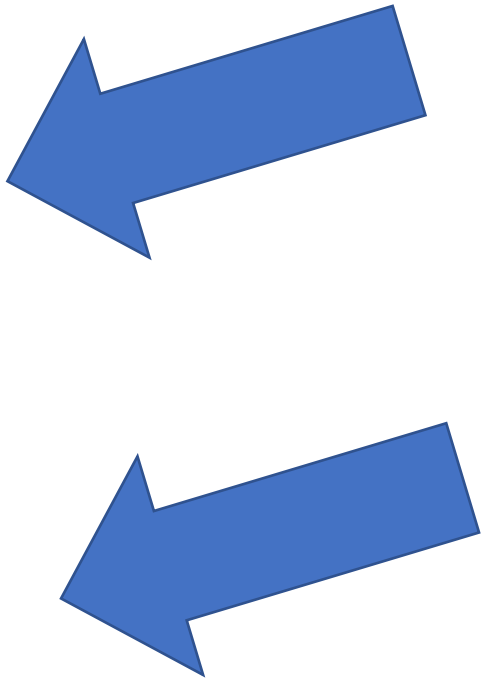
- Collected statistics
- Measurements
- Record of state

Variables / Attributes

Programs process data

- Summarize statistics
- Respond to measurements
- Change state

Methods / Functions / Processes



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 i*nstantiated* 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*.

"Brakes"

"Engine"

"Lights"

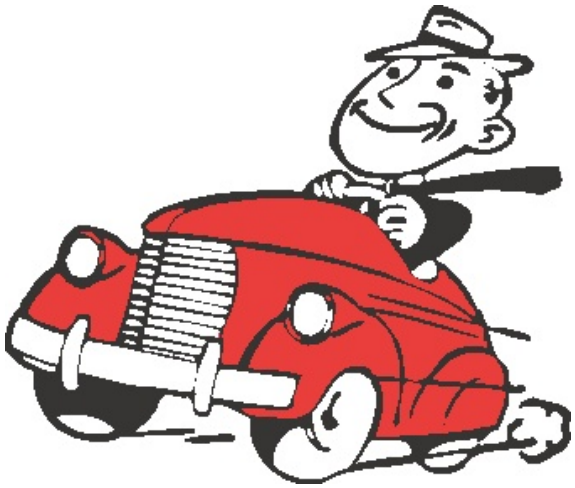
"Transmission"

"Wheels"

"Chassis"

"Seats"

"Power"



"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).

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   }
```

```
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.

```
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  }
```