Classes, Objects & Constructors
Classes

- Classes are a way to define **custom data types**
  - they represent a concept
  - they associate *data* (variables) with *operations* (methods) appropriate for that data.

- *Example:* String class
  - Represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
  - The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.
State & Behavior

- Classes have two things...
  - **state** represented by data fields (sometimes called properties)
  - **behavior** defined by methods (sometimes called member functions)

- For example, the String class has...
  - data such as the string of characters
  - methods such as `charAt()` or `length()`
Modeling Classes

- The book says "they model real world things" This is misleading and wrong.
  - We know Scanner is a class, what real-world object does it model?
- Classes represent some idea or concept that makes sense in the context of a program.
- We've seen lots of classes before.. Character, Scanner.. but it turns out we can create our own classes.
Creating Classes

- For example, let’s create a class description that can be used to describe a character from “The Simpsons”
State (aka Properties or Instance Variables)

- What are some of the characteristics of every Simpsons characters?
  - Name
  - Gender
  - Age
  - Catch Phrase
  - Occupation
Behaviors (aka methods or member functions)

- What are some of the behaviors of all Simpsons characters?
  - Introduce themselves
  - Say Catch Phrase
  - Go to work
  - ......
A Simpsons Class

- Once we have described what a generic Simpsons character is by defining its properties and its methods we can begin to construct a new class that can be used to simulate a Simpsons character.

- We can begin by creating a new class in Eclipse as we normally would. However, this class will look a little different than the classes we’ve been writing so far.
A Simpsons Class - Properties

- Note that the properties that we described for our generic Simpsons characters have been turned into "instance variables". These instance variables have "class scope".

- Note that they do not have the static modifier!

```java
public class SimpsonsCharacter {

    String name;
    char gender;
    int age;
    String catchPhrase;
    String occupation;

    public void speak() {
    }

    public void sayCatchPhrase() {
    }

}
A Simpsons Class - Methods

- We can also see our class methods or "instance" methods
- Note that they do not have the static modifier!
- They are able to access the instance variables above….
Creating Methods for Classes

```java
public class SimpsonsCharacter {
    String name;
    char gender;
    int age;
    String catchPhrase;
    String occupation;

    public void speak() {
        System.out.println("Hi, my name is " + name);
    }

    public void sayCatchPhrase() {
        System.out.println(this.catchPhrase);
    }
}
```
Creating Methods for Classes

- An important difference with the methods we will see today with the ones we have created in the past is that they do not have the “static” modifier.

- “static” is a keyword that tells Java that the method is not associated with a particular instance of a class – it indicates that the method is a “stand alone” method.
With me so far?

- Classes are the bundling of variables and methods into a logical entity that represents a concept in a program.
  - We say they have “state” and “behavior”

- Ok, so now that we have a class, how do we make use of it?

- We need to create objects!
What are objects?

- An object is an 'instance' of a class.

- One way we can understand the distinction by looking at things we already know about.
  - Somebody, somewhere, once upon a time, wrote some code for the String class in Java. We've used it many times.
  - When we call 'new' on a String to create one, we create an 'object' or 'instance' of a String.

```java
String s1 = new String("This is an instance of a String!");

String s2 = new String("This is another instance of a String!");
```
What are objects? *con’t*

```java
String s1 = new String("This is an instance of a String!");
String s2 = new String("This is another instance of a String!");
```

- What's different about these two Strings? Their 'data'. Their 'state'.
- Moreover, a String 'object' or 'instance' is a particular String that has its own data.
**Classes vs. Objects**

- A class in Java can be thought of as a “cookie cutter” that can be used to make any number of cookies with the same structure & layout.
Classes vs. Objects

- Classes do not describe a particular item. They are designed to be generic in nature.
- For example, our Simpsons class does not describe one character in detail (Bart, Maggie, Homer, etc).
Classes vs. Objects

- In order to create a concrete description of a Simpsons character we need to create an Object.

- An object is an ‘instance’ of a class. It allows Java to create a copy of the class and “fill in the blanks” with specific information about a particular item that it is representing.
Classes vs. Objects

- It’s sometimes easier to think about the difference between classes and objects like a recipe for a meal and the meal itself. The class is the recipe, but in order to eat the meal you need to prepare it and bring it into existence.

- We call the process of a class being used to create an object “instantiation” – we are creating a concrete “instance” of the class that uses the class definition as a “blueprint”
Creating “instances” of the Simpsons class

```java
public class TheSimpsons {

    public static void main(String[] args) {
        // Create an "instance" of our Simpsons class.
        // I.e. creating an object
        SimpsonsCharacter bart = new SimpsonsCharacter();
        // ...
    }
}
```
Accessing Data on an Object

- The “new” keyword tells Java to create an instance of the class in question. You can create as many instances of a class as you’d like.

- Each instance contains a reference variable to a copy of your class in memory (on the heap)

- The instance is a unique copy of a class. You can assign data to the member variables inside that class that are unique to that instance.

- You can access the member variables of an instance using dot syntax. For example, if our class has a field called “name” you can access that field by doing the following:

```java
SimpsonsCharacter bart = new SimpsonsCharacter();
bart.name = “Bart J. Simpson”;
```
Accessing Data on an Object

```java
public class TheSimpsons {
    public static void main(String[] args) {
        // Create an "instance" of our Simpsons class.
        // I.e. creating an object
        SimpsonsCharacter bart = new SimpsonsCharacter();

        // Assign data elements to our SimpsonsCharacter object
        bart.name = "Bart Simpson";
        bart.age = 12;
        bart.catchPhrase = "Eat my shorts!";
    }
}
```
The “this” keyword

- Note that when referring to instance variables you can prepend them with the “this” prefix.

- “this” is a prefix that tells Java that we are referring to an instance variable associated with the current class.

- “this” is not required in all cases. However, it can make your programs clearer – at least at the beginning – as it allows you to see which variables are instance variables and which variables are local variables associated with a particular method.

- There are cases however, where “this” is required. We’ll see it shortly.
As mentioned earlier, objects can have both properties and methods.

You can define a method for an object in the same way as you would for any other class.

You can call a method on an object using the same dot syntax that you would use to access an instance variable. For example:

```
// ask the characters to introduce themselves
bart.introduce();
homer.introduce();

// ask the characters to say their catch phrases
bart.sayCatchPhrase();
homer.sayCatchPhrase();
```
With me so far?

- An object is an “instance” of a class
- In other words, it’s a particular case of the class that has its own data.
  - By data we mean values in the instance variables.
- It turns out that we can have a lot of control over how our objects get created when they are “instantiated”
- We can control the instantiation process with ”constructors”
Constructors

- A constructor is a special kind of method that is invoked when you use the new keyword to create an instance of a class.

- We’ve used them before…. 
  
  - `Scanner input = new Scanner(System.in);`
  
  calling Scanner's constructor

- It turns out that we can define our own for our classes.

- And we can use these special methods to give initial values to our data members!
Constructors

- Constructors are methods, but they have some special behaviors and constraints.

- Here are the rules:
  
  - Constructors must have the same name as the class (i.e. if you create a class called Circle your constructor must also be called Circle)
  
  - Constructors do not have a return type. Not even void.

  - Constructors are automatically called when you use the ‘new’ keyword to create a new instance of an object.
Constructors

```java
public class CircleWithConstructor
{
    // data fields for a circle
    double radius;

    // constructor
    public CircleWithConstructor(double r)
    {
        // assign the supplied argument
        // to our radius
        this.radius = r;
    }
}
```
You can also define a constructor to not accept any arguments. In this case we refer to the constructor as a “no-argument” or “no-arg” constructor.

For example:

```java
public CircleWithConstructor() {
    // no argument, supply a default value
    // to radius
    this.radius = 1;
}
```
Overloading Constructors

- Constructors can be overloaded just like other methods.

- This means that an object can have multiple constructors with the same name, and Java will determine which one to call based on the arguments supplied.

```java
// user wants to set up a circle with a radius
public CircleWithConstructor(double r)
{
    // assign the supplied argument to our radius
    this.radius = r;
}

// user didn't supply a radius - just use 1.0
public CircleWithConstructor()
{
    // assign a default radius
    this.radius = 1.0;
}
```