Encapsulation & Access Modification
Encapsulation

- Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

- When we model a class after a concept in our program, we *encapsulate* the logic and data for that concept together.

- This is one of the 4 so-called “Pillars of Object-Oriented Programming”

- Let’s look at our Rectangle class from last classes’ exercise and see how we *encapsulate* the idea of a Rectangle in our code.

- See rectangle/Rectangle.java
Abstraction

- When we encapsulate data and methods together in a class, we endeavor to *hide* the gory details about how we implemented our class.

- The reason we do this is so that users of our class can use our Rectangle and not have to understand how its implemented. (Important when working on teams)
  - Example: users of our Rectangle do not care how `draw()` is implemented. But they can invoke the methods and utilize our class.
  - Another example: do you care about how the String class implements `charAt(..)`?

- This idea is known as *abstraction*. 
In order to encapsulate our classes and abstract their operation, we often need to *hide our data* from users of our class.

As good citizens we try to protect users of our classes from themselves.

Think of our Rectangle example, does a negative width or height make sense? Yet given our current implementation, there is nothing stopping a user from creating an object with such dimensions.

We have to hide our data from the world and guard against invalid inputs.

This is accomplished with *access modifiers* and *getter/setter methods*. 

**Data Hiding**
Access Modifiers

- There are four modifiers that can be used with data fields / methods in a class definitions..

- Only one of these can be specified at a time, but the modifiers can be different for different data fields / methods in the same class.

  - **public** - any other class in the world can access the data field / method
  
  - **package** (default, when no other modifier is specified) - any other class in the same package can access the data field / method
  
  - **protected** -(to be discussed in chapter 11)
  
  - **private** - data field / method only accessible in the same class

- See *AccessModifiers.java*
Getters & Setters

- If we make our data fields private, then the values of those fields can not be modified by a user of the class.

- Sometimes it makes good sense to allow a user of your class to modify the values of the data fields.

- How do we hide data and allow modification of object values?

- Getters & setters (sometimes called accessors & mutators)

- See GettersSetters.java
Programming Exercise

- Update the rectangle class to hide its data properly.
- Make weight and height private and add getters/setters.
Static
Static Variables

- When a number of objects are created from the same class blueprint, they each have their own distinct copies of instance variables.

- In the case of the SimpsonsCharacter, the instance variables are name, occupation, etc.

- Each SimpsonsCharacter object has its own values for these variables, stored in different memory locations.

- *Sometimes, you want to have variables that are common to all objects.*

- This is accomplished with the `static` modifier.
Static Variables *con’t*

- For example, suppose you want to have a hometown property of the SimpsonsCharacter class.

- Obviously this is “Springfield”

- This value should be shared between *all objects of the SimpsonsCharacter class*.

- A common use for static methods is to access static fields.
Static Variables con’t

```java
public class SimpsonsCharacter {

    private static String hometown = "Springfield";

    private String name;
    private String catchPhrase;

    public SimpsonsCharacter(String name, String catchPhrase) {
        this.name = name;
        this.catchPhrase = catchPhrase;
    }

    public void speak() {
        System.out.printf(
            "Ny name is%%s, I'm from %%s",
            name,
            SimpsonsCharacter.hometown
        );
    }
}
```
Static Variables con’t

- Class variables are referenced by the class name itself, as in..

  `SimpsonsCharacter.hometown`

- …this makes it clear that they are class variables.

- Note: If the field is public (discouraged unless final) you can also refer to static fields with an object reference like..

  `bart.hometown;`

- …but this is discouraged because it does not make it clear that they are class variables.

- See `StaticVariables.java`
Static Methods

- The Java programming language supports static methods as well as static variables.

- A static method is called on a class, rather than a specific object.

- Static have the `static` modifier in their definition.

- Static methods should be invoked with the class name, without the need for creating an instance of the class.

  - Example: `Math.random();`
public class SimpsonsCharacter {

    static String hometown = "Springfield";

    private String name;
    private String catchPhrase;

    public SimpsonsCharacter(String name, String catchPhrase) {
        this.name = name;
        this.catchPhrase = catchPhrase;
    }

    public static String getHometown() {
        return SimpsonsCharacter.hometown;
    }

    public void speak() {
        System.out.printf("My name is %s, I'm from %s",
                name,
                SimpsonsCharacter.hometown);
    }
}

Static Methods con’t
Static Method *con’t*

- Not all combinations of instance and static variables and methods are allowed:
  - Instance methods can access instance variables and instance methods directly.
  - Instance methods can access class variables and class methods directly.
  - Class methods can access class variables and class methods directly.
  - Class methods cannot access instance variables or instance methods directly. Also, class methods cannot use the this keyword as there is no instance for this to refer to.

- See *StaticMethodsAndVariables.java*