Modifiers and Inheritance

- Review of Previous Week
- Access modifiers
- Final keyword
- Inheritance
- Overriding

Review of Previous Week

- `this` and `null` keywords
- Class and method declarations
- Exiting methods
- Encapsulation

Review (Class Declaration)

`access modifier` class `Name`

{

`class body`

}

```
public class Ball {
    private double xVelocity;
    private double yVelocity;

    public double getyVelocity() {
        return yVelocity;
    }

    public double getxVelocity() {
        return xVelocity;
    }
}
```

Review (exiting methods)

3 Ways to exit a method:

- the end of the method is reached (for void methods)
- encounter a return statement (for non-void methods)
- exception thrown (more on those later)

Access Modifiers

Can be applied to classes, variables and methods:

- `public` : Accessible to all, everywhere
- `protected` : Accessible via inheritence and the same package
- `default` : Accessible to the subclasses
- `private` : Accessible only to the same class

```java
public String name = "Sam Jones";
private String SSnum = "555-55-5555";
protected String address = "500E 200N, Salt Lake City";
String phoneNum = "(301)-254-3320"; //default
```

Access Modifier Permissions

```
           | Class | Package | Subclass | Subclass | Worl
           |       |         |(same pkg)|(diff pkg)|
-----------+-------+---------+----------+----------+-----
public     |   +   |    +    |    +     |     +    |  +
-----------+-------+---------+----------+----------+-----
protected  |   +   |    +    |    +     |     +    |
-----------+-------+---------+----------+----------+-----
no modifier|   +   |    +    |    +     |          |
-----------+-------+---------+----------+----------+-----
private    |   +   |         |          |          |
+ : accessible
blank : not accessible
```

Access Modifiers Good Practice

- Only `public` if another class needs it
- Make all variables `private` and access them via `public` getters/setters
- `protected` when only related classes should be able to access

Access Modifier Example

```java
public class BankCustomer{
  //All data members (variables) are made private
  private String name = "Sam Jones";
  private double balance = 502.38;
  private String SSnum = "555-55-5555";
  //Only related classes can see the balance
  protected double getBalance(){
    return balance;
  }
  //Private, only the base class should be able to access
  private String getSSnum(){
    return SSnum;
  //Public, everyone should be able to see their name at
  public String getName(){
    return name;
  }
}
```

Final keyword (Overview)

Has 3 uses:

1. Variables-- Most common, equilivent to `const` in other languages
2. Methods --Cannot be overriden (more on that in a bit...)
3. Classes --Has no inheritance/sub-classes

Final keyword example (Variable)

When you don't want a value to change.

```
// the value of pi will never change
final double PI = 3.141592653589793;
// the following line results in a compiler error!!
PI = 3;
```

```
// declared but not initialized
final int number;
// initilized at some later point, this IS valid.
int number = 40;
```

Final keyword example (classes)

```java
// Class marked final, this means no inheritence
public final class Person
{
  //final variables: can't be changed once declared.
  private final String name;
  private int age;
  Person(String name, int age){
    this.name = name;
    this.age = age;
  }
  public getName(){
    return name;
  }
  public getAge(){
    return age;
  }
  public setAge(int newValue)
    age = newValue;
}
```

Final keyword example (classes)

```java
class Child extends Person{
  //COMPILE ERROR, cannot extend a final class
}
```

Final keyword example (methods)

(See Person and Student example code in module)

Inheritance

What does "inheritance" mean?

Classes can *derive* information from other classes.

We say a child class *inherits from* a parent class, or that it *extends* the parent class.

We use inheritance in code when we want a generic version of an idea, along with more specific versions of the idea.

Examples

- A *dog* is a more specific kind of *animal*
- A *Toyota Camry* is a more specific kind of *car*, which is a more specific kind of *vehicle*
- A *video game* is a more specific kind of *program*
- A *Java programmer* is a more specific kind of *person*!

Why do we want that in code?

- Lets us organize our code better, so we can understand it more easily
- Lets us reuse code, so we can write *less code* to do the same thing

The grammar

```
   public    class   Animal {
// ^         ^       ^
// Access    Class   Name
// Modifier

   }

   public    class   Dog    extends  Animal {
// ^         ^       ^      ^        ^
// Access    Class   Name   Extends  Parent
// Modifier                          Class

   }
```

We can have more levels if we want!

```java
public class Vehicle {
  // ...
}

public class Car extends Vehicle {
  // ...
}

public class ToyotaCamry extends Car {
  // ...
}
```

What happens when we use inheritance?

The subclass inherits the functionality of the parent class, and then adds on to it.

```java
public class Animal {
  public void walk() {
    System.out.println("The animal is walking");
  }
}

public class Dog extends Animal {
  public void bark() {
    System.out.println("Bark!");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog spot = new Dog();
    spot.walk(); // Defined in the Animal class
    spot.bark(); // Defined in the Dog class
  }
}
```

The Object class

The very most basic class in Java

If your class does not say `extends`, then it still extends Object!

```java
public class Object {
  // ...
}
```

The animals example

- `Object`
- `Animal` implicitly extends `Object`
- `Dog` explicitly extends `Animal`

You could write this:

```java
public class Animal extends Object {
  // ...
}
```

But you don't need to, because Java does it for you. The result is the same.

Overriding

The animal example

```java
public class Animal {
  public void walk() {
    System.out.println("The animal is walking");
  }
}

public class Dog extends Animal {
  public void bark() {
    System.out.println("Bark!");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog spot = new Dog();
    spot.walk(); // "The animal is walking"
    spot.bark(); // "Bark!"
  }
}
```

Overriding the `walk` method

```java
public class Dog extends Animal {

  @Override
  public void walk() {
    System.out.println("The dog is walking");
  }

  public void bark() {
    System.out.println("Bark!");
  }
}
```

The `@Override` annotation

This reminds everybody that we are overriding the original `walk` method with a different one.

Not required, but helpful.

```
@Override // <--------
public void walk() {
  System.out.println("The dog is walking");
}
```

What will be the output now?

```
public class Animal {
  public void walk() {
    System.out.println("The animal is walking");
  }
}

public class Dog extends Animal {
  @Override
  public void walk() {
    System.out.println("The dog is walking");
  }
  public void bark() {
    System.out.println("Bark!");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog spot = new Dog();
    spot.walk();
    spot.bark();
  }
}
```

We can add `final` to a method to keep it from being overridden

```java
public class Animal {
  public final void walk() { // Now the Dog class cannot
    System.out.println("The animal is walking");
  }
}
```

Videos for next week

- Overloading
- Using Final and Abstract - part of the videos on Inheritance

Additional Resources

- Encapsulation and Access Modifiers
- Applying Access Modifiers
- Inheritance