Week 3 - Monday

COMP 2000

Last time

- What did we talk about last time?
- Defining interfaces
- Extending interfaces

Questions?

Project 1

Concept of Inheritance

Inheritance

- The idea of inheritance is to take one class and generate a child class
- This child class has everything that the parent class has (members and methods)
- But you can also add more functionality to the child
- The child can be considered to be a **specialized** version of the parent

Code reuse

- The key idea behind inheritance is safe code reuse
- You can use old code that was designed to, say, sort lists of **`Vehicle`** objects, and apply that code to lists of **`Car`** objects
- All that you have to do is make sure that **`Car`** is a subclass (or child class) of **`Vehicle`**

Subclass relationship

- Java respects the subclass relationship
- If you have a **`Vehicle`** reference, you can store a **`Car`** object in that reference
- A subclass (in this case a **`Car`**) is a more specific version of the superclass (**`Vehicle`**)
- For this reason, you can use a **`Car`** anywhere you can use a **`Vehicle`**
- You **`cannot`** use a **`Vehicle`** anywhere you would use a **`Car`**

Subclass example

- As long as **`Car`** is a subclass of **`Vehicle`**, we can store a **`Car`** in a **`Vehicle`** reference

```
Vehicle v = new Car("Lancer Evolution"); // okay
```

- Even in an array is fine

```
Vehicle[] vehicles = new Vehicle[100];
for( int i = 0; i < vehicles.length; i++ )
  vehicles[i] = new RocketShip();  // cool
```

- Storing a **`Vehicle`** into a **`Car`** doesn't work

```
Car c = new Vehicle(); // gives error
```

Inheritance Mechanics

Creating a subclass

- All this is well and good, but how do you actually create a subclass?
- Let's start by writing the **`Vehicle`** class

```
public class Vehicle {
  public void travel(String destination) {
     System.out.println("Traveling to " +
     destination);
  }
}
```

Extending a superclass

- We use the **`extends`** keyword to create a subclass from a superclass

```
public class Car extends Vehicle {
  private String model;
  public Car(String s) { model = s; }

  public String getModel() { return model; }

  public void startEngine() {
     System.out.println("Vroooooom!");
  }
}
```

- A **`Car`** can do everything that a **`Vehicle`** can, plus more

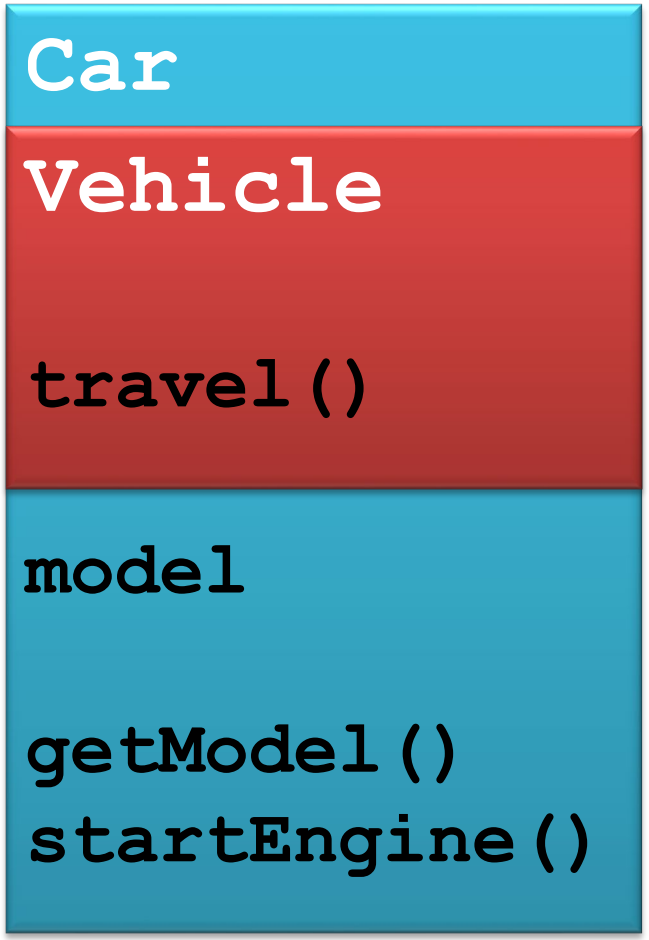
Power of inheritance

- There is a part of the **`Car`** class that knows all the **`Vehicle`** members and methods

```
Car car = new Car("Camry");
// Prints "Camry"
System.out.println( car.getModel() );
// Prints "Vroooooom!"
car.startEngine();
// Prints "Traveling to New York City"
car.travel( "New York City" );
```

A look at a Car

- Each **`Car`** object actually has a **`Vehicle`** object buried inside of it
- If code tries to call a method that isn't found in the **`Car`** class, it will look deeper and see if it is in the **`Vehicle`** class
- The outermost method will always be called



Constructors

Constructors

- A child class has to create a version of the parent class "inside" itself
- Consequently, the first line of a child class constructor is reserved for a call to the parent constructor
- If the parent has a default constructor (with no arguments), no call is necessary
- Otherwise, a call to the parent constructor **must** be made by using the keyword **`super`**, followed by parentheses and the arguments passed to the parent constructor

Food class

- Here's a simple **Food** class we'll use for some constructor examples
- Since **Food** doesn't have a default constructor, any children must call its constructor that takes a **String** followed by an **int**

```
public class Food {
  private String name;
  protected int calories;

  public Food(String name, int calories) {
     this.name = name;
     this.calories = calories;
  }
}
```

FoieGras class

- The **FoieGras** class extends **Food** and consequently must call the **Food** constructor as the first thing in its constructor
- The **FoieGras** constructor can be completely different from the **Food** constructor as long as it calls the **Food** constructor correctly

```
public class FoieGras extends Food {
  private int grams;

  public FoieGras(int grams) {
     super("Foie Gras", 462*grams/100);
     this.grams = grams;
  }
}
```

Using `this`

- In addition to using **`super()`** to call a parent constructor, one constructor in a class could use **`this()`** to call another constructor in the same class to set up the object
- The chain of constructor calls must end with a constructor that calls the parent constructor

```
public class FoieGras extends Food {
  private int grams;

  public FoieGras() { // Default constructor assumes 180 grams
     this(180);
  }
  public FoieGras(int grams) {
     super("Foie Gras", 462*grams/100);
     this.grams = grams;
  }
}
```

`protected` keyword

- In addition to public and private modifiers, the **`protected`** keyword is meaningful in the context of inheritance
  - Methods and members that are **`public`** can be accessed by any code
  - Methods and members that are **`private`** can only be accessed by methods from the same class
  - Methods and members that are **`protected`** can be accessed by code in the same package and by methods of any classes that inherit from the class
- Hard-core OOP people dislike the **`protected`** keyword since it allows child classes to fiddle with stuff that they probably shouldn't

Using protected

- The **Milk** class can change the **calories** field because it's **protected**

```
public class Milk extends Food {
  private boolean isSkim = false;
  public Milk(int cups) {
     super("Milk", 148*cups);
  }
  public void skimFat() {
     if(!isSkim) {
          calories *= 0.56;
          isSkim = true;
     }
  }
}
```

Object class

Object class

- The **Object** class is the parent of all reference types
- You can store any reference in an **Object** reference

```
Object object1 = "Goats";
Object object2 = new Wombat();
Object object3 = new double[100];
```

- Although it's convenient to be able to put anything in an **Object**, you can't do much with it unless you cast it back to something
- **Object** is the only class that doesn't have a parent

`Object` methods

- If you don't explicitly state which class your class extends, it extends **`Object`**
- Because everything inherits (directly or indirectly) from **`Object`**, there are some methods that every object of every class has:
  - **`clone()`**
  - **`equals(Object other)`**
  - **`finalize()`**
  - **`getClass()`**
  - **`hashCode()`**
  - **`notify()`**
  - **`toString()`**
  - **`wait()`**
  - **`wait(long timeout)`**
  - **`wait(long timeout, int nanoseconds)`**

Important `Object` methods

- Some **`Object`** methods come up frequently:

Return type	Method	Use
`boolean`	`equals(Object other)`	Tests if two objects are the same, should be overridden by classes to be meaningful
`Class<?>`	`getClass()`	Returns an object representing the class of the object
`int`	`hashCode()`	Returns a hash value for the object, useful for hash tables, should be overridden by classes to be meaningful
`String`	`toString()`	Returns a **`String`** representation of the object, should be overridden by classes to be meaningful

Even primitives…sort of

- You can even store a primitive value into an object reference
- But it will use a feature called **automatic boxing**

```
Object number = 7;
```

- In other words, the primitive type is boxed into an appropriate wrapper class
- In this case, an **`Integer`** object is created that contains 7
- There are situations where we have to box primitive types into reference types, but doing so is inefficient

Inheritance Examples

The `Person` class

- We can imagine a hierarchy of inheritance starting with a **`Person`** with the following members:
  - Name (final)
  - Age
- **`Student`** extends **`Person`** and adds:
  - Major
  - GPA
- **`Politician`** extends **`Person`** and adds:
  - Political party
- **`OtterbeinStudent`** extends **`Student`** and adds:
  - ID number (final)
- Members should have getters and setters as appropriate
- All classes should override the **`toString()`** and **`equals()`** methods

Upcoming

Next time…

- Lab 3 is tomorrow
- On Wednesday, we'll talk about overriding methods and polymorphism

Reminders

- Read Chapter 17
- Keep working on Project 1