Lecture 6

Interfaces

Pet-able!
The Weak Spot Under the Hood

• “New high-end cars are among the most sophisticated machines on the planet, containing 100 million or more lines of code. Compare that with about 60 million lines of code in all of Facebook or 50 million in the Large Hadron Collider.”

• “The increasing reliance on code raises questions about how these hybrids of digital and mechanical engineering are being regulated. Even officials at the National Highway Traffic Safety Administration acknowledge that the agency doesn’t have the capacity to scrutinize the millions of lines of code that now control automobiles.”

• Car companies hiring Cybersecurity experts, beefing up their programming staffs
Inheritance - Review (Part I)

- Review of Inheritance
  - a method of modeling similar capabilities among similar objects
- Abstract methods and classes
- Interfaces
  - method of modeling similar capabilities among dissimilar objects
- Interface Examples
- Classes vs Interfaces
Inheritance - Review (Part I)

- Last lecture we covered inheritance
  - specialized class (subclass) inherits properties and capabilities from more general class (superclass)

- Subclass can override or partially override methods inherited from its superclass, allowing the subclass to behave in a more specialized manner
Inheritance - Review (Part II)

- Here’s the *inheritance hierarchy* we saw last class

- **BrownCSStudent** is a subclass of **BrownStudent** and the superclass of **CS15Student**

- Let’s review how these classes are implemented
Inheritance - Review (Part III)

- The BrownStudent class implements the method study

- When told to study, a BrownStudent will go to the SciLi, open a book, read, check Facebook, and then take a nap

```java
public class BrownStudent {
    /* Constructor elided */

    public void study() {
        this.goToSciLi();
        this.openBook();
        this.read();
        this.checkFacebook();
        this.takeNap();
    }
}
```
Modeling Similarity

- Subclasses can respond to same message in different ways
  - Sportscar, CS15Mobile, and Van all responded to move() slightly differently
- These three classes have a lot in common
- Makes sense to model this similarity with Vehicle inheritance hierarchy
Modeling Similarity

- What if a group of classes has very little in common, except for the ability to respond to a particular message?
- *Vehicle, Animal, Ball* might all have a *move* method
- They have almost nothing else in common, except for being *Objects*
- Doesn’t make sense to model this with inheritance hierarchy
Introducing… Interfaces!

- **Interfaces** model similar capabilities of classes that might otherwise be very different

- Interfaces models “act as” relationships while inheritance models “is a”

- *Vehicle*, *Animal*, and *Ball* all could implement a *Mover* interface despite being different types, because they can all “act as” *Movers*: objects that have the capability to move

- They could implement *Move* in their own way without using *Mover* interface, but implementing it induces consistency, and lets compiler check that they’ve done it consistently
Interfaces

- An interface’s only role is to **declare methods** that all implementing classes must have.
- It cannot define any of these methods.
- Every class that implements a particular interface must define all methods declared in the interface (if it’s an abstract class, may leave definition to concrete subclasses).
  - Interface acts as a **contract** that Java must enforce.
  - If it’s an abstract class, you may declare the method abstract and leave definition to concrete subclasses.
- An interface is similar to an abstract superclass, but even *more* abstract—every single method is abstract.
- Additionally, interfaces *only* define sets of methods – they cannot have properties like abstract classes.
Why Interfaces?

- A class may only extend *one* superclass, but it may implement an unlimited number of interfaces!
- A class may extend a superclass and also implement any number of interfaces
- Basically, interfaces are super flexible and general!
- No code sharing - purely a contract that Java can enforce
- See their real power in next lecture on *polymorphism*…
Interface Example

public interface Mover {
    public void move();
}

● Here’s the Mover interface!
● (Yep, that’s the whole thing)
● An interface belongs in its own .java file, just as a class does. The Mover interface would be in the file Mover.java
Interface Example

```java
public interface Mover {
    public void move();
}
```

- Notice we declare it as an `interface` rather than a `class`
Interface Example

```java
public interface Mover {
    public void move();
}
```

- Here, we declare the methods that all implementing classes must define - the **contract**
- In this case, only one method is required: `move`
- An interface’s methods are by definition **abstract**, so we can leave out that keyword
Interface Example

public interface Mover {
    public void move();
}

- Interfaces are not allowed to define any methods! Can only declare them
- That’s why we end method declaration with semicolon rather than curly braces
Implementing an Interface: Concrete Class

public class Ball {
    public Ball() {
        // this is the constructor
    }

    public void bounce() {
        // code for ball to bounce
    }
}

- Let’s modify concrete class Ball so that it implements the Mover interface
- Right now, Ball only has one method: bounce
Implementing an Interface: Concrete Class

public class Ball implements Mover {
    public Ball() {
        // this is the constructor
    }

    public void bounce() {
        // code for ball to bounce
    }
}

- First, add “implements Mover” to class declaration
- This promises the compiler that Ball will define all methods in the Mover interface: in this case, the move method
Implementing an Interface: Concrete Class

```java
public class Ball implements Mover {
    public Ball() {
        // this is the constructor
    }

    public void bounce() {
        // code for ball to bounce
    }

    public void move() {
        // code for ball to move
    }
}
```

- Next, honor the contract by defining `move` method
- **Method signature** (name and number/type of arguments) must match method signature declared in interface
- If `Mover` had multiple methods, `Ball` would have to implement all of them
Implementing an Interface

```java
public class Ball implements Mover {
    public void move() {
        this.bounce();
    }
    // other methods and constructor elided
}

public class Animal implements Mover {
    public void move() {
        this.walk();
    }
    // other methods and constructor elided
}

public abstract class Vehicle implements Mover {
    public void move() {
        this.drive();
    }
    // other methods and constructor elided
}
```

- Again, any object that “acts as” a Mover can implement the Mover interface
- Balls, Animals, and Vehicles all move – but they do so in different ways
- Notice that Vehicle is an abstract class – we’ll get to this on the next slide!
Implementing an Interface: Abstract Class

```java
public abstract class Vehicle {
    private Human _driver;

    public Vehicle(Human driver) {
        _driver = driver;
    }

    public abstract void move();

    /* Other methods elided */
}
```

- Let's look at abstract superclass `Vehicle` and its subclasses
- Now we’ll modify this example so `Vehicle` implements the `Mover` interface
Implementing an Interface: Abstract Class

public abstract class Vehicle implements Mover {
   private Human _driver;
   public Vehicle(Human driver) {
      _driver = driver;
   }
   public abstract void move();
   /* Other methods elided */
}

- First, we add “implements Mover” to class declaration
- Now we’ve promised that all Vehicles will know how to move
- This means that everything that “is a” Vehicle (all subclasses of Vehicle) must know how to move
Implementing an Interface: Abstract Class

```java
public abstract class Vehicle implements Mover {
    private Human _driver;

    public Vehicle(Human driver) {
        _driver = driver;
    }

    public abstract void move();

    /* Other methods elided */
}
```

- Notice the previously declared abstract `move` method that made sure all concrete subclasses implemented it.
- Now that `Vehicle` implements `Mover`, this declaration is no longer necessary--all subclasses now must also implement `Mover` by default.
Implementing an Interface: Abstract Class

public abstract class Vehicle implements Mover {

    private Human _driver;

    public Vehicle(Human driver) {
        _driver = driver;
    }

    /* Other methods elided */
}

- Compiler wouldn’t yell at us for leaving move declaration in there, but we’ll take it out since it’s now redundant, given the Mover interface
Implementing an Interface: Concrete Subclass

public class Sportscar extends Vehicle {

   /* Other methods elided */

   @Override
   public void move() {
      // code to move super fast!
   }
}

- Concrete subclass Sportscar must define move method!
- Don’t have to specify in class declaration that Sportscar “implements Mover”—this is implicit since we’ve already declared that superclass Vehicle implements Mover
Another Interface Example

```java
import javafx.scene.paint.Color;

public interface Colorable {
    public void setColor(Color color);
    public Color getColor();
}
```

- Here’s the interface `Colorable` that declares accessor and mutator methods for an object’s color.
- Again, note the `);`; instead of braces with an actual body – it’s a contract, not an implementation!

Bear in mind: other classes (such as `Vehicle`, `Furniture`, `Ball` etc.) could also implement `Colorable`!
import javafx.scene.paint.Color;

public class Ball implements Mover, Colorable {
    private Color _color;

    public Ball() {
        _color = Color.WHITE;
    }

    public void move() {
        // code for ball to move
    }

    public void setColor(Color color) {
        _color = color;
    }

    public Color getColor() {
        return _color;
    }

    // other methods elided
}

- Now we’ve modified the Ball class to implement two interfaces: Mover and Colorable
- Let’s walk through how we’ve implemented the Colorable interface and defined the capabilities it declares
import javafx.scene.paint.Color;

public class Ball implements Mover, Colorable {
    private Color _color;
    
    public Ball() {
        _color = Color.WHITE;
    }
    
    public void move() {
        // code for ball to move
    }
    
    public void setColor(Color color) {
        _color = color;
    }
    
    public Color getColor() {
        return _color;
    }
    
    // other methods elided
}

- We’ve added Colorable to the class declaration as an implemented interface
- All of a class’s implemented interfaces are declared in a comma-separated list
- If we wanted Ball to also extend a superclass, the declaration might look something like “public class Ball extends SportsEquipment implements Mover, Colorable {”
import javafx.scene.paint.Color;

public class Ball implements Mover, Colorable {

    private Color _color;

    public Ball() {
        _color = Color.WHITE;
    }

    public void move() {
        // code for ball to move
    }

    public void setColor(Color color) {
        //you define this method
    }

    public Color getColor() {
        //you define this method
    }

    // other methods elided
}

- The **Colorable** interface declares accessor and mutator methods for an instance variable
- **Ball** creates an instance variable called _color and gives it an initial value in the constructor
import javafx.scene.paint.Color;

public class Ball implements Mover, Colorable {

    private Color _color;

    public Ball() {
        _color = Color.WHITE;
    }

    public void move() {
        // code for ball to move
    }

    public void setColor(Color color) {
        _color = color;
    }

    public Color getColor() {
        return _color;
    }

    // other methods elided

    // We follow the standard pattern for accessor and mutator methods to define setColor and getColor
}
import javafx.scene.paint.Color;

public class Ball implements Mover, Colorable {
    private Color _color;
    public Ball() {
        _color = Color.WHITE;
    }
    public void move() {
        // code for ball to move
    }
    public void setColor(Color color) {
        _color = color;
    }
    // oops! We forgot to define getColor!
}

● What happens if we forget to define a method declared by an implemented interface?

● We get a compiler error, because we didn’t fulfill the contract laid out by the interface!
Interfaces & Inheritance

● Like classes, interfaces can extend other interfaces...
● But interfaces can extend any number of other interfaces
  ○ This is because interfaces never specify implementation
  ○ To extend multiple interfaces, just provide interfaces as a comma-separated list after keyword “extends”: for example, “public interface Artistic extends Colorable, Decorable”
“Performer” Interface Example

- What if we defined an interface for being a Performer? After all, singers and dancers are different but both groups can perform.
- Can you think of any more methods that might go in this interface?

```java
public interface Performer {
    public void perform();
}
```
Classes vs. Interfaces

Class:
- Models an object with properties and capabilities
- Factors out common properties and capabilities of similar objects
- Declares methods and may define some or all of them
- Can extend only one superclass (but can implement any number of interfaces)
- Allows code reuse for subclasses’ method implementations

Interface:
- Models a role; defines a set of responsibilities
- Factors out common capabilities, not properties, of (often) dissimilar objects
- Declares but does not define methods
- Can extend multiple interfaces
- Does not provide code for method implementation
Interfaces: Summary

- Interfaces factor out common capabilities from otherwise unrelated objects. They model an “acts as” relationship.
- An interface defines a contractual obligation:
  - implementing classes must provide definitions for all methods declared in interface
  - Java compiler verifies this at compile-time
- A class can implement as many interfaces as it wants, and an interface can extend as many interfaces as it wants.
- The core Java library uses interfaces a lot, especially for graphics.
Conflicting Methods in Two Interfaces

What if a class implements two interfaces that contain a method of the same name?

- If they have same exact signature, i.e., not only identical name but also parameter list, they should specify same behavior, so only need to define method once

- If they have different signatures, should specify different behaviors, and must define each method separately

- Note: return types are not part of a method’s signature—”The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.” <JavaDocs>
Why Use Interfaces?

- Interfaces are a way to provide a ‘checklist’ of methods that a class should implement--this makes it easier for the compiler to check for errors

- This isn’t that strong a motivation: different classes (like Ball and Vehicle) could all just declare separate move methods on their own without implementing an interface

- The real motivation for interfaces is **polymorphism**; stay tuned for next lecture
A Note about Generality and Factoring

● Use the power of abstraction to provide generality. This will make your code more extensible as your programs get bigger.

● **Classes** factor out commonality between instances

● **Inheritance** factors commonality between classes

● **Abstract methods** factor out commonality of inherited behaviors without providing specifics of implementation

● **Interfaces** factor out specification of methods, without any implementation – they are abstract classes taken to the limit
Announcements

• Working From Home Help Session 10/4 at 8pm in Salomon 001

• Make sure to sign up for CS15 TA Hours, not CS17 on SignMeUp

• LiteBrite early handin is tonight at 11:59PM, on-time handin is on Thursday