Interfaces

Lecture 10

Object-Oriented Programming

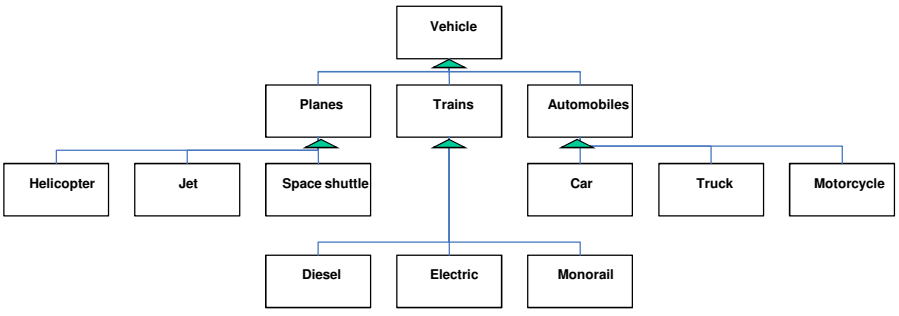
Agenda

- What do we have in our programming Toolbox?
- Design Problem
- Diamond Problem
- Interfaces
- Interface Naming
- Implementing Interfaces
- Interfaces and Inheritance
- Multiple Interfaces
- Interface Intricacies
- Interface vs. Abstract Class
- Abstraction

What do we have in our programming Toolbox

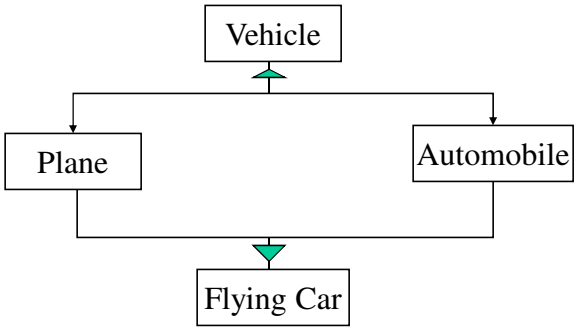
- Objects
- Classes
- Composition
- Arrays
- Iteration
- Recursion
- Inheritance
  - Abstract Classes

Design Problem

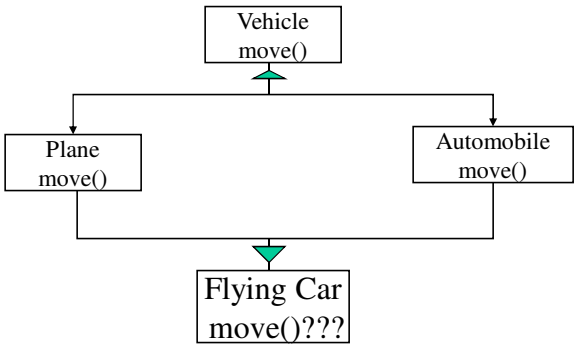


Where are we going to fit a Flying Car

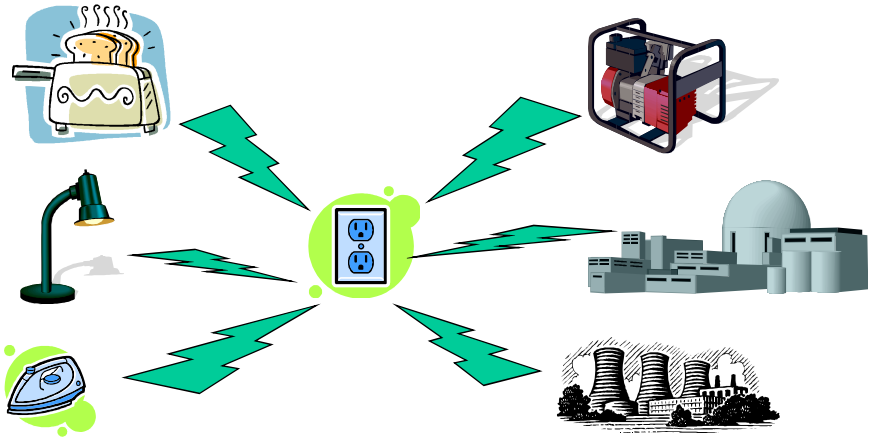
Solution 1 - Multiple Inheritance



Diamond Problem



Interfaces



Interfaces

- Much like two objects are related if they *inherit from* the same superclass, objects can be related if they *implement* the same interface
- Superclasses "factor out" responsibilities among similar classes that model an "is-a" hierarchy
- Interfaces are mainly used to "factor out" responsibilities among *very different* classes whose only commonality are specific shared capabilities
  - models the "acts-as" relationship

Interfaces

- Interfaces only declare capabilities that object
  - must have — there are *no* definitions
  - no code! (except declarations)
- Interfaces are similar to abstract classes in that
  - *all* methods are abstract, except that they have:
  - no constructor
  - usually no instance variables
  - list of responsibilities (public methods) and
  - *nothing else*
- Interfaces prescribe very generic *roles*
  - professor, Head TAs and TAs all "teach" as part of
  - their responsibility but they come from different class hierarchies and teach in different ways
  - another role they share is "UETIANs"

Interface Naming

- Interface names are often adjectives, ending in
- *-able* or *-ive*.
  - examples: **Colorable**, **Rotatable**
- Interface names that specify roles often end in -er
  - examples: **Container**, **Mover**, **Teacher**
- What might a **Colorable** interface look like?

Code Example

```
package wheels; // Short for training wheels

/**
 * This interface models something
 * with a changeable color. It
 * specifies that all classes
 * implementing it must allow their
 * color to be set and accessed.
 */

public interface Colorable {


  // set the color of the implementing object
  public void setColor( java.awt.Color c );


  // get the color of the implementing object
  public java.awt.Color getColor();
}
```


Code Example

- **`public interface Colorable {`**
  - declaration for an interface
  - same as for class, except “interface” instead of
  - “class”
- **`public void setColor( java.awt.Color c );`**
  - specifies that anything that acts as a **Colorable** must
  - have the capability to have its color set
- **`public java.awt.Color getColor();`**
  - specifies that anything that acts as a **Colorable** must
  - have the capability to give access to its color

Code Explanation

- Note the simplicity of methods
  - **Colorable** interface only specifies accessor and
  - mutator methods for **Color**
  - methods can only be **abstract**, so that keyword
  - can be left off
  - methods can only be **public**, but we leave that
  - keyword in for the sake of clarity
- That's it!
  - interfaces are really simple!
  - in fact, this is almost verbatim the code in the *actual* **wheels.Colorable** interface
- Note that there is no executable code and hence
  - no code sharing with interfaces - it is purely a "contractual" mechanism that puts all
  - implementation responsibilities on the
  - implementors

Implementing Interfaces

- Classes can extend only *one* other class
- Classes can implement *any number* of interfaces
- Classes can extend a superclass *and* implement interfaces

Implementing Interfaces

- The **Car**, for example, could implement interfaces which categorize objects that are able to move, hold passengers, be driven, be repaired, etc.
- Interfaces thus create the fourth mechanism for factoring that is even more general and flexible than others:
  - classes and instances
  - methods and parameters
  - superclasses and subclasses
  - interfaces and implementations

Implementing Interfaces Code Example

```
package Demos.Car;

/**
 * This class models a CS15Mobile
 * that implements the Colorable
 * interface.
 */

public class OOPCar extends Car implements  Colorable {

  private java.awt.Color _color;

  // other instance variables

  // constructor

  public void setColor( java.awt.Color c ) {
    _color = c;
  }

  public java.awt.Color getColor() {
    return _color;
  }
  // other methods
}
```

Code Explained

- `public class OOPCar extends Car implements Colorable {`
  - **OOPCar extends Car** but also **implements Colorable**
  - implementing an interface is as simple as using the word **implements**
  - to implement multiple interfaces, just separate them by a comma
    - **e.g., implements Colorable, Locatable, Mover**
- **OOPCar** defines the **Colorable** interface methods by making them simple accessors and mutators
  - could also do something more complicated
  - Java doesn't care *how* you define the methods of an interface, just that you *do* define them

Code Explained

- What happens if you don't define a method of an interface you implement?
  - you get a compiler error – the same thing that happens if you don't define an abstract method of a superclass
  - class must be declared **abstract**

Interfaces and Inheritance

- Like classes, interfaces can *extend* other interfaces
- Unlike classes, interfaces can extend *any number* of other interfaces
  - this is because interfaces merely declare policy — they never specify any implementation
  - just put a comma between interface names after **extends**

Multiple Interfaces

- Extending multiple interfaces is useful for objects that have some things in common but otherwise behave very differently
  - example: GUI components (e.g., **PushButton**, **TextBox**, **Menu**)
  - they all behave and react very differently
  - but they all have the capability to be located on the screen and sized
  - so a **Component** interface is created that extends both **Locatable** and **Sizeable**
- Remember: objects inherit all capabilities from their superclasses, so an object inherits all interfaces from its superclass
  - therefore if a superclass implements an interface, the subclass implements it too!

Code Example

Implementing multiple interfaces is easy
Just implement the union of all their methods!

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

public interface Shaker {
  public void shake();
}

public class Politician implements Mover, Shaker {

    //constructor

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

    public void shake() {
      //code to shake
    }
}
```


Interface Intricacies

- What if we have a method declared in two different interfaces with the same exact signature?
  - this should happen only if you really mean them to specify the same behavior
  - thus we only need to define the method once
- What if we have methods with the same name but with different signatures (e.g., parameter lists) declared in different interfaces?
  - this should happen if different behaviors were meant, and the coincidence of the names being identical was either accidental or meant to suggest similarity in behaviors
  - either way, we must define each method appropriately

Interface Intricacies

- What if these methods have the same name and the same signature but I want them to mean different things?
  - bummer!
  - you should rename one of those methods to avoid the semantic conflict
- What if we have methods with the same signature, but different return types? (Recall: return types are not part of the signature.)
  - **ERROR!**
  - have to either not implement one of the interfaces, or change the signatures of the methods in the interfaces.

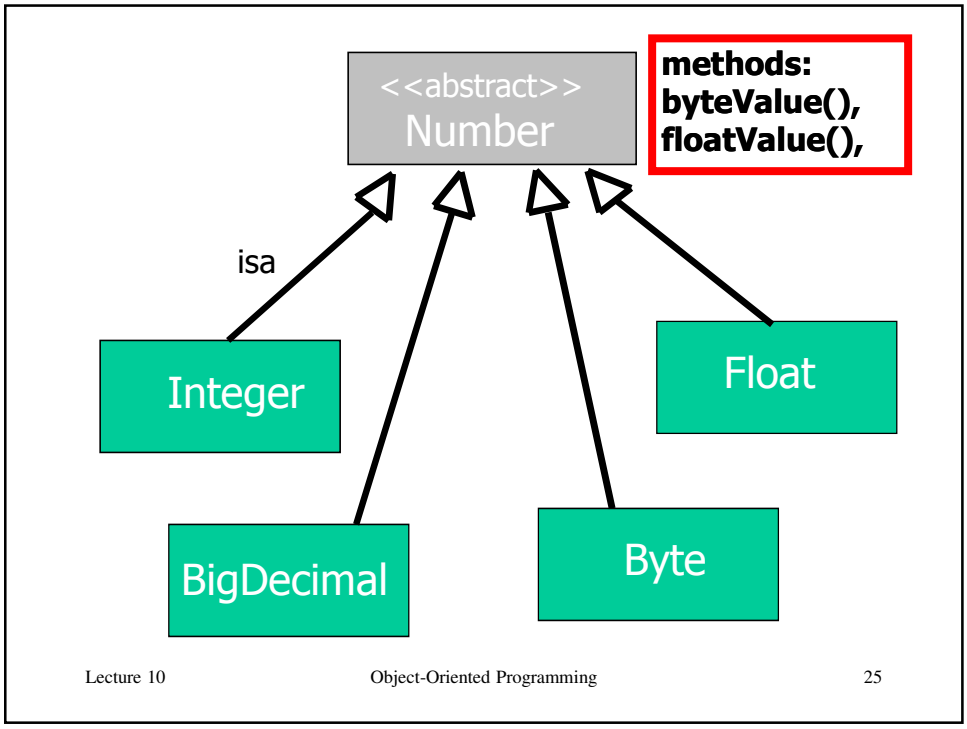
Interface vs. Abstract Class

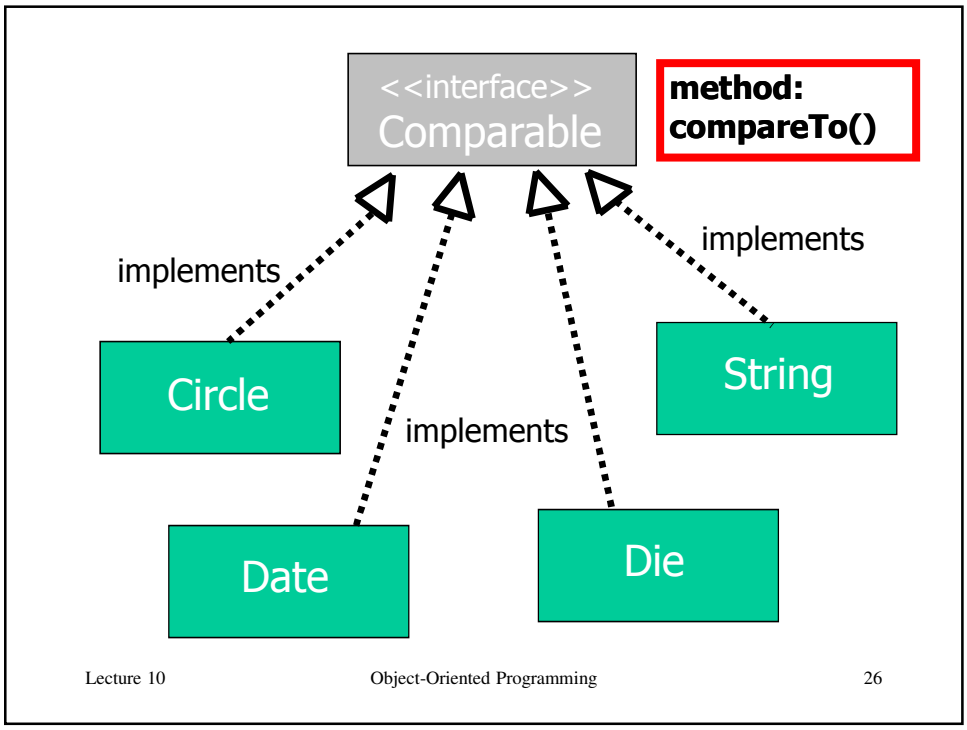
An interface is simply a list of unimplemented, and therefore abstract, methods. So how does an interface differ from an abstract class? The differences are significant.

• An interface cannot implement any methods, whereas an abstract class can.

• A class can implement many interfaces but can have only one superclass.

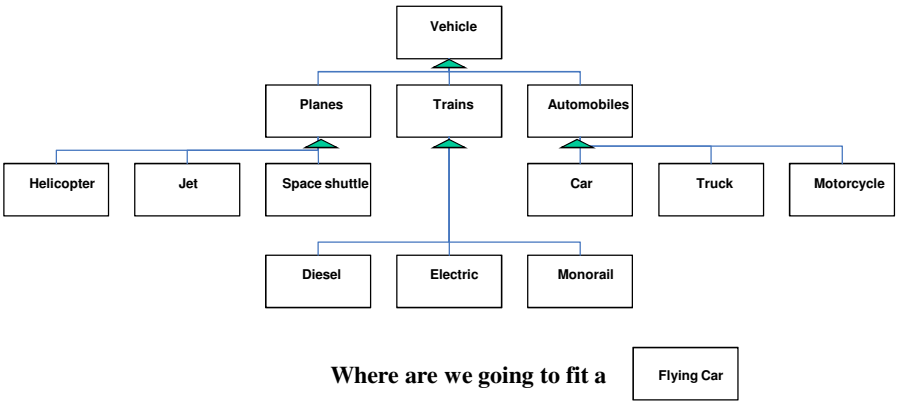
• An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.


<<abstract>>
Number
methods:
byteValue(),
floatValue(),
isa
Integer
Float
BigDecimal
Byte


<<interface>>
Comparable
method:
compareTo()
implements
implements
Circle
String
implements
Date
Die

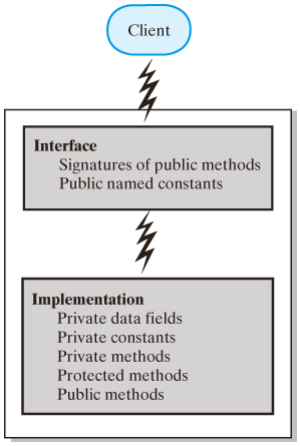
- Use an abstract class when you want a template for a collection of subclasses
- Subclass when you want to extend a class and add some functionality, whether or not the parent class is abstract
- Subclass when you must (e.g. Applets)
- Define an interface when there is no common parent class with the desired functionality, and when you want only certain unrelated classes to have that functionality
- Use interfaces as "markers" to indicate something about a class (e.g. Cloneable – can make a copy)

Design Problem



Abstraction

- Separation of **specification** from **implementation** in a class definition
  - Interface
  - Implementation
- **Interface** describe things a programmer needs to know to use the class.
- **Implementation** defines the details of how the methods are done



Readings

- **Book Name:** Head First JAVA
  Author Name: Kathy Sierra & Bert Bates
  Content: Chapter # 7 & 8