CSE 1201
Object Oriented Programming

Abstract Classes and Interfaces

Acknowledgement

- For preparing the slides I took materials from the following sources
  - Course Slides of Dr. Tagrul Dayar, Bilkent University
  - Java book “*Java Software Solutions*” by Lewis & Loftus.

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

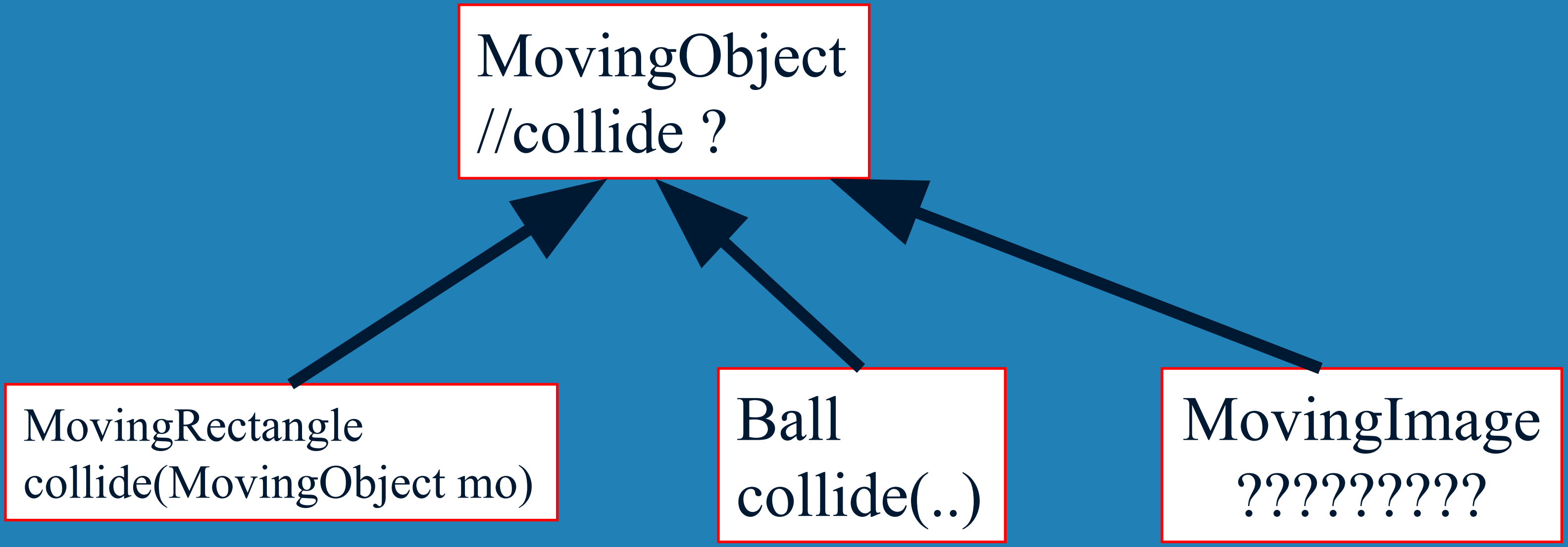
```
public abstract class Whatever
{
   // contents
}
```

Abstract Example

- A good example is the MovingObject class. MovingObject is just an abstract or synthetic concept to help us capture commonalities.
- To make sure nobody creates an instance of class MovingObject, we need to declare it abstract.

Abstract Example

- Let’s say you want the MovingObjects to be able to collide with each other, but cannot define a collide method since the outcome of collision depends on specific object



Abstract Classes

- An abstract class often contains abstract methods with no definitions
- In addition to forcing sub-classes to override to become concrete classes, it enables one to write polymorphic methods
- An abstract class typically contains non-abstract methods (with bodies), which can even call abstract methods
- A class declared as abstract does not need to contain abstract methods

Vehicle example

```
public abstract class Vehicle {
    private Position position;
    public getPosition() { return position; }
    public abstract void start();
    public abstract void move();
    public abstract void turnLeft();
    public abstract void turnRight();
    public abstract void stop();
    public void goto(Position pos) {
      start();
      if (position.getX() > pos.getX())
            turnLeft();
```

- ……

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as `final` (because it must be overridden) or `static` (because it has no definition yet)
- The use of abstract classes is a design decision – it helps us establish common elements in a class that is too general to instantiate

AbstractMethods

- to make sure every MovingObject has a collide() method, you can declare an abstract MovingObject.collide() method without an implementation, to be provided by more specific sub-classes

```
public class MovingObject {
…..
    public abstract void collide (MovingObject other);
………
}
```

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it must define all methods in the interface

Interfaces

```
public class CanDo implements Doable
                 {
         public void doThis ()
                   {
               // whatever
                   }

         public void doThat ()
                   {
               // whatever
                   }

               // etc.
                 }
```

implements is a reserved word

Each method listed in Doable is given a definition

Interfaces

- A class that implements an interface can implement other methods as well
- See Complexity.java
- See Question.java
- See MiniQuiz.java
- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants

Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements interface1, interface2
                        {
          // all methods of both interfaces
                        }
```

Interfaces

- The Java standard class library contains many helpful interfaces
- The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects
- The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order

The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
          if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

- The value returned from `compareTo` should be negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`

- When a programmer designs a class that implements the `Comparable` interface, it should follow this intent

The Comparable Interface

- It's up to the programmer to determine what makes one object less than another
- For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number
- The implementation of the method can be as straightforward or as complex as needed for the situation

Interfaces

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Interfaces are a key aspect of object-oriented design in Java

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

Polymorphism via Interfaces

- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```