1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 11: Abstract classes

References and learning objectives

- Today's sessions are mostly based on:
  - Evans, B. and Flanagan, D., 2018. ***Java in a Nutshell: A Desktop Quick Reference***, 7th edition. O'Reilly Media.
  - Deitel, H., 2018. ***Java How to Program, Early Objects****, Global Edition*, 11th Edition. Pearson.
- After today's session, you should be able to:
  - Use abstract classes and interfaces in your Java programs
  - Design appropriate class hierarchies with abstract classes and interfaces

Abstract classes

- ***Abstract classes*** are classes you cannot instantiate as objects:
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called ***abstract superclasses***.
  - Cannot be used to instantiate objects—abstract classes are *incomplete*.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

Abstract vs. concrete classes

- Classes that can be used to instantiate objects are *concrete classes*:
  - Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

Declaring abstract classes

- You make a class abstract by declaring it with keyword **`abstract`**
- An abstract class normally contains one or more *abstract methods*
  - An abstract method is defined with keyword `abstract`, e.g.:

```
public abstract void draw(); // abstract method
```

  - Abstract methods do not provide implementations
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods
- Each concrete subclass of an abstract superclass also must provide concrete implementations of the superclass's abstract methods

Example of defining an abstract class

```
1 public abstract class Shape {
2     public abstract double area();
3     public abstract double perimeter();
4 }
```

Example of defining an abstract class

```
1  public abstract class Shape {
2      public abstract double area();
3      public abstract double perimeter();
4  }
```

Abstract methods:
note semicolon ; instead of body { }

Note that public class must be in its own java file!

Example of extending an abstract class (1)

```
1  class Circle extends Shape {
2      public static final double PI = 3.14159265358979323846;
3      protected double r;
4      public Circle(double r)       { this.r = r;    }
5      public double getRadius()     { return r;      }
6      public double area()          { return PI*r*r; }
7      public double perimeter() { return 2*PI*r; }
8  }
```

Example of extending an abstract class (2)

```
1   class Rectangle extends Shape {
2       protected double w, h;
3       public Rectangle(double w, double w) {
4           this.w = w;
5           this.h = h;
6       }
7       public double getWidth()       { return w;        }
8       public double getHeight()      { return h;        }
9       public double area()           { return w*h;      }
10      public double perimeter() { return 2*(w+h); }
11  }
```

Example of testing inherited classes

```
1  class TestShape {
2      public static void main(String[] args) {
3          Shape shape;
4          shape = new Circle(5);
5          System.out.println("Area: " + shape.area());
6          shape = new Rectangle(5,10);
7          System.out.println("Area: " + shape.area());
8      }
9  }
```

```
Area: 78.53981633974483
Area: 50.0
```

Overriding abstract methods

- A subclass can override public non-static methods from its parent class
  - If the superclass contains abstract methods, a concrete subclass ***must*** override them!
- Use of **`@Override`** annotation is optional
  - However, if you don't use **`@Override`** annotation, the compiler will not check if you are really overriding an existing method

Example of method overriding (1)

```
1  abstract class Animal {
2    public abstract void
3      makeSound();
4  }
5
6  class Cat extends Animal {
7    public void makeSound() {
8      System.out.println("Meow!");
9    };
10 }
11
```

```
12 class Dog extends Animal {
13   public void makeSound() {
14     System.out.println("Woff woff!");
15   };
16 }
16 public class AnimalTest {
18   public static void main(String[] args){
19     Cat cat = new Cat(); cat.makeSound();
20     Dog dog = new Dog(); dog.makeSound();
21   }
22 }
```

```
$ java AnimalTest
Meow!
Woff woff!
```

Example of method overriding (2)

```
1  abstract class Animal {
2    public abstract void
3      makeSound();
4  }
5
6  class Cat extends Animal {
7    public void makeNoise() {
8      System.out.println("Meow!");
9    };
10 }
11
```

```
12 class Dog extends Animal {
13   public void makeSound() {
14     System.out.println("Woff woff!");
15   };
16 }
16 public class AnimalTest {
18   public static void main(String[] args){
19     Cat cat = new Cat(); cat.makeSound();
20     Dog dog = new Dog(); dog.makeSound();
21   }
22 }
```

```
$ javac AnimalTest.java
error: Cat is not abstract and does not override abstract method makeSound() in Animal
```

Example of method overriding (3)

```
1  abstract class Animal {
2    public void makeSound() {
3      System.out.println(“Burp!");
4    }
5  }
6
7  class Cat extends Animal {
8    public void makeNoise() {
9      System.out.println("Meow!");
10   };
11 }
```

```
12 class Dog extends Animal {
13   public void makeSound() {
14     System.out.println("Woff woff!");
15   };
16 }
16 public class AnimalTest {
18   public static void main(String[] args){
19     Cat cat = new Cat(); cat.makeSound();
20     Dog dog = new Dog(); dog.makeSound();
21   }
22 }
```

```
$ java AnimalTest
Burp!
Woff woff!
```

Example of method overriding (4)

```
1  abstract class Animal {
2    public void makeSound() {
3      System.out.println(“Burp!");
4    }
5  }
6  class Cat extends Animal {
7    @Override
8    public void makeNoise() {
9      System.out.println("Meow!");
10   };
11 }
```

```
12 class Dog extends Animal {
13   public void makeSound() {
14     System.out.println("Woff woff!");
15   };
16 }
16 public class AnimalTest {
18   public static void main(String[] args){
19     Cat cat = new Cat(); cat.makeSound();
20     Dog dog = new Dog(); dog.makeSound();
21   }
22 }
```

```
$ javac AnimalTest.java
error: method does not override or implement a method from a supertype
```

Dynamic binding

- *Dynamic binding* or (*late binding*): e.g., Java decides which class's method to call at execution time, not at compile time
  - A superclass reference can be used to invoke only methods of the *superclass*—the *subclass* method implementations are invoked *polymorphically*
- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error
- Operator **`instanceof`** may be used to check if the object can be cast into a particular type

Example of polymorphic processing

```
1   class TestShape {
2       public static void main(String[] args) {
3           Shape[] shapes = new Shape[3];
4           shapes[0] = new Circle(3.0);
5           shapes[1] = new Rectangle(5.0,2.0);
6           shapes[2] = new Rectangle(4.0,4.0);
7           double totalArea = 0.0;
8           for(int i=0; i<shapes.length; i++)
9               totalArea += shapes[i].area();
10          System.out.println("Total area: " + totalArea);
11      }
12  }
```

```
$ java TestShape
Total area: 54.27433388230814
```

Example using instanceof

```java
class TestShapeInstanceof {
    public static void main(String[] args) {
        Shape shape;
        shape = new Circle(5);
        if (shape instanceof Rectangle) {
            System.out.println("Shape is Rectangle!");
        }
        if (shape instanceof Circle) {
            System.out.println("Shape is Circle!");
        }
    }
}
```

```
$ java TestShapeInstanceof
Shape is Circle!
```

Example of casting to a subclass

```
1   class ShrinkShape2 {
2       public static void main(String[] args) {
3           Shape shape = new Rectangle(1.0,3.0);
4           System.out.println("Original area: " + shape.area());
5           if(shape instanceof Rectangle) {
6               Rectangle rect = (Rectangle)shape;
7               double w = rect.getWidth();
8               double h = rect.getHeight();
9               shape = new Rectangle(w/2, h/2);
10              System.out.println("New area: " + shape.area());
11          }
12      }
13  }
```

```
$ java ShrinkShape2
Original area: 3.0
New area 0.75
```

Get information about a class

- Every object *knows its own class* and can access this information through the **`getClass()`** method, which all classes inherit from class `Object`
  - The `getClass` method returns an object of type **`Class`** (from package `java.lang`), which contains information about the object's type, including its class name
    - Note that keyword `class` and class `Class` are different things!
  - The result of the `getClass` call is used to invoke **`getName()`** to get the object's class name

Example of getClass()

```
1  abstract class Animal {
2      public abstract void makeSound();
3  }
4  class Cat extends Animal {
5      public void makeSound() {System.out.println("Meow!");}
6  }
7  class Dog extends Animal {
8      public void makeSound() {System.out.println("Woff woff!");}
9  }
10 public class AnimalGetClass {
11     public static void main(String[] args) {
12         Animal animal = new Cat();
13         Class cl = animal.getClass();
14         System.out.println("Animal is " + cl.getName());
15     }
16 }
```

```
$ java AnimalGetClass
Animal is Cat
```

Final methods and classes

- A *final method* in a superclass cannot be overridden in a subclass
  - Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass
  - Methods that are declared `static` are implicitly `final`
  - A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as *static binding*
- A *final class* cannot be extended to create a subclass
  - All methods in a `final` class are implicitly `final`

Final classes in Java API

- Class `String` is an example of a `final` class
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected
  - Since class `String` cannot be extended, programs using `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions
- Note that in the JAVA API, most of the classes are ***not*** declared final

Calling methods from constructors

- ***Do not call overridable methods from constructors***: when creating a *subclass* object, this could lead to an overridden method being called before the *subclass* object is fully initialized
  - Recall that when you construct a *subclass* object, its constructor ***first*** calls one of the direct *superclass's* constructors
  - If the *superclass* constructor calls an overridable method, the *subclass's* version of that method will be called by the *superclass* constructor—before the *subclass* constructor's body has a chance to execute
  - Difficult-to-detect errors can occur if the *subclass* method depends on initialization not yet been performed in the *subclass* constructor
- However, it is acceptable to call a `static` method from a constructor

Example of casting to a subclass

```
1  abstract class Animal {
2      public Animal() {
3          System.out.println("Called constructor Animal");
4      }
5  }
6  abstract class Mammal extends Animal {
7      public Mammal() {
8          System.out.println("Called constructor Mammal");
9      }
10 }
11 class Cat extends Mammal {
12     public Cat() {
13         System.out.println("Called constructor Cat");
14     }
15 }
16 public class ConstructorExample1 {
17     public static void main(String[] args) {
18         Cat cat = new Cat();
19     }
20 }
```

```
$ java ConstructorExample1
Called constructor Animal
Called constructor Mammal
Called constructor Cat
```

Example of casting to a subclass

```
1  abstract class Animal {
2      public String sound() { return "nothing"; }
3      public Animal() {
4          System.out.println("Animal says “ + sound());
5      }
6  }
7  class Cat extends Animal {
8      public String sound() { return "meow"; }
9      public Cat() {
10         System.out.println("Cat says “ + sound());
11     }
12 }
13 public class ConstructorExample2 {
14     public static void main(String[] args) {
15         Cat cat = new Cat();
16     }
17 }
```

```
$ java ConstructorExample1
Animal says meow
Cat says meow
```

Questions, comments?