Lecture 23

Abstract Classes and Interfaces
Abstract Classes

• When talking about inheritance, our subclasses are more specific and the superclass is more generic

• Sometimes, superclasses can be so generic that it doesn’t make sense to ever create an instance of that object outside of a subclass

• That’s when we make it an Abstract class
Abstract Classes

• Let’s use the GeometricObject class from the book as an example
Abstract Classes

• unmodified

```java
public class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    /** Construct a default geometric object */
    protected GeometricObject(){
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color *
     * and filled value */
    protected GeometricObject(String color, boolean filled){
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor(){
        return color;
    }

    /** Set color */
    public void setColor(String color) {
        this.color = color;
    }

    /** Return filled. Since filled is boolean, 
     * its getter method is named isFilled */
    public boolean isFilled() {
        return filled;
    }

    /** Set a new filled */
    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    /** Get dateCreated */
    public java.util.Date getDateCreated() {
        return dateCreated;
    }

    /** Return a string representation of this object */
    public String toString() {
        return "created on " + dateCreated + "color: " + color + " and filled: " + filled;
    }
}
```
Abstract Classes

• For our GeometricObject, we extend it to make Circle and Rectangle objects

• Grab the GeometricObject, Circle and Rectangle files from the course site if you’d like to follow along
Abstract Classes

• Circle:

```java
public class Circle extends GeometricObject {
    private double radius;

    public Circle() {
    }

    public Circle(double radius) {
        this.radius = radius;
    }

    /** Return radius */
    public double getRadius() {
        return radius;
    }

    /** Set a new radius */
    public void setRadius(double radius) {
        this.radius = radius;
    }

    /** Return area */
    public double getArea() {
        return radius * radius * Math.PI;
    }

    /** Return diameter */
    public double getDiameter() {
        return 2 * radius;
    }

    /** Return perimeter */
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    // Print the circle info *
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() + " and the radius is " + radius);
    }
}
```

• Rectangle:

```java
public class Rectangle extends GeometricObject {
    private double width;
    private double height;

    public Rectangle() {
    }

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    /** Return width */
    public double getWidth() {
        return width;
    }

    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }

    /** Return height */
    public double getHeight() {
        return height;
    }

    /** Set a new height */
    public void setHeight(double height) {
        this.height = height;
    }

    /** Return area */
    public double getArea() {
        return width * height;
    }

    // Print the rectangle info *
    public void printRectangle() {
        System.out.println("The rectangle is created " + getDateCreated() + " with width " + width + " and height " + height);
    }
}```
public class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    /** Construct a default geometric object */
    protected GeometricObject(){
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color * and filled value */
    protected GeometricObject(String color, boolean filled){
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor(){
        return color;
    }

    /** Set color */
    public void setColor(String color) {
        this.color = color;
    }

    /** Return filled. Since filled is boolean, its getter method is named isFilled */
    public boolean isFilled() {
        return filled;
    }

    /** Set a new filled */
    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    /** Get dateCreated */
    public java.util.Date getDateCreated() {
        return dateCreated;
    }

    /** Return a string representation of this object */
    public String toString(){
        return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled;
    }
}
Abstract Classes

• Notice that in our Circle and our Rectangle we have getArea() and getPerimeter() methods

• These are methods that can be used for *any* geometric object, right? So they should go in our GeometricObject class

• but how do you compute the area or perimeter for a generic geometric object?
Abstract Classes

• Can’t be implemented in the generic class, so these methods declared as *abstract* in the superclass

```java
public abstract double getArea();
public abstract double getPerimeter();
```

• **Note:** there are no curly braces here

• Once you add these methods, the class needs to be declared as *abstract* as well, and the constructors declared as *protected*

```java
public abstract class GeometricObject {
```
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    /** Construct a default geometric object */
    protected GeometricObject(){
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color *
     * and filled value */
    protected GeometricObject(String color, boolean filled){
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor(){
        return color;
    }

    /** Set color */
    public void setColor(String color) {
        this.color = color;
    }

    /** Return filled. Since filled is boolean, 
    its getter method is named isFilled */
    public boolean isFilled() {
        return filled;
    }

    /** Set a new filled */
    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    /** Return a string representation of this object */
    public String toString(){
        return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled;
    }

    /** Abstract method to return the area of a geometric object */
    public abstract double getArea();

    /** Abstract method to return the perimeter of a geometric object */
    public abstract double getPerimeter();
}
Abstract Classes

- Abstract classes are mostly used like a superclass.

- *But* you can’t create an instance of it anymore (using the *new* keyword like we have been).

- It must be inherited, that’s why the constructor is protected.
public class Circle extends GeometricObject {
    private double radius;

    public Circle() {
    }

    public Circle(double radius) {
        this.radius = radius;
    }

    /** Return radius */
    public double getRadius() {
        return radius;
    }

    /** Set a new radius */
    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override /** Return area */
    public double getArea() {
        return radius * radius * Math.PI;
    }

    /** Return diameter */
    public double getDiameter() {
        return 2 * radius;
    }

    @Override /** Return perimeter */
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    /* Print the circle info */
    public void printCircle() {
        System.out.println("The circle is created "+
                getDateCreated() +
                " and the radius is " + radius);
    }
}

public class Rectangle extends GeometricObject {
    private double width;
    private double height;

    public Rectangle() {
    }

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    /** Return width */
    public double getWidth() {
        return width;
    }

    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }

    /** Return height */
    public double getHeight() {
        return height;
    }

    /** Set a new height */
    public void setHeight(double height) {
        this.height = height;
    }

    @Override /** Return area */
    public double getArea() {
        return width * height;
    }

    @Override /** Return perimeter */
    public double getPerimeter() {
        return width * height;
    }

    @Override /** Return area */
    public double getArea() {
        return width * height;
    }

    @Override /** Return perimeter */
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
Why use abstract methods?

• Methods that need more specific information than is in the superclass (the abstract class) should be declared as abstract and implemented in the subclasses.

• For our example, we can’t get the area or perimeter without knowing what kind of object we’re working with.

• So why not just implement those methods in the subclass? Is there any benefit to having them in the superclass?
Why use abstract methods?

- Let’s look at an example to explore this
Why use abstract methods?

```java
public class TestGeometricObject {

    public static void main(String[] args) {
        GeometricObject geo1 = new Circle(5);
        GeometricObject geo2 = new Rectangle(5, 3);

        System.out.println("Is the area the same? " + equalArea(geo1, geo2));

        printShapeData(geo1);
        printShapeData(geo2);
    }

    public static boolean equalArea(GeometricObject geo1, GeometricObject geo2) {
        return geo1.getArea() == geo2.getArea();
    }

    public static void printShapeData(GeometricObject geo) {
        System.out.println();
        System.out.println("The area of the object is: " + geo.getArea());
        System.out.println("The perimeter of the object is: " + geo.getPerimeter());
    }
}
```
Why use abstract methods?

- Because then we can use those subclasses polymorphically!
- Meaning, we can use them anywhere we use the superclass
- If you just defined them in the individual subclasses, we couldn’t do those last 2 methods, comparing and printing the objects in a flexible way
Notes about abstract classes

• Abstract methods *must* be in an abstract class. An abstract class can have 0 or more abstract methods.

• A non-abstract subclass *must* implement *all* the abstract methods of the superclass (can’t pick and choose, unless you make that subclass abstract as well…)

• The opposite of an abstract class is a **concrete class**

• A subclass of a concrete class could be abstract (**Object** class is concrete)

• You can’t make instances of an abstract class, but you can use them in as a datatype, like in an array:

```java
GeometricObject[] objects = new GeometricObject[5];
```
Let’s look at another example: Number

- The `Number` class is an abstract class, let’s take a look

- Command-click or Control-click on `Double`, then `Number` to see the source

```java
Double doub = new Double(5);
```
Let’s look at another example: Number

- The **Number** class is abstract and has abstract methods we’ve seen before like `intValue()`, `floatValue()`, `doubleValue()` that let’s us convert between formats
Interfaces

• Interfaces are similar to abstract classes, but they are used to define common behavior of classes whether or not the classes are related

• Some uses of this would be creating classes that are comparable, cloneable, or edible :)

• Syntax like this:

```java
public interface Edible {
    public abstract String howToEat();
}
```
Interfaces

• Put the interface in a new file, but not a new class!
Interfaces

• An **Interface** is treated like a special class in java

• Like an abstract class, you can use it as a datatype (like starting an array), but can’t create instances of it

• instead of extends, we use implements
Interfaces

- Let’s look at an example
Interfaces

```java
public class TestEdible {
    public static void main(String[] args) {
        Object[] things = {new Tiger(), new Chicken(), new Apple();

        for (int i = 0; i < things.length; i++) {
            if (things[i] instanceof Edible) {
                System.out.println(((Edible)things[i]).howToEat());
            }
            if (things[i] instanceof Animal) {
                System.out.println(((Animal)things[i]).sound());
            }
        }
    }
}

abstract class Animal{
    public abstract String sound();
}

abstract class Fruit implements Edible {
}

class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: fry it!";
    }
    public String sound() {
        return "Chicken: Cluck, Cluck";
    }
}

class Tiger extends Animal {
    public String sound() {
        return "Tiger: RAWR!";
    }
}

class Apple extends Fruit {
    public String howToEat() {
        return "Apple: slice and eat with peanut butter";
    }
}
```
Interfaces

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.

```
«interface» Edible
+howToEat(): String
```

```
Animal
+sound(): String
```

```
Fruit
Orange
Apple

Chicken

Tiger
```
Interfaces

• Interfaces define common behaviors for classes.

• So the Edible interface defines common behaviors for all edible objects
Built-in Interfaces

- Java has some interfaces already created that we can use with our classes
Comparable Interface

```java
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```

- This interface allows you to define the order of 2 objects of the same class (like which is bigger, or older, etc), for example, two rectangles, or two dates, or two apples.

- We’ve seen this used in **Number** subclasses already like **Integer** and **Double**
Comparable Interface

- This is super useful when you sorting objects in an array using `java.util.Arrays.sort(Object[])`

- We can sort strings and numbers already because they implement the `Comparable` interface (and so have a `compareTo()` method)

- In order to sort our objects, we can do the same
public class ComparableRectangle extends GeometricObject implements Comparable<ComparableRectangle> {
    private double width;
    private double height;

    public ComparableRectangle() {
    }

    public ComparableRectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public int compareTo(ComparableRectangle rect2) {
        if (getArea() > rect2.getArea()) {
            return 1;
        } else if (getArea() < rect2.getArea()) {
            return -1;
        } else {
            return 0;
        }
    }

    /** Return width */
    public double getWidth() {
        return width;
    }

    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }

    /** Return height */
    public double getHeight() {
        return height;
    }

    /** Set a new height */
    public void setHeight(double height) {
        this.height = height;
    }

    @Override
    /** Return area */
    public double getArea() {
        return width * height;
    }

    @Override
    /** Return perimeter */
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
Comparable Interface

```java
public class SortRectangles {

    public static void main(String[] args) {
        ComparableRectangle[] rects = {
            new ComparableRectangle(3,4),
            new ComparableRectangle(3,5),
            new ComparableRectangle(3,2),
            new ComparableRectangle(3,8),
            new ComparableRectangle(3,7),
        };

        java.util.Arrays.sort(rects);
        for (ComparableRectangle rect: rects){
            System.out.println(
                "width: " + rect.getWidth()
                + " height: " + rect.getHeight()
                + " area: " + rect.getArea());
        }
    }
}
```
Abstract Classes vs. Interfaces

<table>
<thead>
<tr>
<th>Variables</th>
<th>Constructors</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class</td>
<td>No restrictions.</td>
<td>No restrictions.</td>
</tr>
<tr>
<td>Interface</td>
<td>All variables must be <strong>public static final</strong>.</td>
<td>All methods must be public abstract instance methods</td>
</tr>
</tbody>
</table>

- Java only lets you *inherit* from one class, but you can *implement* multiple interfaces, you just add commas between them

    ```java
    public interface NewInterface extends Interface1, ..., InterfaceN {
        // constants and abstract methods
    }
    ```
Abstract Classes vs. Interfaces

• Another way to think of it is using the “is-a” relationship test.

• A strong “is-a” relationship should use classes (for example, a Cat “is-a” Animal)

• A weak “is-a” relationship, or “is-kind-of” relationship should use interfaces (for example, a Fruit is a kind of Edible object)

• In general, (and this is very broad) interfaces are preferred over abstract classes because they are more flexible and can be used for unrelated classes
Abstract Classes vs. Interfaces

- Classes are nouns, interfaces are nouns or adjectives
Chapter 13.10 has some great guidelines on designing classes!
Practice

- Check out the case studies in the chapter (13)
- Problems 13.6, 13.7, 13.9, 13.10, 13.12