Chapter 11 - Abstract Class and Interfaces

Section 11.1 - Abstract classes: Introduction

Object-oriented programming (OOP) is a powerful programming paradigm, consisting of several features. One feature involves a *class*, which encapsulates data and behavior to create objects. Another feature is *inheritance*, which allows one class (a subclass) to be based on another class (a base class or superclass). For example, a Shape class may encapsulate data and behavior for geometric shapes, like setting/getting the Shape's name and color, while a Circle class may be a subclass of a Shape, with additional features like setting/getting the center point and radius.

A third feature is the idea of an abstract class. An ***abstract class*** is a class that guides the design of subclasses but cannot itself be instantiated as an object. For example, a Shape class might not only have behavior for setting/getting the Shape's name and color, but also specifies that any subclass must define a method named computeArea().

Participation Activity

11.1.1: Classes, inheritance, and abstract classes.

Start

Shape (abstract)

+Get/set name
+Get/set color
+Compute area (abstract)

Circle (derived from Shape)

+Get/set center point
+Get/set radius
+Compute area

Objects:

~~shape1~~
~~shape2~~

circle1
circle2

An example of abstract classes in action is the hierarchy of classification used in biology. The upper levels of the hierarchy specify features in common across all members below that level of the hierarchy. As with concrete classes that implement all abstract methods, no creature can actually be instantiated except at the species level.

Participation Activity

11.1.2: Biological classification uses abstract classes.

Start

Domain

Kingdom — Kingdom Animalia specifies animals

Phylum

Class — Class Mammalia specifies animals with mammary glands

Order — Order Carnivora specifies animals who eat meat

Family

Genus — Genus Canis and Species lupus familaris is the domestic dog.

Species

The hierarchy of biological classification is an example of abstract classes

P Participation Activity

11.1.3: Abstract classes.

#	Question	Your answer
1	An abstract class can be instantiated as an object.	True False
2	From the example above, the Shape class is an abstract class and the Circle class is a concrete class.	True False
3	Consider a program that catalogs the types of trees in a forest. Each tree object contains the tree's species type, age, and location. This program will benefit from an abstract class to represent the trees.	True False
4	Consider a program that catalogs the types of trees in a forest. Each tree object contains the tree's species type, age, location, and estimated size based on age. Each species uses a different formula to estimate size based on age. This program will benefit from an abstract class.	True False
5	Consider a program that maintains a grocery list. Each item, like eggs, has an associated price and weight. Each item belongs to a category like produce, meat, or cereal, where each category has additional features, such as meat having a "sell by" date. This program will benefit from an abstract class.	True False

Section 11.2 - Abstract classes

An ***abstract class*** is a class that cannot be instantiated as an object, but is the superclass for a subclass and specifies how the subclass must be implemented. A ***concrete class*** is a class that is not abstract, and hence *can* be instantiated. An abstract class is denoted by the keyword ***abstract*** in front of the class definition. The example program below manages sets of shapes. Shape is an abstract class, and Circle and Rectangle are concrete classes. The Shape abstract class merely specifies that any derived class must define a method computeArea() that returns type double.

Figure 11.2.1: Shape is an abstract class. Circle and Rectangle are concrete classes that extend the Shape class.

Shape.java specifies how a programmer interacts with shapes

```
public abstract class Shape {

   abstract double computeArea();
}
```

Point.java holds the x, y coordinates for a

```
public class Point {

   private double x;
   private double y;

   public Point(double x, double y)
      this.x = x;
      this.y = y;
   }

   public double getX() {
      return x;
   }

   public double getY() {
      return y;
   }
}
```

Circle.java defines a Circle class

```
public class Circle extends Shape {

   private double radius;
   private Point center;

   public Circle(Point center, double radius)
      this.radius = radius;
      this.center = center;
   }

   @Override
   public double computeArea() {
      return (Math.PI * Math.pow(radius, 2));
   }
}
```

Rectangle.java defines a Rectangle class

```
public class Rectangle extends Shape

   private Point lowerLeft, upperRig

   Rectangle(Point lowerLeft, Point
      this.lowerLeft = lowerLeft;
      this.upperRight = upperRight;
   }

   @Override
   public double computeArea() {
      double length = 0.0;
      double height = 0.0;

      length = upperRight.getX() - l
      height = upperRight.getY() - l

      return (length * height);
   }
}
```

TestShapes.java tests the Shape class

```
public class TestShapes {
   public static void main(String[] args) {
      Circle circle1 = new Circle(new Point(0.0, 0.0), 1.0);
      Circle circle2 = new Circle(new Point(0.0, 0.0), 2.0);

      Shape rectangle = new Rectangle(new Point(0.0, 0.0), new Point(1.0, 1.0));

      System.out.println("Area of circle 1 is: " + circle1.computeArea());
      System.out.println("Area of circle 2 is: " + circle2.computeArea());
      System.out.println("Area of rectangle is: " + rectangle.computeArea());

      return;
   }
}
```

```
Area of circle 1 is: 3.141592653589793
Area of circle 2 is: 12.566370614359172
Area of rectangle is: 1.0
```

A program cannot use the new operator to create an instance of an abstract class. For example, the variable initialization `Shape shape1 = new Shape();` generates a compiler error like the following:

Figure 11.2.2: Sample compiler error when trying to define an object of an abstract base class type.

```
javac TestShapes.java
TestShapes.java:5: error: Shape is abstract; cannot be instantiated
      Shape shape1 = new Shape();
                     ^
1 error
```

An abstract class can contain methods and variables that are shared by subclasses. An abstract class may also contain abstract methods, such as method computeArea() in class Shape. An ***abstract method*** is a method that each subclass must implement to be a concrete class. If a subclass does not implement an abstract method, then the subclass must also be defined as abstract.

P Participation Activity

11.2.1: Abstract and concrete classes.

Run the code and observe that the code calls the correct area method. Add a new abstract method `double computePerimeter()` to the Shape class and implement the method within each of the concrete classes. Modify the main() method in the TestShapes class to use the computePerimeter() method for the Circle and Rectangle objects.

Point.java	Shape.java	Circle.java	Rectangle.java

```
1
2  public class Point {
3
4     private double x;
5     private double y;
6
7     public Point(double x, double y) {
8        this.x = x;
9        this.y = y;
10    }
11
12    public double getX() {
13       return x;
14    }
15
16    public double getY() {
17       return y;
18    }
19 }
```

Run

P Participation Activity

11.2.2: Abstract class instantiation.

Given the above Shape example, select the line that will fail to compile.

#	Question

1	`Shape shape1 = new Shape();`
	`Shape shape2 = new Circle(new Point(0.0, 0.0), 1.0);`
	`Shape shape3 = new Rectangle(new Point(0.0, 0.0), new Point(2.0, 2.0));`

P Participation Activity

11.2.3: Abstract class example.

Some questions refer to the above shapes example.

#	Question	Your answer
1	Shape is what kind of class?	Subclass
		Concrete
		Abstract
2	Circle is what kind of class?	Abstract
		Concrete
3	Can the Shape class define and provide code for non-abstract methods?	Yes, an abstract class can include both method signatures for abstract methods and complete code for non-abstract methods.
		Yes, but the class can only have one non-abstract method.
		No, all methods of an abstract class must be abstract.
4	If the Circle class omitted the computeArea() implementation, could Circle objects be instantiated?	Yes, a subclass of an abstract class can be instantiated.
		No, in that case the subclass must be defined as abstract.

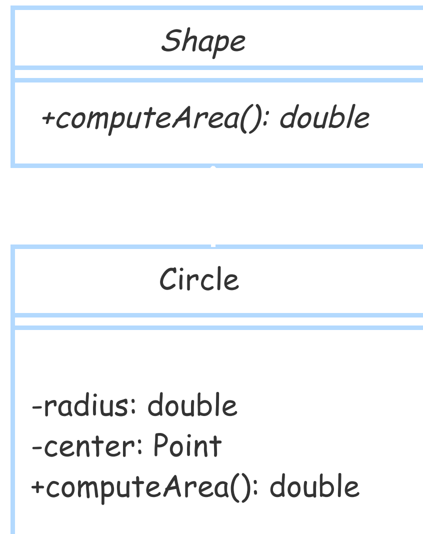
Section 11.3 - UML for abstract classes

UML uses italics to denote abstract classes. In particular, UML uses italics for the abstract class' name, and for any abstracts methods in the class. As a reminder, a superclass does not have to be abstract. Also, any class with an abstract method must be abstract.

P Participation Activity

11.3.1: UML uses italics for abstract classes and methods.

Start

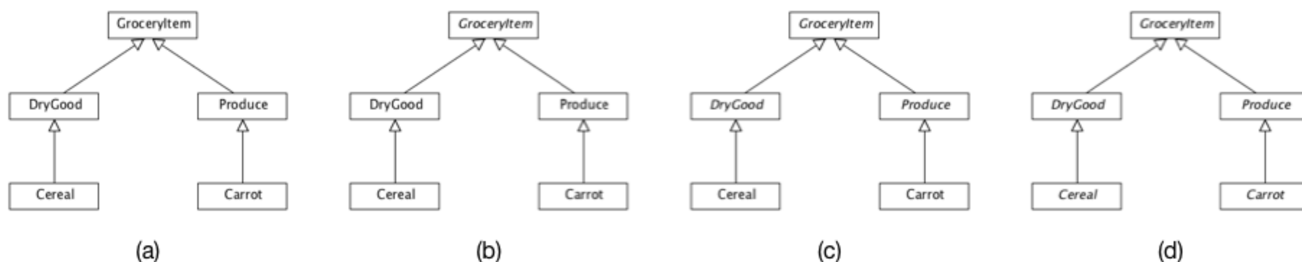


UML for abstract and concrete classes

P Participation Activity

11.3.2: UML for abstract classes.

Match the UML diagram to the best description for that diagram. Each of the questions concerns a different implementation of a grocery store inventory system.



(d) (c) (a) (b)

Drag and drop above item	GroceryItems are abstract because they require all subclasses to implement specific methods. DryGoods and Produce can be created as classes.
	GroceryItems are abstract because they require all subclasses to implement specific methods. DryGoods and Produce are also abstract as they require subclasses to implement methods specific to each class.
	All classes are concrete.
	All classes are abstract.

Reset

Section 11.4 - Abstract classes and polymorphism

Abstract classes provide runtime polymorphism, which enables a programmer to use an abstract method without worrying about which concrete class implements the abstract method. When the program executes, the JVM will automatically call the method of the concrete subclass. Abstract

classes are especially powerful when used in combination with arrays or Java Collections Framework classes, including ArrayList, Map, etc.

Figure 11.4.1: Polymorphism example.

```
import java.util.ArrayList;

public class PolymorphismExample {
   public static void main(String[] args) {
      ArrayList<Shape> shapesList = new ArrayList<Shape>();

      Circle circle = new Circle(new Point(0.0, 0.0), 1.0);
      shapesList.add(circle);

      Rectangle rectangle = new Rectangle(new Point(0.0, 0.0), new Point(2.0, 2.0));
      shapesList.add(rectangle);

      for (Shape shape : shapesList) {
         System.out.println("Shape is: " + shape.getClass() + " and area is: " + sha
      }

      return;
   }
}
```

Participation Activity

11.4.1: Abstract and concrete classes.

Run the code and observe that the code calls the correct method for computeArea() even though the ArrayList is using the abstract superclass of Shape. Add print statements to the computeArea() method to ensure the code really calls the right class.

PolymorphismExample.java	Shape.java	Point.java
Rectangle.java		

Reset

```
1
2 import java.util.ArrayList;
3
4 public class PolymorphismExample {
5    public static void main(String[] args) {
6       ArrayList<Shape> shapesList = new ArrayList<Shape>();
7
8       Circle circle = new Circle(new Point(0.0, 0.0), 1.0);
9       shapesList.add(circle);
10
```

```
10
11          Rectangle rectangle = new Rectangle(new Point(0.0, 0.0), new Point(2.0, 2.0));
12          shapesList.add(rectangle);
13
14          for (Shape shape : shapesList) {
15             System.out.println("Shape is: " + shape.getClass() + " and area is: " + sha
16          }
17
18          return;
19       }
```

Run

P Participation Activity

11.4.2: Polymorphism and ArrayLists.

Given the Shape, Circle, and Rectangle classes, select the block of code that will correctly compile.

#	Question
1	(code blocks below)

```
ArrayList<Circle> circlesList = new
ArrayList<Circle>();
Circle circle1 = new Circle(new Point(0.0, 0.0),
1.0);
circlesList.add(circle1);
Rectangle rectangle1 = new Rectangle(new Point(0.0,
0.0), new Point(2.0, 2.0));
circlesList.add(rectangle1);
```

```
ArrayList<Rectangle> rectanglesList = new
ArrayList<Rectangle>();
Circle circle2 = new Circle(new Point(0.0, 0.0),
1.0);
rectanglesList.add(circle2);
Rectangle rectangle2 = new Rectangle(new Point(0.0,
0.0), new Point(2.0, 2.0));
rectanglesList.add(rectangle2);
```

```
ArrayList<Shape> shapesList = new ArrayList<Shape>
();
Circle circle3 = new Circle(new Point(0.0, 0.0),
1.0);
shapesList.add(circle3);
Rectangle rectangle3 = new Rectangle(new Point(0.0,
0.0), new Point(2.0, 2.0));
shapesList.add(rectangle3);
```

Section 11.5 - Interfaces

Java provides ***interfaces*** as another mechanism for programmers to state that a class adheres to

rules defined by the interface. An ***interface*** specifies a set of methods that an implementing class must override and define. Although inheritance and polymorphism allow a class to override methods defined in the superclass, a class can only inherit from a single superclass. A class can ***implement*** multiple interfaces. Each Interface a class implements means the class will adhere to the rules defined by the interface class.

Example 11.5.1: Interface example.

The *Serializable* interface is a useful interface that illustrates how to use interfaces and why interfaces can be so powerful. The Circle class from above has been modified to implement the Serializable interface. This interface tells Java that objects of type Circle can be written to and read from files (or other I/O Streams). Serializable is an extremely useful interface for large programs that need to save their state.

```
import java.io.Serializable;

public class Circle extends Shape implements Serializable {

   private double radius;
   private Point center;

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

   @Override
   public double computeArea() {
      return (Math.PI * Math.pow(radius, 2));
   }
}
```

To create an interface, a programmer uses the keyword ***interface*** in the class definition. The following code illustrates an interface named DrawableInterface that contains a method declaration for a method drawMe(). A ***method declaration*** within an interface only specifies the method's return type, name, and parameters. The Drawable interface requires classes implementing the interface to define a method called drawMe().

Figure 11.5.1: Creating an interface.

```
public interface DrawableInterface {

   public void drawMe();
}
```

Any class that implements the interface must list the interface name after the keyword ***implements***. A

class can implement multiple interfaces using a comma separated list. For example, Circle can implement both the Serializable and DrawableInterface.

Figure 11.5.2: Implementing an interface.

```
import java.io.Serializable;

public class Circle extends Shape implements Serializable, DrawableInterface {

   private double radius;
   private Point center;

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

   @Override
   public double computeArea() {
      return (Math.PI * Math.pow(radius, 2));
   }

   @Override
   public void drawMe() {
      // TODO: code to draw a circle
   }
}
```

P Participation Activity

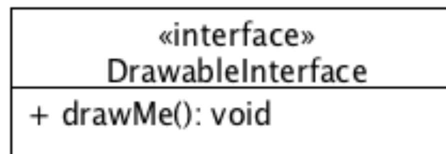
11.5.1: Comparison of interfaces and abstract classes.

Interfaces and abstract classes can seem superficially similar but they have different purposes. The following questions will help clarify these differences. Choose whether an interface or abstract class is the best choice for each situation.

#	Question	Your answer
1	A class that provides default code to other classes that use that class.	Interface Abstract class
2	A class that provides only static final fields.	Interface Abstract class
3	A class provides default variables.	Interface Abstract class
4	A class that provides an API that must be implemented and no other code.	Interface Abstract class

UML Diagrams denote interfaces using the keyword interface, inside double angle brackets, above the class name. Classes that implement the interface have a dashed line with an unfilled arrow pointing at the interface. Following UML conventions is important for clear communication between programmers.

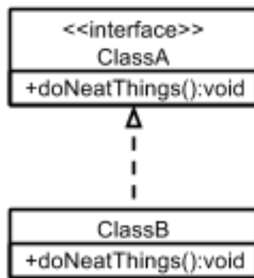
Figure 11.5.3: UML for DrawableInterface.



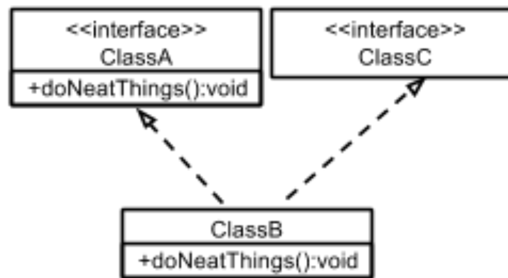
Participation Activity

11.5.2: UML interfaces.

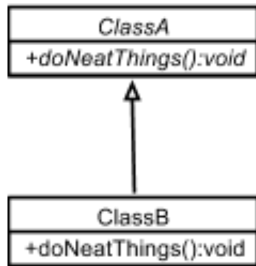
Match the UML diagram from above to the code block that it describes.



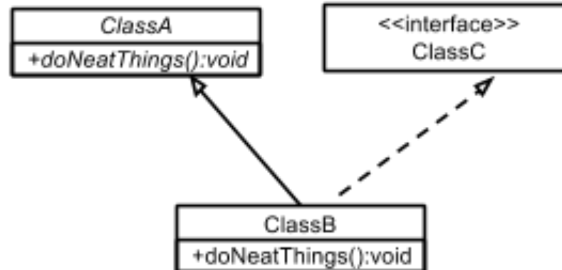
(a)



(b)



(c)



(d)

(a) (d) (c) (b)

Drag and drop above item

```
public abstract class ClassA {
   public abstract void doNeatThings();
}

public interface ClassC {

}
```

```
public class ClassB extends ClassA implements
ClassC {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}
```

```
public interface ClassA {
   public void doNeatThings();
}

public class ClassB implements ClassA {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}
```

```
public abstract class ClassA {
   public abstract void doNeatThings();
}

public class ClassB extends ClassA {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}
```

```
public interface ClassA {
   public void doNeatThings();
}

public interface ClassC {

}

public class ClassB implements ClassA, ClassC {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}
```

Reset

Section 11.6 - Java example: Employees and instantiating from an abstract class

Participation Activity

11.6.1: Employees example: Abstract class and interface.

The classes below describe an abstract class named EmployeePerson and two derived concrete classes, EmployeeManager and EmployeeStaff, both of which extend the EmployeePerson class. The main program creates objects of type EmployeeManager and EmployeeStaff and prints them.

1. Run the program. The program prints manager and staff data using the EmployeeManager's and EmployeeStaff's printInfo methods. Those classes override EmployeePerson's getAnnualBonus() method but simply return 0.
2. Modify the EmployeeManager and EmployeeStaff getAnnualBonus methods to return the correct bonus rather than just returning 0. A manager's bonus is 10% of the annual salary and a staff's bonus is 7.5% of the annual salary.

EmployeeMain.java	EmployeePerson.java	EmployeeManager.java

Reset

```
1  public class EmployeeMain {
2
3     public static void main(String [] args) {
4
5        // Create the objects
6        EmployeeManager manager = new EmployeeManager(25);
7        EmployeeStaff   staff1  = new EmployeeStaff("Michele");
8
9        // Load data into the objects using the Person class's method
10       manager.setData("Michele", "Sales", "03-03-1975", 70000);
11       staff1.setData ("Bob",     "Sales", "02-02-1980", 50000);
12
13       // Print the objects
14       manager.printInfo();
15       System.out.println("Annual bonus: " + manager.getAnnualBonus());
16       staff1.printInfo();
17       System.out.println("Annual bonus: " + staff1.getAnnualBonus());
18
19       return;
```

Pre-enter any input for program, then press run

Run

P Participation Activity

11.6.2: Employees example: Abstract class and interface (solution).

Below is the solution to the above problem. Note that the EmployeePerson class is unchanged.

EmployeeMain.java	EmployeePerson.java	EmployeeManager.java

Reset

```
1  public class EmployeeMain {
2
3     public static void main(String [] args) {
4
5        // Create the objects
6        EmployeeManager manager = new EmployeeManager(25);
7        EmployeeStaff   staff1  = new EmployeeStaff("Michele");
8
9        // Load data into the objects using the Person class's method
10       manager.setData("Michele", "Sales", "03-03-1975", 70000);
11       staff1.setData ("Bob",     "Sales", "02-02-1980", 50000);
12
13       // Print the objects
14       manager.printInfo();
15       System.out.println("Annual bonus: " + manager.getAnnualBonus());
16       staff1.printInfo();
17       System.out.println("Annual bonus: " + staff1.getAnnualBonus());
18
19       return;
```

Pre-enter any input for program, then press run

Run