15.1 Abstract classes: Introduction (generic)

Abstract classes

Object-oriented programming (OOP) is a powerful programming paradigm, consisting of several features. Three key features include:

- ***Classes:*** A class encapsulates data and behavior to create objects.
- ***Inheritance:*** Inheritance allows one class (a subclass) to be based on another class (a base class or superclass). Ex: 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.
- ***Abstract classes:*** An ***abstract class*** is a class that guides the design of subclasses but cannot itself be instantiated as an object. Ex: An abstract Shape class might also specify that any subclass must define a computeArea() method.

PARTICIPATION ACTIVITY 15.1.1: Classes, inheritance, and abstract classes.

Animation content:

undefined

Animation captions:

1. A class provides data/behaviors for objects.
2. Inheritance creates a Circle subclass that implements behaviors specific to a circle.
3. The abstract Shape class specifies "Compute area" is a required behavior of a subclass. Shape does not implement "Compute area", so a Shape object cannot be created.
4. The Circle class implements "Compute area". The Circle class is a non abstract, which is also called a concrete class, and Circle objects can be created.

PARTICIPATION ACTIVITY 15.1.2: Classes, inheritance, and abstract classes.

Consider the example above.

1) The Shape class is an abstract class, and the Circle class is a concrete class.

- True
- False

2) The Shape class can be instantiated as an object.

- True
- False

3) The Circle class can be instantiated as an object.

- True
- False

4) The Circle class must implement the computeArea() method.

- True
- False

Example: Biological classification

An example of abstract classes in action is the classification hierarchy 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 15.1.3: Biological classification uses abstract classes.

Animation content:

`undefined`

Animation captions:

1. Each level of the biological hierarchy specifies behaviors common to that level.
2. At this level of the hierarchy, a lot of behavior for the organism is known but the organism is not yet specified.
3. At the final level (species), the organism can be fully described, just as a concrete class can be fully instantiated.

PARTICIPATION ACTIVITY 15.1.4: Abstract classes.

1) 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

2) 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

3) 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

15.2 Abstract classes

Abstract and concrete classes

An ***abstract method*** is a method that is not implemented in the base class, thus all derived classes must override the function. An abstract method is denoted by the keyword `abstract` in front of the method signature. A ***method signature*** defines the method's name and parameters. Ex: `abstract double computeArea();` declares an abstract method named computeArea().

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. An abstract class is denoted by the keyword `abstract` in front of the class definition. Any class with one or more abstract methods must be abstract.

A ***concrete class*** is a class that is not abstract, and hence *can* be instantiated.

PARTICIPATION ACTIVITY 15.2.1: A Shape class with an abstract method is an abstract class.

Animation content:

undefined

Animation captions:

1. The Shape class has the abstract computeArea() method.
2. The Shape class is abstract due to having an abstract method, and must contain the abstract keyword in the declaration.
3. An abstract class cannot be instantiated.

PARTICIPATION ACTIVITY 15.2.2: Shape class.

1) Shape is an abstract class.
   - True
   - False

2) The Shape class defines and provides code for non-abstract methods.
   - True
   - False

3) Any class that inherits from Shape must implement the computeArea(), getPosition(), setPosition(), and movePositionRelative() methods.

- True
- False

Ex: Shape classes

The example program below manages sets of shapes. Shape is an abstract class, and Circle and Rectangle are concrete classes. The Shape abstract class specifies that any derived class must define a method computeArea() that returns type double.

©zyBooks 12/08/22 21:33 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 15.2.1: Shape is an abstract class. Circle and Rectangle are concrete classes that extend the Shape class.

Shape.java implements the Shape base class

```
public abstract class Shape {

   protected Point position;

   abstract double computeArea();

   public Point getPosition() {
      return this.position;
   }

   public void setPosition(Point
position) {
      this.position = position;
   }

   public void
movePositionRelative(Point position)
{
      double x =
this.position.getX() +
position.getX();
      double y =
this.position.getY() +
position.getY();

      this.position.setX(x);
      this.position.setY(y);
   }
}
```

Point.java holds the x, y coordinates for a point


```
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;
   }

   public void setX(double x) {
      this.x = x;
   }

   public void setY(double y) {
      this.y = y;
   }
}
```

Circle.java defines a Circle class

```
public class Circle extends Shape {

   private double radius;

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

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

Rectangle.java defines a Rectangle class

```
public class Rectangle extends
Shape {

   private double length,
height;

   Rectangle(Point upperLeft,
double length, double height) {
      this.position =
upperLeft;
      this.length = length;
      this.height = height;
   }

   @Override
   public double computeArea()
```

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

TestShapes.java tests the Shape class


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

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

      // Print areas
      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());
      System.out.println();

      // Print positions
      System.out.println("Circle 1 is at: (" +
circle1.getPosition().getX() +
                         ", " + circle1.getPosition().getY() + ")");

      System.out.println("Rectangle is at: (" +
rectangle.getPosition().getX() +
                         ", " + rectangle.getPosition().getY() + ")");
      System.out.println();

      // Move shapes
      circle1.setPosition(new Point(3.0, 1.0));
      rectangle.movePositionRelative(new Point(1.0, 1.0));

      // Print positions
      System.out.println("Circle 1 is at: (" +
circle1.getPosition().getX() +
                         ", " + circle1.getPosition().getY() + ")");

      System.out.println("Rectangle is at: (" +
rectangle.getPosition().getX() +
                         ", " + rectangle.getPosition().getY() + ")");
      System.out.println();
   }
}
```



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

```
Circle 1 is at: (1.0, 1.0)
Rectangle is at: (0.0, 1.0)

Circle 1 is at: (3.0, 1.0)
Rectangle is at: (1.0, 2.0)
```

PARTICIPATION ACTIVITY 15.2.3: Shape classes.

1) Since the Circle and Rectangle classes both implement the computeArea() method, Circle and Rectangle are both abstract.
   - ○ True
   - ○ False

2) An instance of the ____ class cannot be created.
   - ○ Shape
   - ○ Point
   - ○ Circle

3) The getPosition() method of the Circle class is implemented in the _____ class.
   - ○ Circle
   - ○ Rectangle
   - ○ Shape

4) If the Circle class omitted the computeArea() implementation, could Circle objects be instantiated?
   - ○ Yes
   - ○ No

CHALLENGE ACTIVITY 15.2.1: Abstract classes.

422352.2723990.qx3zqy7

Start

Type the program's output

TestPerson.java | Person.java | Student.java | Teacher.java


```
public class TestPerson {
   public static void main(String[] args) {
      Student myStudent = new Student("Paula", 11, 3.5);
      Teacher myTeacher = new Teacher("Monet", 37, "Art");

      myStudent.printInfo();
      System.out.println();
      myTeacher.printInfo();
   }
}
```

Paul
GPA

Mone
Subj

1

Check Try again

15.3 Interfaces

Java provides ***interfaces*** as another mechanism for programmers to state that a class adheres to rules defined by the interface. An ***interface*** can specify a set of abstract methods that an implementing class must override and define. In an interface, an abstract method does not need the `abstract` keyword in front of the method signature.

To create an interface, a programmer uses the keyword ***interface***. The following code illustrates two interfaces named Drawable and DrawableASCII.

Figure 15.3.1: Creating an interface.


```
import java.awt.Graphics2D;

public interface Drawable {
   public void draw(Graphics2D
graphicsObject);
}
```

```
public interface DrawableASCII
{
   public void drawASCII(char
drawChar);
}
```

Drawable declares a draw() method for drawing using Java Swing components, which are discussed elsewhere. DrawableASCII declares a drawASCII() method for drawing using ASCII characters.

Any class that implements an interface must:

- List the interface name after the keyword ***implements***
- Override and implement the interface's abstract methods

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 using a comma separated list. Each Interface a class implements means the class will adhere to the rules defined by the interface. Ex: Square can implement both the Drawable and DrawableASCII interfaces.

Figure 15.3.2: Implementing an interface.

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Rectangle;

public class Square implements Drawable, DrawableASCII {
   private int sideLength;

   public Square(int sideLength) {
      this.sideLength = sideLength;
   }

   @Override
   public void draw(Graphics2D graphicsObject) {
      Rectangle shapeObject = new Rectangle(0, 0, this.sideLength,
this.sideLength);
      Color colorObject = new Color(255, 0, 0);
      graphicsObject.setColor(colorObject);
      graphicsObject.fill(shapeObject);
   }

   @Override
   public void drawASCII(char drawChar) {
      int rowIndex;
      int columnIndex;

      for (rowIndex = 0; rowIndex < this.sideLength; ++rowIndex) {
         for (columnIndex = 0; columnIndex < this.sideLength;
++columnIndex) {
            System.out.print(drawChar);
         }
         System.out.println();
      }
   }
}
```


PARTICIPATION ACTIVITY 15.3.1: Comparison of interfaces and abstract classes.

©zyBooks 12/08/22 21:33 1361995 John Farrell COLOSTATECS165WakefieldFall2022

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.

1) Provides only static final fields.

- Interface
- Abstract class

2) Provides variables/fields.

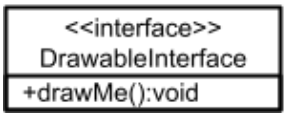
- Interface
- Abstract class

3) Provides an API that must be implemented and no other code.

- Interface
- Abstract class

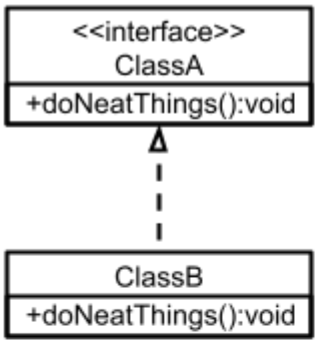
UML diagrams denote interfaces using the keyword interface, inside double angle brackets, above the interface name. Classes that implement the interface have a dashed line with an unfilled arrow pointing at the interface. Following UML conventions is important for clear communication between programmers.

Figure 15.3.3: UML for DrawableInterface.

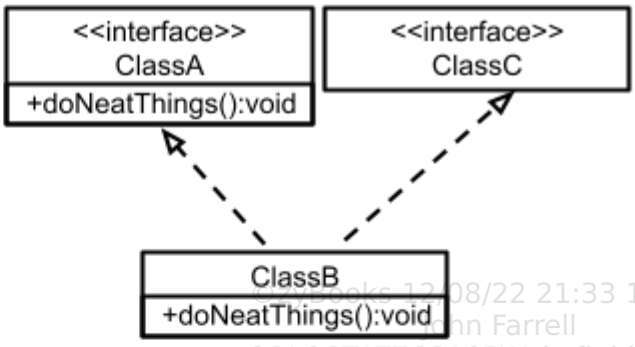


PARTICIPATION ACTIVITY 15.3.2: UML interfaces.

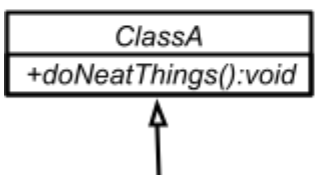
Match the UML diagram from above to the code block that it describes.

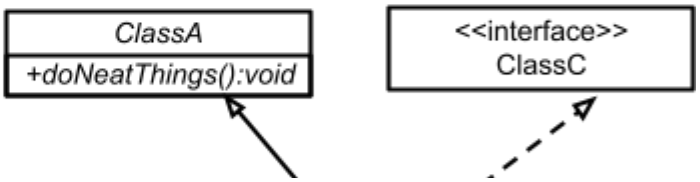


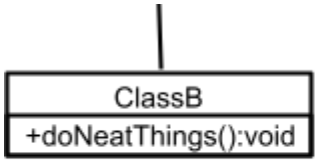
(a)



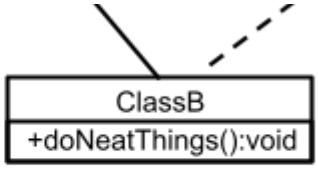
(b)







(c)



(d)

If unable to drag and drop, refresh the page.

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

```
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

CHALLENGE ACTIVITY 15.3.1: Enter the output of the class implementing interfaces.

422352.2723990.qx3zqy7

Start

Type the program's output

PrintMedia.java | MultiMedia.java | Show.java | Media.java

```
public class PrintMedia {
   public static void main(String[] args) {
      Media med = new Media();

      med.setDirector("Charlie Brooker");
      med.setDuration(45);

      med.printDuration();
      med.printDirector();
   }
}
```

```
45 minutes
Director: Char
```


1 2

15.4 Java example: Employees and instantiating from an abstract class

This section has been set as optional by your instructor.

zyDE 15.4.1: Employees example: Abstract class and interface.

The classes below describe an abstract class named EmployeePerson and two deriv concrete classes, EmployeeManager and EmployeeStaff, both of which extend the EmployeePerson class. The main program creates objects of type EmployeeManage 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 ov EmployeePerson's getAnnualBonus() method but simply return 0.
2. Modify the EmployeeManager and EmployeeStaff getAnnualBonus methods to the correct bonus rather than just returning 0. A manager's bonus is 10% of the salary and a staff's bonus is 7.5% of the annual salary.

Current file: **EmployeeMain.java** Load default ter

```
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());
```

Pre-enter any input for program, then press run.

Run

zyDE 15.4.2: Employees example: Abstract class and interface (solution).

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

Current file: EmployeeMain.java Load default ter

```
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());
```

Pre-enter any input for program, then press run.

Run

©zyBooks 12/08/22 21:33 1361995
John Farrell
COLOSTATECS165WakefieldFall2022