CS 213 – Programming Language 2

Dr. Ahmed Hesham Mostafa

Lecture 4 – OOP Part 2

Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*. Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

PolymorphismDemo

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

DynamicBindingDemo

Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compile error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of
  Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

CastingDemo

```java
public class CastingDemo {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two objects
    Object object1 = new CircleFromSimpleGeometricObject(1);
    Object object2 = new RectangleFromSimpleGeometricObject(1, 1);
    // Display circle and rectangle
    displayObject(object1);
    displayObject(object2);
  }
  /** A method for displaying an object */
  public static void displayObject(Object object) {
    if (object instanceof CircleFromSimpleGeometricObject) {
      System.out.println("The circle area is " +
          ((CircleFromSimpleGeometricObject)object).getArea());
      System.out.println("The circle diameter is " +
          ((CircleFromSimpleGeometricObject)object).getDiameter());
    }
    else if (object instanceof
        RectangleFromSimpleGeometricObject) {
      System.out.println("The rectangle area is " +
          ((RectangleFromSimpleGeometricObject)object).getArea());
    }
  }
}
```

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return this == obj;
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

NOTE

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

The ArrayList Class

You can create an array to store objects. But the array’s size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(o: E) : void`	Appends a new element `o` at the end of this list.
`+add(index: int, o: E) : void`	Adds a new element `o` at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element `o`.
`+get(index: int) : E`	Returns the element from this list at the specified index.
`+indexOf(o: Object) : int`	Returns the index of the first matching element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object) : int`	Returns the index of the last matching element in this list.
`+remove(o: Object): boolean`	Removes the element `o` from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int) : boolean`	Removes the element at the specified index.
`+set(index: int, o: E) : E`	Sets the element at the specified index.

Generic Type

ArrayList is known as a generic class with a generic type E. You can specify a concrete type to replace E when creating an ArrayList. For example, the following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();

ArrayList<String> cities = new ArrayList<>();
```

TestArrayList

Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

DistinctNumbers

Array Lists from/to Arrays

Creating an ArrayList from an array of objects:

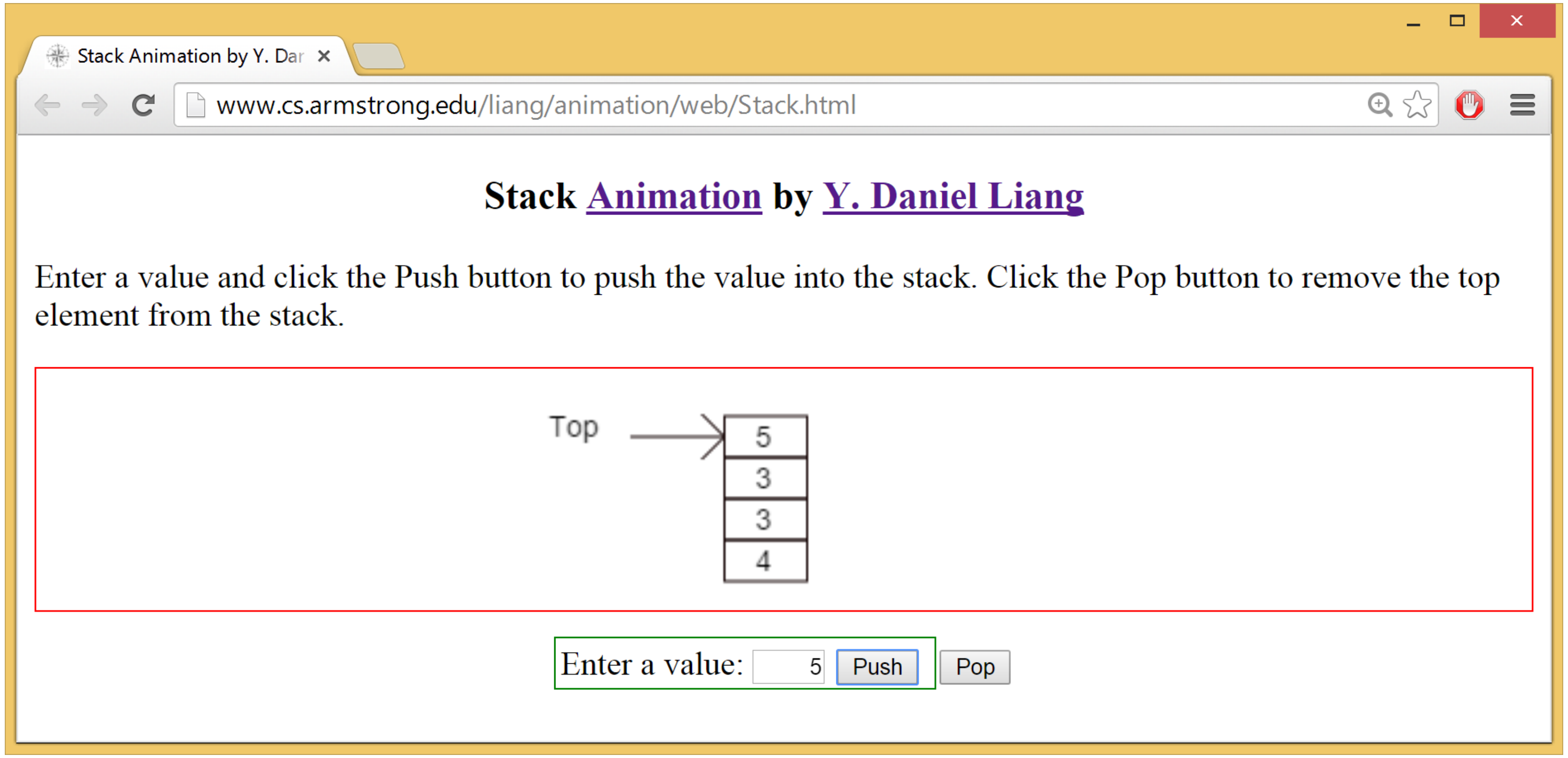
```
String[] array = {"red", "green", "blue"};
  ArrayList<String> list = new
ArrayList<>(Arrays.asList(array));
```

Creating an array of objects from an ArrayList:

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

Stack Animation

https://liveexample.pearsoncmg.com/dsanimation/StackeBook.html


Stack Animation by Y. Dar

www.cs.armstrong.edu/liang/animation/web/Stack.html

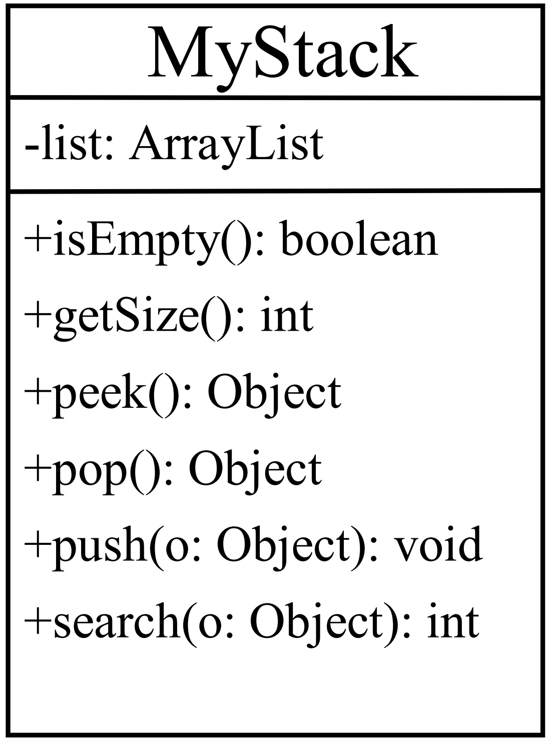
Stack Animation by Y. Daniel Liang

Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack.

The MyStack Classes

A stack to hold objects.

MyStack



MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

```java
import java.util.ArrayList;
public class MyStack {
   private ArrayList<Object> list = new ArrayList<>();
   public boolean isEmpty() {
      return list.isEmpty();
   }
   public int getSize() {
      return list.size();
   }
   public Object peek() {
      return list.get(getSize() - 1);
   }
   public Object pop() {
      Object o = list.get(getSize() - 1);
      list.remove(getSize() - 1);
      return o;
   }
   public void push(Object o) {
      list.add(o);
   }
   @Override /** Override the toString in the Object class */
   public String toString() {
      return "stack: " + list.toString();
   }}
```

```java
public class Main {
   public static void main(String[] args) {
      MyStack obj=new MyStack();
      obj.push(5);
      obj.push("ahmed");
      obj.push(3.7);
      obj.push('a');
      while(!obj.isEmpty()){
         System.out.println(obj.pop());
      }
   }
}
```

```
a
3.7
ahmed
5
```

The `protected` Modifier

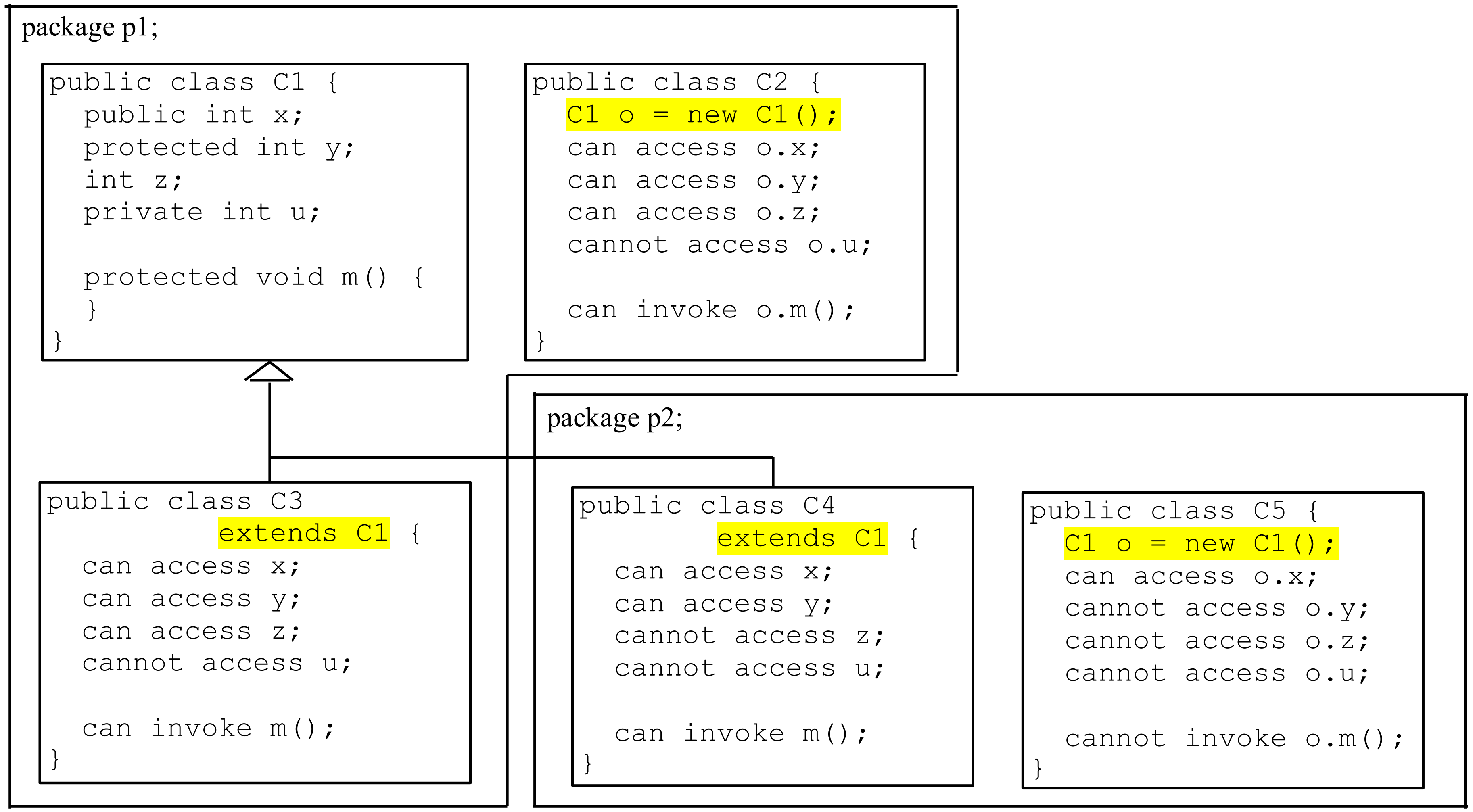
- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public

```
             Visibility increases
            ------------------------>
private, none (if no modifier is used), protected, public
```

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



package p1;

```
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3
          extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

package p2;

```
public class C4
          extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
    ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

abstract method in abstract class

An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

object cannot be created from abstract class

An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

abstract class without abstract method

A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

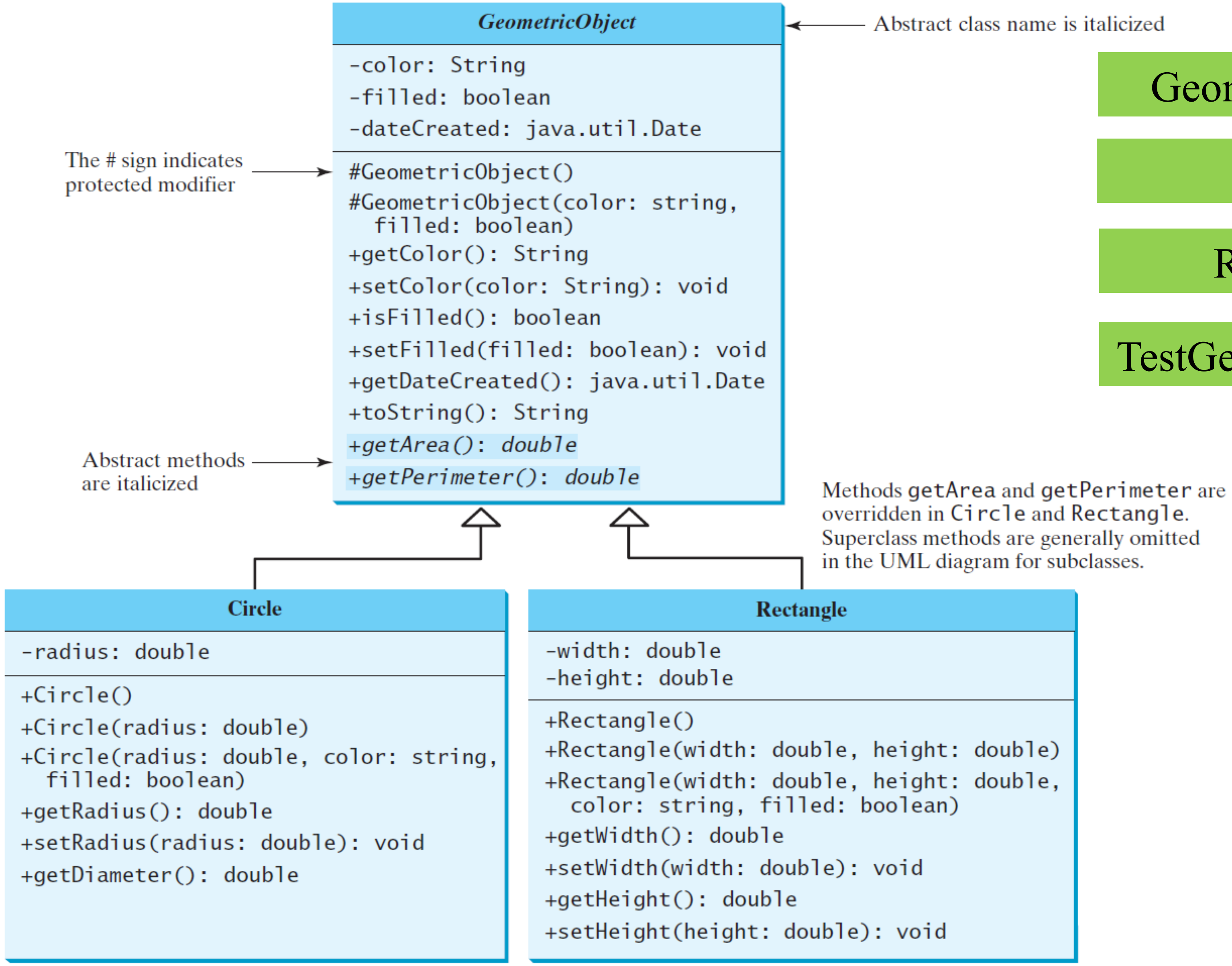
A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

abstract class as type

You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new    GeometricObject[10];
```

Abstract Classes and Abstract Methods



GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
Abstract class name is italicized
The # sign indicates protected modifier
Abstract methods are italicized
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

GeometricObject

Circle

Rectangle

TestGeometricObject

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

     protected GeometricObject() {
        dateCreated = new java.util.Date();
    }

    protected GeometricObject(String color, boolean filled)
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    public String getColor() {
        return color;
    }
public void setColor(String color) {
        this.color = color;
    }
```

```java
public boolean isFilled() {
        return filled;
    }

    public void setFilled(boolean filled) {
        this.filled = filled;
    }

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

    @Override
    public String toString() {
        return "created on " + dateCreated + "\ncolor: " + color +
                " and filled: " + filled;
    }

    public abstract double getArea();

    public abstract double getPerimeter();
}
```

```java
public class Circle extends GeometricObject {
   private double radius;
   public Circle() {
   }
   public Circle(double radius) {
      this.radius = radius;
   }
   public double getRadius() {
      return radius;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   @Override /** Return area */
   public double getArea() {
      return radius * radius * Math.PI;
   }
   public double getDiameter() {
      return 2 * radius;
   }
   @Override /** Return perimeter */
   public double getPerimeter() {
      return 2 * radius * Math.PI;
   }
   public void printCircle() {
      System.out.println("The circle is created " + getDateCreated() +
            " and the radius is " + radius);
   }
}
```

```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;
   }
   public double getWidth() {
      return width;
   }
  public void setWidth(double width) {
      this.width = width;
   }
   public double getHeight() {
      return 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);
   }
}
```

```java
public class Main {
   public static void main(String[] args) {
       GeometricObject geoObject1 = new Circle(5);
      GeometricObject geoObject2 = new Rectangle(5, 3);
      System.out.println("The two objects have the same area? " +
            equalArea(geoObject1, geoObject2));
      displayGeometricObject(geoObject1);
     displayGeometricObject(geoObject2);
   }

  public static boolean equalArea(GeometricObject object1,
                                GeometricObject object2) {
    return object1.getArea() == object2.getArea();
  }
  public static void displayGeometricObject(GeometricObject object) {
    System.out.println();
    System.out.println("The area is " + object.getArea());
    System.out.println("The perimeter is " + object.getPerimeter());
  }
  }
```

```
The two objects have the same area? false

The area is 78.53981633974483
The perimeter is 31.41592653589793

The area is 15.0
The perimeter is 16.0
```

Interfaces

What is an interface?

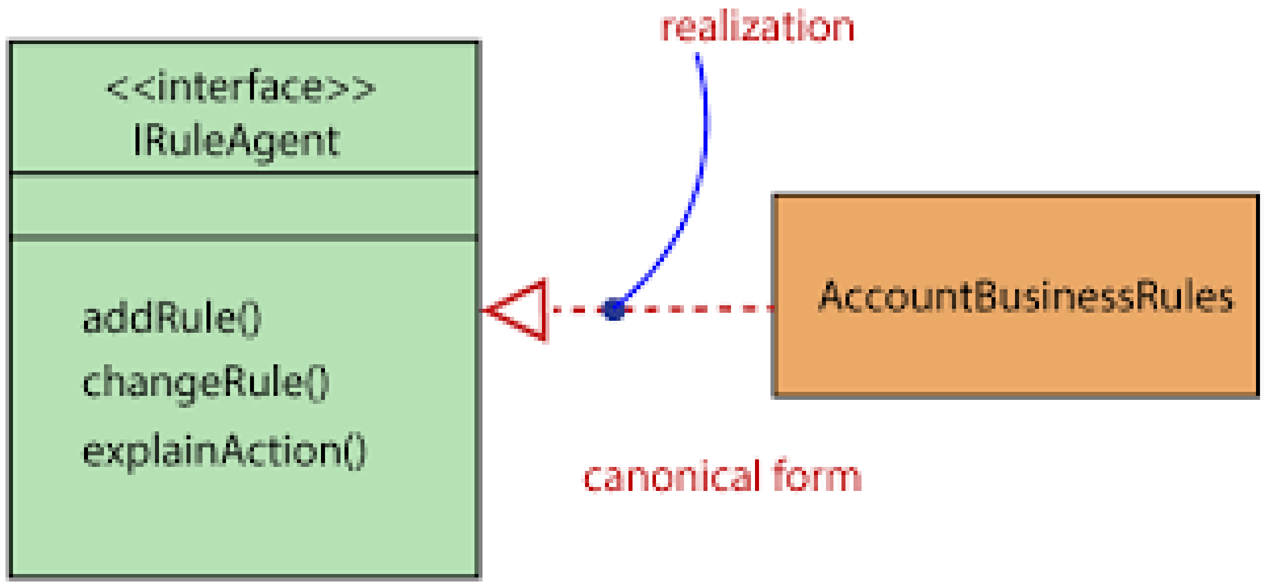
Why is an interface useful?

How do you define an interface?

How do you use an interface?

Realization

- Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.



What is an interface?
Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  abstract method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interface is a Special Class

An interface is treated like a special class in Java.

Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class.

For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```java
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

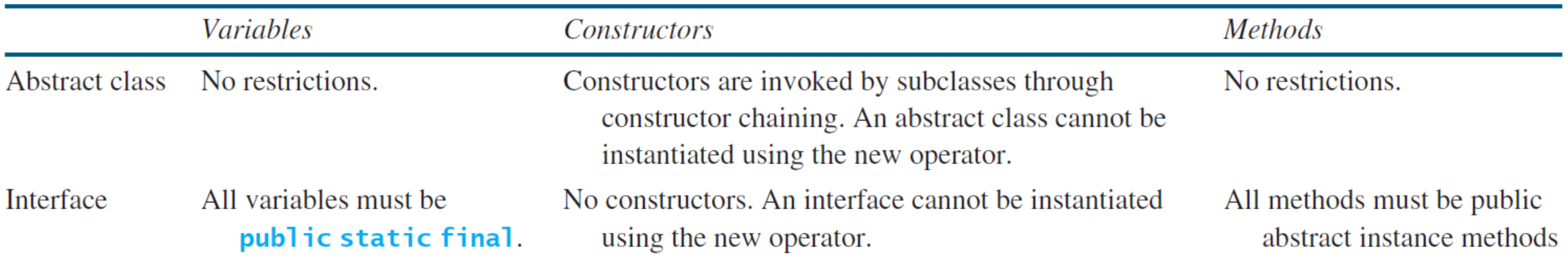
```java
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

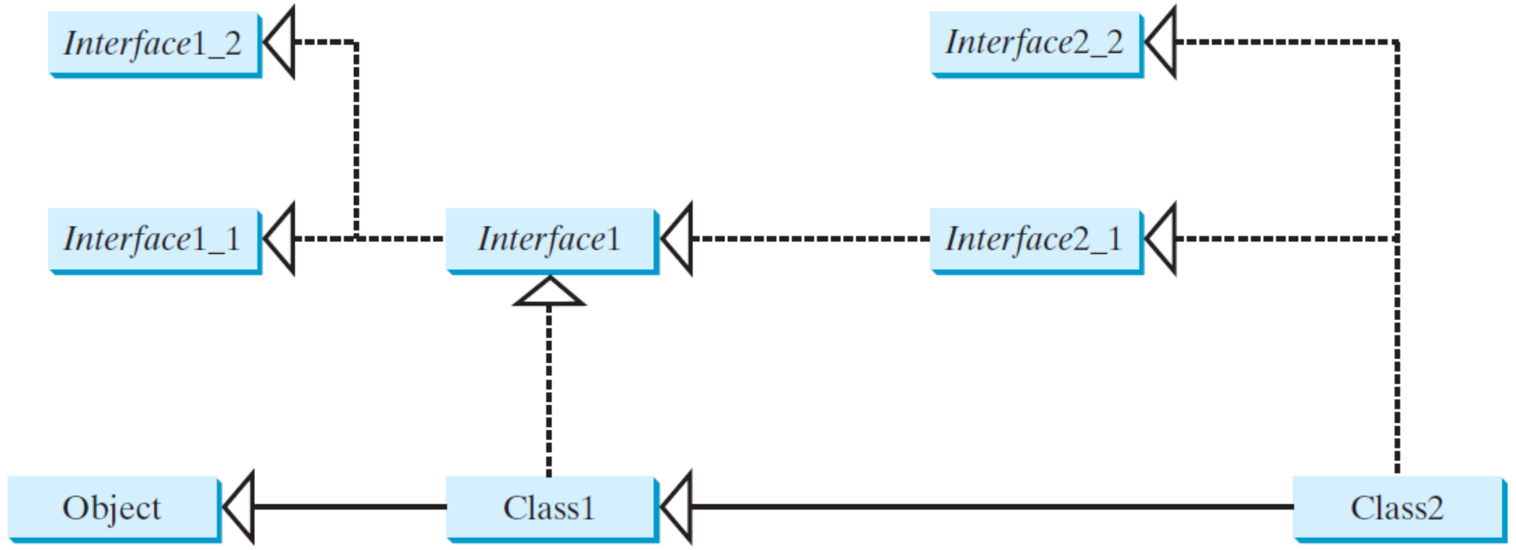


	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

48

Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Interface Example

```
public interface Employee {
    public double calculateSalary(double workedHours);
}
```

Interface Example

```
public class Payroll {
    Employee employee;
    public Payroll(Employee employee) {
        this.employee = employee;
    }

    public double calculateSalary(double workedHours){
        return this.employee.calculateSalary(workedHours);
    }
}
```

Interface Example

```
public class Developer implements Employee{
   private double workedHourCost;
   public double getWorkedHourCost() {
      return workedHourCost;
   }
   public void setWorkedHourCost(double workedHourCost) {
      this.workedHourCost = workedHourCost;
   }
   @Override
   public double calculateSalary(double workedHours) {
      return workedHours*getWorkedHourCost();
   }
}
```

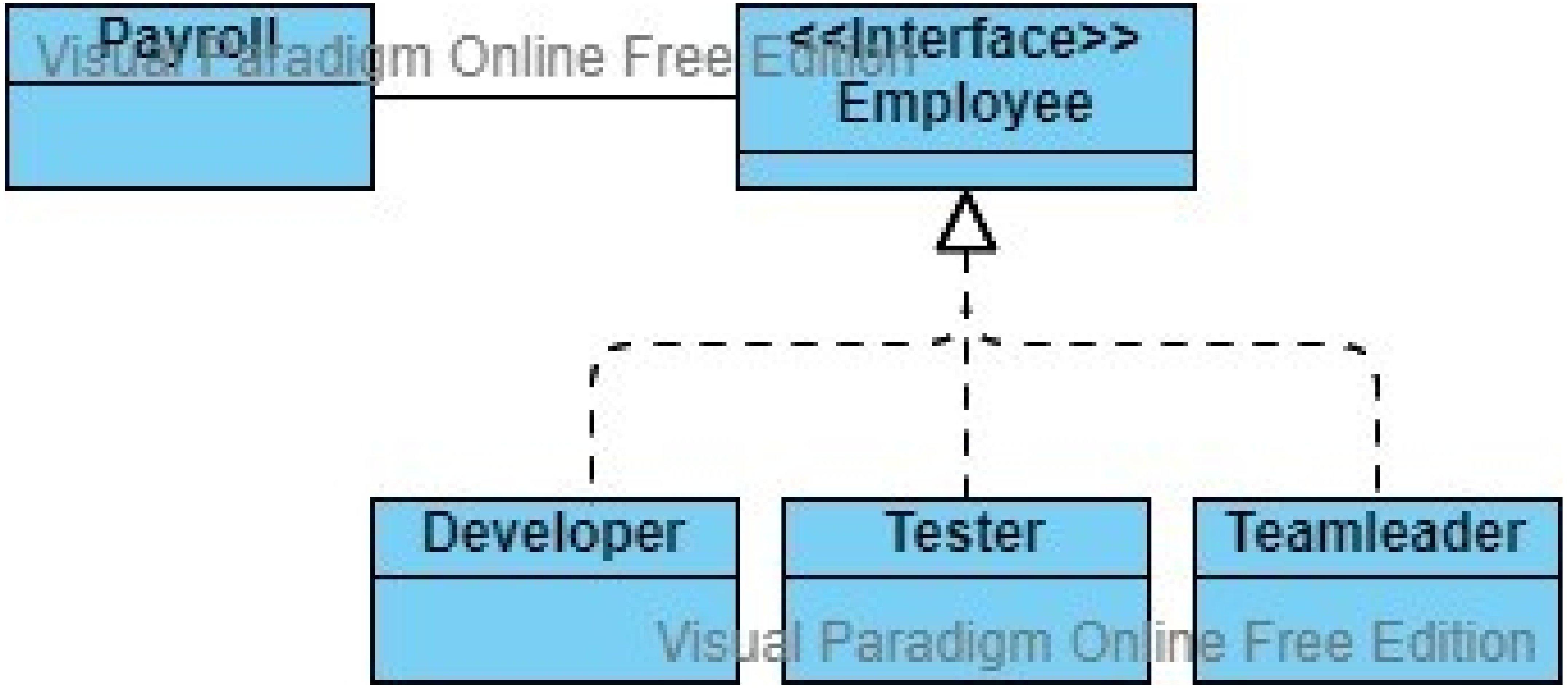
Interface Example

```
public class Tester implements Employee{
    private double workedHourCost;
    public double calculateBounce(double hoursworked){
        if(hoursworked>100)return 500;
        return 0;
    }
    public double getWorkedHourCost() {
        return workedHourCost;
    }
    public void setWorkedHourCost(double workedHourCost) {
        this.workedHourCost = workedHourCost;
    }
    @Override
    public double calculateSalary(double workedHours) {
        return workedHours*getWorkedHourCost();
    }
}
```

Interface Example

```
public class TeamLeader implements Employee{
    private double workedHourCost;
    public double getWorkedHourCost() {
        return workedHourCost;
    }

    public void setWorkedHourCost(double workedHourCost) {
        this.workedHourCost = workedHourCost;
    }
    @Override
    public double calculateSalary(double workedHours) {
        return workedHours*getWorkedHourCost();
    }
}
```

Interface Example



Interface Example

```
public class Main {
   public static void main(String[] args) {
      Developer ahmed=new Developer();
      ahmed.setWorkedHourCost(12);
      TeamLeader hesham=new TeamLeader();
      hesham.setWorkedHourCost(20);
      Tester mostafa=new Tester();
      mostafa.setWorkedHourCost(8);

      System.out.println("Dev salary = "+ new Payroll(ahmed).calculateSalary(100));
      System.out.println("Leader salary = "+new Payroll(hesham).calculateSalary(100));
      System.out.println("Tester salary = "+ new Payroll(mostafa).calculateSalary(100));
   }
}
```

```
Dev salary = 1200.0
Leader salary = 2000.0
Tester salary = 800.0
```

Thanks

References

- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- This slides based on slides provided by Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang