# Object Oriented Development with Java

(CT038-3-2 and Version VC1)

ASIA PACIFIC UNIVERSITY OF TECHNOLOGY & INNOVATION

## Abstract Classes
## Interfaces

Object Oriented Programming

# Topic & Structure of the lesson

- Abstract Classes & Methods
  - Example
- Interfaces
  - Example
- Interfaces vs. Abstract Classes

# Learning outcomes

- At the end of this lecture you should be able to:
  - Understand the concept of abstract classes
  - Understand the concept of interfaces
  - Distinguish between abstract classes and interfaces

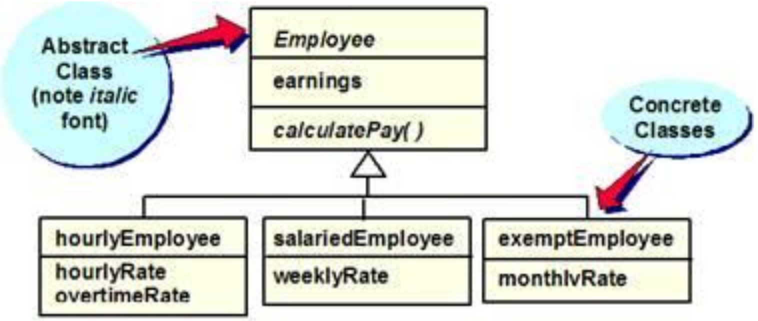
# Key terms you must be able to use

If you have mastered this topic, you should be able to use the following terms correctly in your assessments:

- Interface
- abstract

# Abstract Classes vs. Concrete Classes

- Abstract classes
  - – Are superclasses (called abstract superclasses)
  - – Cannot be instantiated (but can be subclassed)
  - – Incomplete
    - subclasses fill in "missing pieces"
- Concrete classes
  - – Can be instantiated
  - – Implement every method they declare
  - – Provide specifics



# Abstract Classes

- Abstract classes not required, but reduce client code dependencies
- To make a class abstract
  - Declare with keyword `abstract`
- Application example
  - Abstract class `Shape`
    - Declares `draw` as abstract method
  - `Circle`, `Triangle`, `Rectangle` extends `Shape`
    - Each must implement `draw`

# Abstract Methods

- An abstract class may or may not contain *abstract methods*

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

Abstract classes have a ; instead of a { }

- Abstract methods are declared without an implementation (ie. no method body), must be overridden
- An abstract method cannot exist without an abstract class
- A subclassed abstract class must provide implementation for abstract methods in

# Abstract Classes & Methods Example

```java
// Shape.java
// Shape abstract-superclass declaration.

public abstract class Shape extends Object {

// return area of shape; 0.0 by default
public double getArea(){
   return 0.0;
}

// return volume of shape; 0.0 by default
public double getVolume() {
   return 0.0;
}

// abstract method, overridden by subclasses
public abstract String getName();

} // end abstract class Shape
```

Abstract class

Abstract method

# Abstract Classes & Methods Example

```
// Point.java
// Point class declaration inherits from Shape.

public class Point extends Shape {
    private int x;  // x part of coordinate pair
    private int y;  // y part of coordinate pair

// no-argument constructor; x and y default to 0
public Point() {
// implicit call to Object constructor occurs here
}

// constructor
public Point(int xValue, int yValue) {
// implicit call to Object constructor occurs here
    x = xValue;  // no need for validation
    y = yValue;  // no need for validation
}

// set x in coordinate pair
public void setX(int xValue) {
   x = xValue;  // no need for validation
}
```

# Abstract Classes & Methods Example

```java
// return x from coordinate pair
public int getX() {
   return x;
}

// set y in coordinate pair
public void setY(int yValue) {
   y = yValue;  // no need for validation
}

// return y from coordinate pair
public int getY() {
   return y;
}

// override abstract method getName to return "Point"
public String getName() {
   return "Point";
}

// override toString to return String representation of Point
public String toString(){
   return "[" + getX() + ", " + getY() + "]";
}
} // end class Point
```

Override abstract method from superclass

# Abstract Classes & Methods Example

```
// Circle.java
// Circle class inherits from Point.

public class Circle extends Point {
   private double radius;  // Circle's radius

// no-argument constructor; radius defaults to 0.0
public Circle(){
// implicit call to Point constructor occurs here
}

// constructor
public Circle(int x, int y, double radiusValue) {
   super( x, y );  // call Point constructor
   setRadius( radiusValue );
}

// set radius
public void setRadius(double radiusValue) {
   radius = (radiusValue < 0.0 ? 0.0 : radiusValue);
}
```

# Abstract Classes & Methods Example

```
// return radius
public double getRadius(){
   return radius;
}

// calculate and return diameter
public double getDiameter() {
   return 2 * getRadius();
}

// calculate and return circumference
public double getCircumference(){
   return Math.PI * getDiameter();
}

// override method getArea to return Circle area
public double getArea() {
   return Math.PI * getRadius() * getRadius();
}
```

# Abstract Classes & Methods Example

```
// override abstract method getName to return "Circle"
public String getName(){
   return "Circle";
}

// override toString to return String representation of Circle
public String toString(){
   return "Center = " + super.toString() + "; Radius = " +
                getRadius();
}

} // end class Circle
```

Override abstract method from superclass

# Abstract Classes & Methods Example

```
// HierarchyRelationshipTest1.java
// Assigning superclass and subclass references to superclass- and
// subclass-type variables
import javax.swing.JOptionPane;

public class HierarchyRelationshipTest1 {

public static void main(String[] args) {
// assign superclass reference to superclass-type variable
Point3 point = new Point3( 30, 50 );

// assign subclass reference to subclass-type variable
Circle4 circle = new Circle4( 120, 89, 2.7 );

// invoke toString on superclass object using superclass variable
String output = "Call Point3's toString with superclass" +
" reference to superclass object: \n" + point.toString();

// invoke toString on subclass object using subclass variable
output += "\n\nCall Circle4's toString with subclass" +
" reference to subclass object: \n" + circle.toString();
```

Assign superclass reference to superclass-type variable

Assign subclass reference to subclass-type variable

# Abstract Classes & Methods Example

```
// invoke toString on subclass object using superclass variable
Point3 pointRef = circle;

output += "\n\nCall Circle4's toString with superclass" +
" reference to subclass object: \n" + pointRef.toString();

JOptionPane.showMessageDialog( null, output );  // display output

System.exit( 0 );

} // end main

} // end class HierarchyRelationshipTest1
```

# Interfaces

- Use *interface types* to make code more reusable
- In Java, an *interface type* is used to specify required operations
- Interface declaration lists all methods that the interface type requires
- Contain *only* constants, method signatures, default methods, static methods, and nested types
- Use `interface` keyword to create an

# Interfaces

- May be *implemented* by classes or *extended* by other interfaces
- A class that implements an interface must implement ***all*** of the interface's methods, unless if the class is defined as *abstract*
- Use `implements` keyword to indicate that a class implements an interface type

# Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- All methods in an interface type are abstract; they don't have an implementation
- All methods in an interface type are automatically `public`
- Attribute of interface is `public`, `static` and `final`
- An interface type does not have instance fields (no constructor)
- An interface can’t extend any class but it can

# Syntax: Defining an Interface

```
 public interface InterfaceName {
   // method
 }
```

Example:

```
  public interface Measurable {
    double getMeasure();
}
```

Purpose:
To define an interface and its method.
The methods are automatically public.

# Syntax: Implementing an Interface

```
public class ClassName implements InterfaceName,
InterfaceName, ...
{
      // methods
      // instance variables
}

Example:
public class BankAccount implements Measurable
{
 // Other BankAccount methods
     public double getMeasure()
     {
          // Method implementation
  }
}
```

Purpose:
To define a new class that implements the methods of an interface

# Interfaces vs. Abstract Classes

| Abstract Classes | Interfaces |
|---|---|
| Both *cannot* be instantiated | |
| Both may contain a mix of methods declared with or without an implementation | |

# Interfaces vs. Abstract Classes

| Abstract Classes | Interfaces |
|---|---|
| Can declare fields that are *not* static and final, and define public, protected, and private concrete methods | All fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public |
| Can extend only one class | Any number of interfaces may be implemented |
| Abstract class can be inherited by a class or an abstract class | Interfaces can be extended only by interfaces. Classes has to implement them instead of extend |
| The keyword 'abstract' is mandatory to declare a method as an abstract | The keyword 'abstract' is optional to declare a method as an abstract because all the methods are abstract by default |

# Interfaces vs. Abstract Classes When to use?

| Abstract Classes | Interfaces |
|---|---|
| To share code among several closely related classes | Unrelated classes implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes |
| Classes that extend the abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private). | To specify the behavior of a particular data type, but not concerned about who implements its behavior. |
| When using non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong. | To take advantage of multiple inheritance. |

# Quick Review Questions

- What is an abstract class?
- What is an interface?
- Discuss the similarities and differences between interfaces and abstract classes
- When would it be more appropriate to use interfaces and abstract classes?

# Summary of Main Teaching Points

- – **Abstract**
- – **Interface**

# Q & A

# Next Session

- Polymorphism
  - -Overloading
  - -Overriding

Encapsulation