CMPS 251

Read Chapter 10

Lecture 10

Java Abstract Class and Interface

Abstract Classes

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user (what it offers, not how it offers)
- **Abstract class is** a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method can only** be used in an abstract class, and it does not have a body, that must be provided by the subclass (inherited from).
- Use an abstract class when you want to define a **template** to guarantee that all **subclasses** in a hierarchy will have certain common methods
- An **abstract class** can have **both** abstract and regular methods
- Syntax

```
public abstract class SomeClass {
    public abstract SomeType method1(…);   // No body
    public SomeType method2(…) { … } // Not abstract    }
```

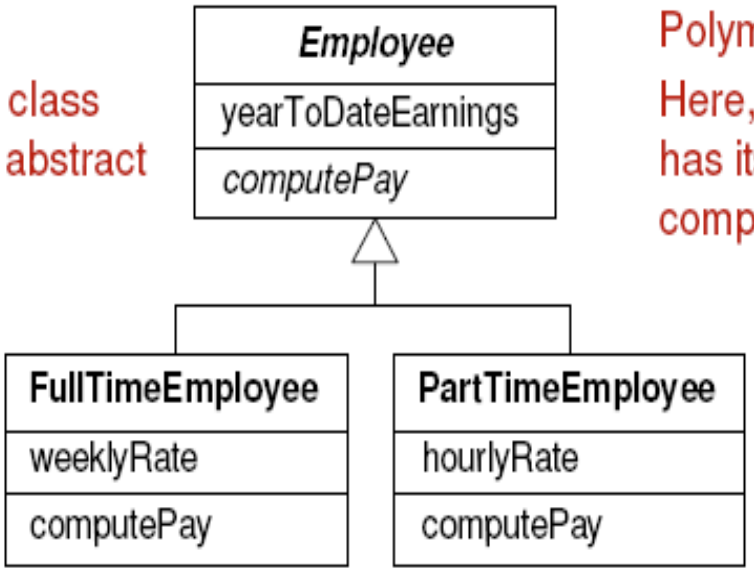
- Motivation
  - Guarantees that all subclasses will have certain methods => **enforce a common design**.
- There are two ways to achieve abstraction in java
  - **Abstract class** (0 to 100%)
  - **Interface** (100%)

Abstract Classes

- An abstract class has one or more abstract methods that subclasses can override
  - Abstract methods do not provide implementations because they **cannot be implemented in a general way**
  - Constructors and static methods cannot be declared abstract
- An abstract class cannot be instantiated
- An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method

Abstraction:

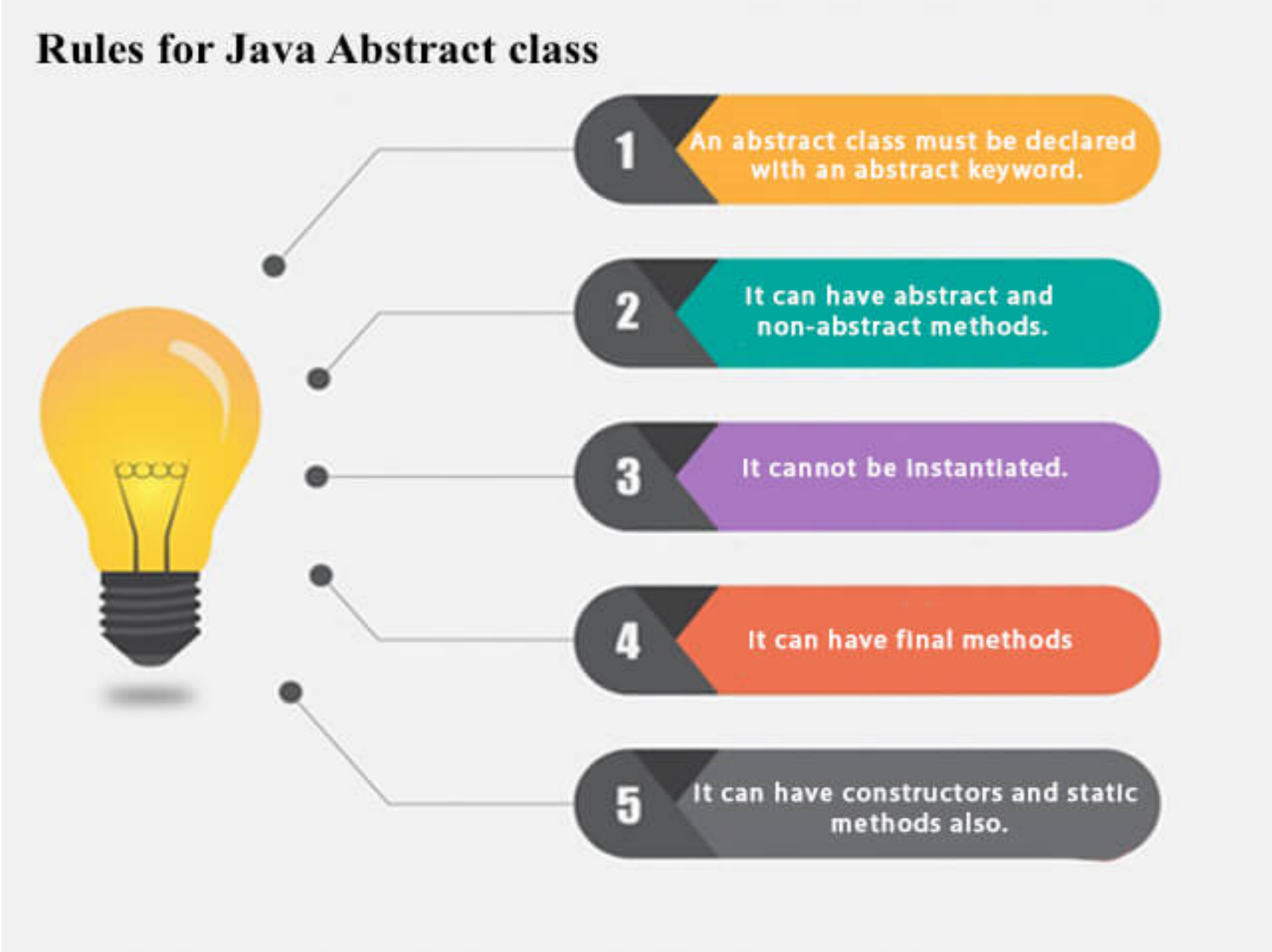
Employee is an abstract class and *computePay()* is an abstract operation (italicized)



Polymorphism:

Here, each type of Employee has its own version of computePay()

Rules for Java Abstract Class



Abstract Class Example

Shape.java

```
public abstract class Shape {

    public abstract double getArea();

    public abstract String getName();
```

Shape myObj = new Shape(); // will generate an error

Rectangle.java

```
public class Rectangle extends Shape{
    private double width;
    private double height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

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

    @Override
    public String getName() {
        return "Rectangle";
    }
}
```

Abstract Class Example

Shape.java

```
public abstract class Shape {

    public abstract double getArea();

    public abstract String getName();

}
```

Circle.java

```
public class Circle extends Shape {
    private double r;

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

    @Override
    public double getArea() {
        return Math.PI * r * r;
    }

    @Override
    public String getName() {
        return "Circle";
    }
}
```

Example Using Abstract Classes + Polymorphism

- You have Circle and Rectangle classes, each with getArea methods
- Goal: Get sum of areas of an array of Circles and Rectangles

=> Declare an array using an abstract class ***Shape***

```
Shape[] shapes = { new Circle(…), new Rectangle(…) … };
double areaSum = 0;
for(Shape s : shapes)
          areaSum += s.getArea();
System.out.printf("Sum of area of all shapes %.2f ", areaSum);
```

Class Modifiers

- **Public** - publicly accessible
  - without this modifier, a class is only accessible within its own package
- **abstract** – cannot be instantiated
  - its abstract methods **must** be implemented by its subclass; otherwise that subclass must be declared `abstract` also
- **final** class cannot be extended (e.g., String class)
- **final** method in a superclass cannot be overridden in a subclass

Example-1 of Abstract Class and Subclasses

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  } }
// Subclass (inherit from Animal)
Class Cat extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The cat says: meow meow");
  } }
class Main {
  public static void main(String[] args) {
    Cat myCat = new Cat(); // Create a Cat object
    myCat.animalSound();
    myCat.sleep();
  } }
```

Example-2: Abstract Class and Subclasses

```
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");
}  }
class Circle1 extends Shape{
void draw(){ System.out.println("drawing circle");}
}
class TestAbstraction1{
public static void main(String args[]){
Shape s = new Circle1();
s.draw();  //print drawing circle
}
```

Why And When To Use Abstract Classes and Methods?

- To achieve security, hide certain details, show the important details of an object.
- Abstraction can also be achieved with Interfaces,

Interfaces

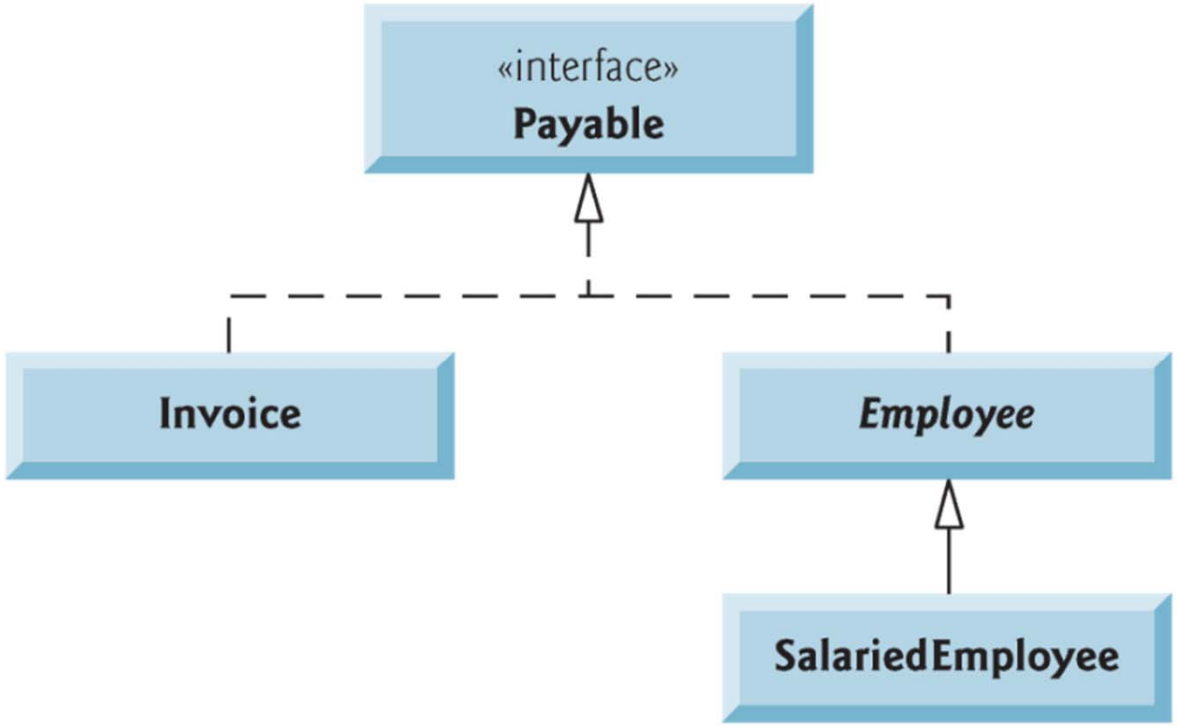
Back

Interfaces

- Another way to achieve abstraction in Java, is with interfaces.
- An interface is a completely "**abstract class**" that is used to group related methods with empty bodies
- **Interfaces** are used to define a set of common methods that must be implemented by **classes not related by inheritance**
- The interface specifies ***what*** operations a class must perform but does not specify ***how*** they are performed
- Interfaces enables requiring that **unrelated classes implement a set of common methods**
- **Benefit from polymorphism:** objects of unrelated classes that implement a certain interface can be **processed polymorphically**
- To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends).
- The body of the interface method is provided by the "implement" class

Interface Example

- A finance system has Employees and Invoices
- `Employee` and `Invoice` are not related by inheritance
- But to the company, they are both ***Payable***



Interface Example

Payable.java

```
public interface Payable {
double getPaymentAmount();
}
```

Employee.java

```
public class Employee implements Payable{
    private double salary;
    ...
    @Override
    public double getPaymentAmount() {
        return this.salary;
    }
    ...
}
```

Invoice.java

```
public class Invoice implements Payable {
   private double totalBills;
   ...
    @Override
    public double getPaymentAmount() {
        return this.totalBill;
    }
    ...
}
```

Example of Interface

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}
// Cat "implements" the Animal interface
Class Cat implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The cat says: meow meow");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
class Main {
  public static void main(String[] args) {
    Cat myCat = new Cat();  // Create a Cat object
    myCat.animalSound();
    myCat.sleep();
  }
```

Rules of Interface

- Like abs**tract classes, i**nterfaces can**not be** used to create objects (
  - In the example in the previous slide, it is not possible to create an "Animal" object)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)

Creating Interfaces

- An interface declaration begins with the keyword **interface** and contains only **constants** and **abstract methods**
  - All interface members must be public
  - All methods declared in an interface are **implicitly public abstract methods**
  - All attributes are implicitly public, static and final
- A class **implementing** the interface must declare each method in the interface with specified signature

Implementing an Interface

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        //code for driving car
    }

    @Override
    public void move() {
        this.drive();
    }

}
```

- **Car** implements **Transporter** interface
  - declare that Car "acts-as" Transporter
- Promises compiler that Car will define all methods in Transporter interface i.e., move()
- Method ***signature*** (name and number/type of parameters) must match how it's declared in interface
- Otherwise…

"Error: Car does not override method move() in Transporter"

Implementing an Interface

```
public class Car implements Transporter {

    public Car() {
        //code elided
    }
    public void drive() {
        //code elided
    }

    @Override
    public void move() {
        this.drive();
    }
    //more methods ...
}
```

```
public class Bike implements
Transporter {
    public Bike() {
        //code elided
    }
    public void pedal() {
        //code elided
    }

    @Override
    public void move() {
        this.pedal();
    }

    //more methods ...
}
```

@Override is an annotation – a signal to the compiler to enforce that the interface actually has the method declared

Implementing Multiple Interfaces

- Classes can implement **multiple** interfaces
  - "I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person."
  - The Car can implement both the Transporter and the Colorable interface
- Class implementing interfaces must define every single method from each interface

```
public interface Colorable {

    public void setColor(Color c);
    public Color getColor();

}

public class Car implements Transporter, Colorable {

    public Car(){ //body ... }

    public void drive(){ //body ... }
    public void move(){ //body ... }
    public void setColor(Color c){ //body ... }
    public Color getColor(){ //body ...  }
}
```

Example of Multiple Interface

```
interface FirstInterface {
  public void myMethod(); // interface method  }
interface SecondInterface {
  public void myOtherMethod(); // interface method
}
class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text");
  }
  public void myOtherMethod() {
    System.out.println("Some other text");
  }  }
class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();  } }
```

- Java does not support "multiple inheritance" (a class can only inherit from one superclass).
- However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.
- To implement multiple interfaces, separate them with a comma

Abstract Class vs. Interface

- Abstract classes and interfaces cannot be instantiated
- Abstract classes and interfaces may have abstract methods that must be implemented by the subclasses
- Classes that implement an interface **can be from different inheritance hierarchies**
  - An interface is often used when unrelated classes need to provide **common methods** or use common constants
  - When a class implements an interface, it establishes an ***IS-A*** relationship with the interface type. Therefore, interface references can be used to invoke polymorphic methods just as an abstract superclass reference can.
- Concrete subclasses that extend an abstract superclass are **all related to one other by inheriting from a shared superclass**
- Interfaces cannot define instance attributes and constructors
  - Interfaces can have abstract methods, methods with a default implementation, static methods and static constants.
- Classes can extend only ONE abstract class but they may implement more than one interface

Summary

- Inheritance = “factor out” the common attributes and methods and place them in a single superclass

  => Removing code redundancy will result in a smaller, more flexible program that is easier to maintain.

- Interfaces are contracts, can’t be instantiated
  - force classes that implement them to define specified methods
- Polymorphism allows for generic code by using superclass/interface type variables to manipulate objects of subclass type
  - make the client code more **generic** and ease extensibility

default Interface Methods

- Interfaces also may contain `public default methods` with concrete default implementations used when an implementing class does not override the methods.
- To declare a default method, place the keyword **default** before the method's return type and provide a concrete method implementation.
- Any class that implements the original interface will not break when a default method is added.
  - The class simply receives the new default method.
- Interfaces can also have static methods.

Question

Given the following class:

```
public class Laptop implements Typeable, Clickable { //two interfaces
   public void type() {
     // code elided
   }
   public void click() {
       //code elided
}
```

Given that `Typeable` has declared the `type()` method and `Clickable` has declared the `click()` method, which of the following calls is valid?

A.
```
Typeable macBook= new Typeable();
macBook.type();
```

B.
```
Clickable macBook = new Clickable();
macBook.type();
```

C.
```
Typeable macBook= new Laptop();
macBook.click();
```

D.
```
Clickable macBook = new Laptop();
macBook.click();
```