Advanced Programming Techniques in Java

COSI 12B

Polymorphism & Abstract Classes

Lecture 14

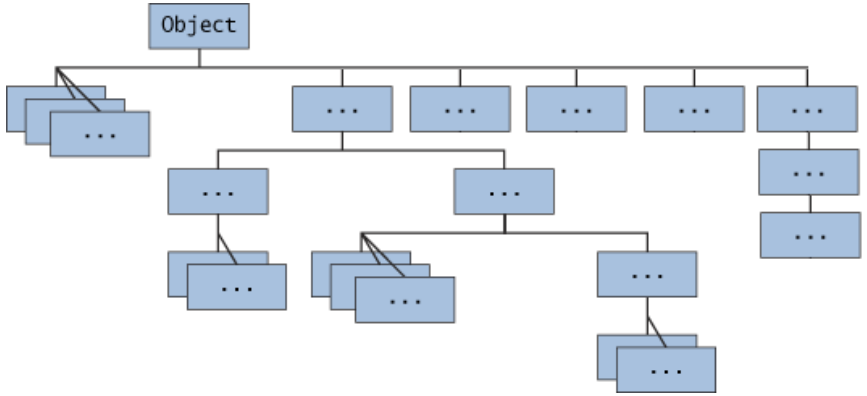
Class Objectives

- Polymorphism (section 9.3)
- Abstract Classes (last subsection of 9.6)
- Interfaces (section 9.5)

Review: Inheritance and Polymorphism

- **Inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior
- **Polymorphism**: Ability for an object to be used as if it was of different type

Review: The `Object` class



- The `Object` class is the parent class of all the classes in java
  - All classes are derived from the `Object` class (i.e. every class implicitly extends `Object`)
  - It defines and implement the behavior common to all classes
  - It is defined in the `java.lang` package
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

Review: Object Casting

- Casting allows the use of an object of one type in place of another type
  - It applies among the objects permitted by inheritance
- **Upcasting**: an object of a subclass type can be treated as an object of any superclass type
  - Upcasting is automatic in Java (**implicit casting**)
- **Downcasting**: treating a superclass object as its real subclass
  - Downcasting must be specified (**explicit casting**)

Review: Polymorphism and parameters

- You can pass any subtype of a parameter's type

```
public class EmployeeMain3 {
    public static void main(String[] args) {
        Lawyer law = new Lawyer();
        Secretary sec= new Secretary();
        printInfo(law);
        printInfo(sec);
    }

    public static void printInfo(Employee empl){
          empl.getSalary();
          empl.getVacationDays();
          empl.getVacationForm();
    }
}
```

You can pass both `Lawyer` and `Secretary` objects

Depending on the type you passed it calls the corresponding method

OUTPUT

```
I earn $40,000
I receive 3 weeks vacation
Use the pink vacation form

I earn $40,000
I receive 2 weeks vacation
Use the yellow vacation
form
```

Review: Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements

```
public class EmployeeMain4 {
    public static void main(String[] args) {
        Employee[] e = {new Lawyer(), new Secretary(),
                        new Marketer(), new Employee() };
        for (int i = 0; i < e.length; i++) {
            e[i].getSalary();
            e[i].getVacationDays();
            System.out.println();
        }
    }
}
```

You can store objects of different subtypes or of the superclass

You can only call methods of the `Employee` class
e.g., `empl.sue()` is illegal because it is a method of the `Lawyer` class

Output:

```
I earn $40,000
I receive 3 weeks vacation
I earn $40,000
I receive 2 weeks vacation
I earn $50,000
I receive 2 weeks vacation
I earn $40,000
I receive 2 weeks vacation
```

Lawer

Secretary

Marketer

Employee

Polymorphism problem

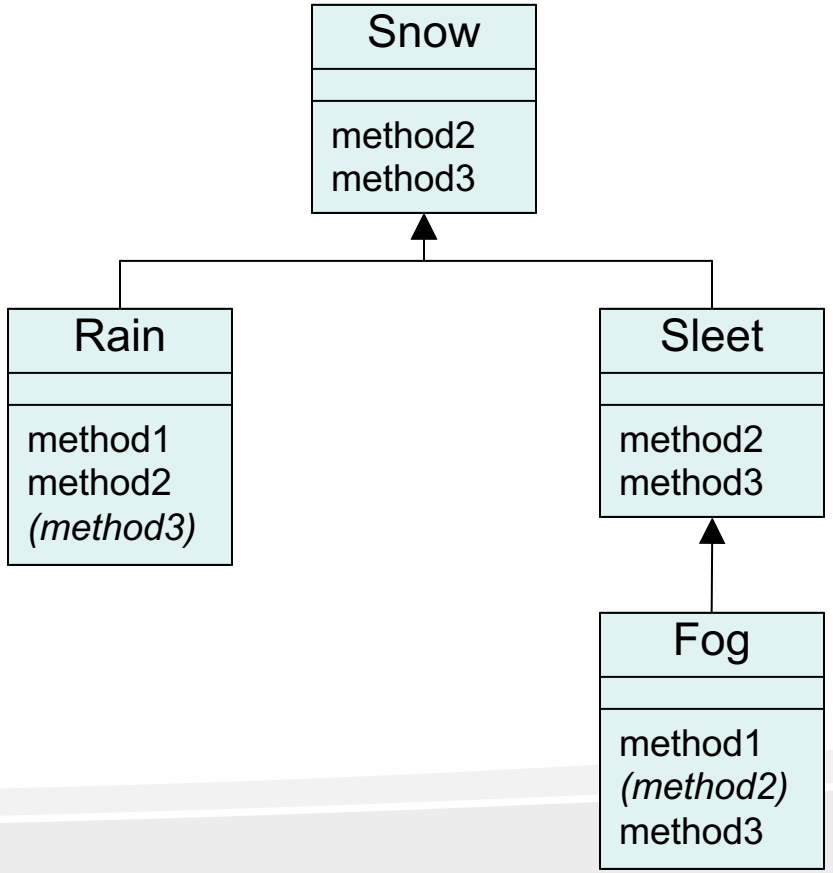
- 4-5 classes with inheritance relationships are shown
- A client program calls methods on objects of each class
- You must read the code and determine the client's output

```
public class Snow {
      public void method2() {
          System.out.println("Snow 2");
      }

      public void method3() {
          System.out.println("Snow 3");
      }
}
public class Rain extends Snow {
      public void method1() {
          System.out.println("Rain 1");
      }

      public void method2() {
          System.out.println("Rain 2");
      }
}
public class Sleet extends Snow {
    public void method2() {
        System.out.println("Sleet 2");
        super.method2();
        method3();
      }

      public void method3() {
          System.out.println("Sleet 3");
      }
}
public class Fog extends Sleet {
      public void method1() {
          System.out.println("Fog 1");
      }

      public void method3() {
          System.out.println("Fog 3");
      }
}
```

Problem2

Technique 1: diagram

- Diagram the classes from top (superclass) to bottom



Problem 2

- What happens when the following examples are executed?
  - Example 1:

```
Snow var1 = new Sleet();
var1.method2();
```

  - Example 2:

```
Snow var2 = new Rain();
var2.method1();
```

  - Example 3:

```
 Snow var3 = new Rain();
((Sleet) var3).method3();
```

  - Example 4:

```
Fog var4 = new Fog();
var4.method2();
```

```
public class Snow {
      public void method2() {
          System.out.println("Snow 2");
      }

      public void method3() {
          System.out.println("Snow 3");
      }
}
public class Rain extends Snow {
      public void method1() {
          System.out.println("Rain 1");
      }

      public void method2() {
          System.out.println("Rain 2");
      }
}
public class Sleet extends Snow {
    public void method2() {
        System.out.println("Sleet 2");
        super.method2();
        method3();
      }

      public void method3() {
          System.out.println("Sleet 3");
      }
}
public class Fog extends Sleet {
      public void method1() {
          System.out.println("Fog 1");
      }

      public void method3() {
          System.out.println("Fog 3");
      }
}
```

Problem 2

- Example 1:

```
Snow var1 = new Sleet();
var1.method2();
```

- Output:

```
Sleet 2
Snow 2
Sleet 3
```

Problem 2

- Example 2:

```
Snow var2 = new Rain();
var2.method1();
```

- Output:

  None!

  There is a (syntax) error, because `Snow` does not have a `method1`.

Problem 2

- Example 3:

```
Snow var3 = new Rain();
((Sleet) var3).method2();
```

- Output:

  None!

  There is a (runtime) error because a `Rain` is not a `Sleet`.

Problem 2

- Example 4

```
Fog var4 = new Fog();
var4.method2();
```

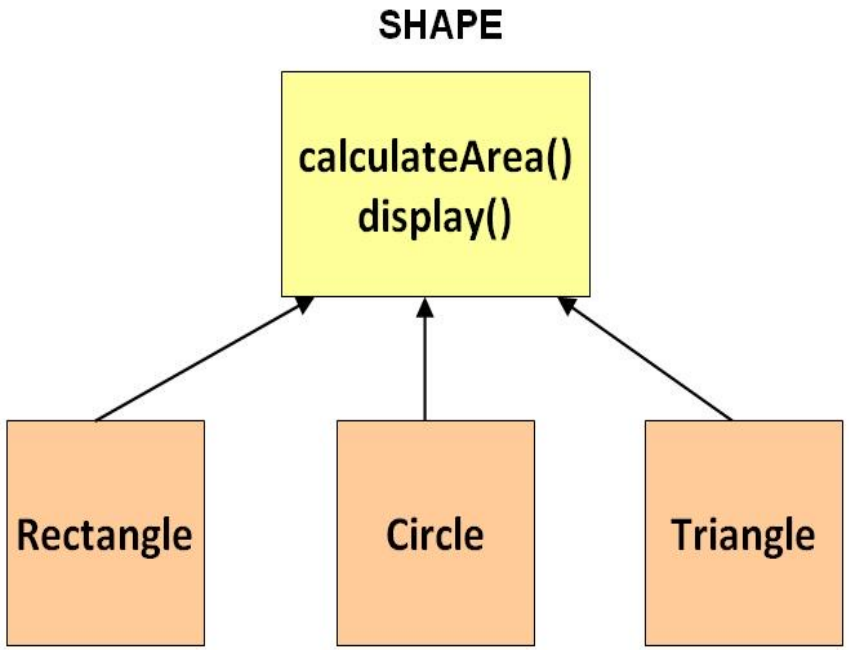
- Output:

```
Sleet 2
Snow 2
Fog 3
```

Abstract Classes

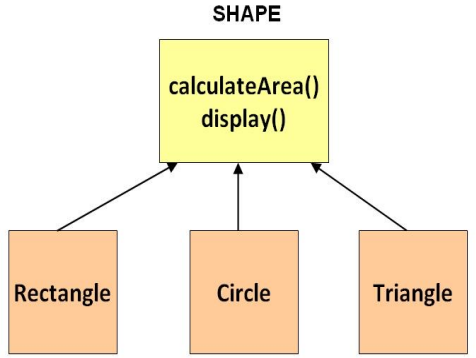
Abstract Classes

- Consider the following class hierarchy



- The `Shape` class is created to save on common attributes and methods shared by the `Rectangle`, `Circle`, and `Triangle` classes

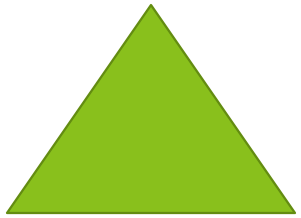
Abstract Classes



- Assume now that you write code to create objects for these classes:

```
Rectangle obj = new Rectangle();
```

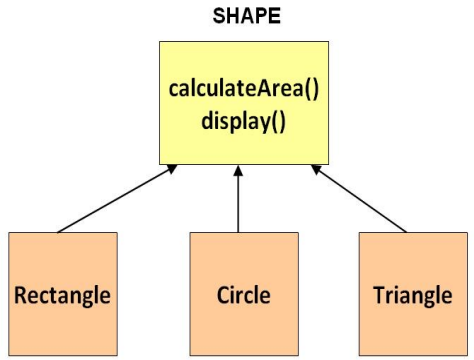
```
Triangle obj = new Triangle();
```



```
Shape obj = new Shape();
```

????

Abstract Classes



- Assume now that you write code to create objects for these classes:

```
Rectangle obj = new Rectangle();
```

```
Triangle obj = new Triangle();
```

```
Shape obj = new Shape();
```

????

- The Shape class serves in **achieving inheritance and polymorphism,** but it was not built to be instantiated
- **Abstract classes**

Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- Why?
  - The use of abstract classes is a design decision; it helps us establish common elements in a class that are too general to instantiate

Abstract Classes

- To declare a class as abstract we use the modifier `abstract` on the class header
- **Syntax**

```
public abstract class <name> {
    // contents
}
```

- **Example**

```
public abstract class Shape{
    // contents
}
```

- If the client code tries to create a `Shape` object, we get a compilation error

`Cannot instantiate the type Shape`

Abstract Classes

- Abstract classes can choose to provide implementation of some methods and not others
- Abstract classes can choose **not** to implement methods. This functionality is left to the subclasses
- How to choose not to implement a method?

abstract methods

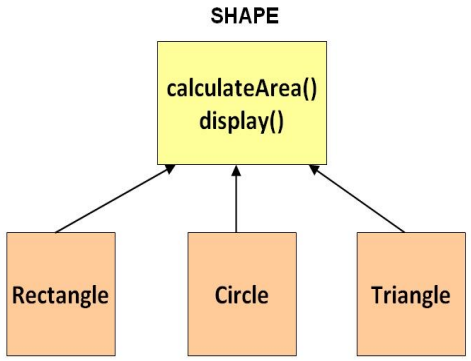
Abstract Method

- An abstract method is a method that has just the signature but does not contain implementation

```
public abstract <type> <name>(<type> <name>, ..);
```

- A class declared as `abstract` **does not** need to contain `abstract` methods

Why Abstract Method?



- The formula for calculating the area of a rectangle, circle, & triangle is different
- The `calculateArea()` methods in the `Shape` class needs to be overridden by the inheriting classes, thus it makes no sense writing it in the `Shape` class
- But we need to make sure that all the inheriting classes do have the method, such method is labeled `abstract`

Abstract Classes

```
public abstract class Shape{

   public void display(){
     System.out.println("This is a display method");
   }

   public abstract double calculateArea();
}
```

- An `abstract` method cannot be defined as `final` (because it must be overridden)
- The child of an `abstract` class must override the `abstract` methods of the parent
- Methods can call `abstract` methods

Abstract Classes and Inheritance

- `abstract` classes can be inherited
- Subclass of abstract class inherits `abstract` method from parent as well
  - Subclasses must provide implementation for inherited `abstract` method

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

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
    }
}
```

Abstract Classes & Constructors

- Can an abstract class have a constructor?
  - An abstract class can have a constructor. You can either explicitly provide a constructor to an abstract class or if you don't, the compiler will add a default constructor
- Why can an abstract class have a constructor?
  - When a class extends an abstract class, the constructor of subclass will invoke the constructor of super class either implicitly or explicitly

Abstract Classes & Constructors

```
public abstract class Fruit {
        private String color;
        private boolean seasonal;

        protected Fruit(String color, boolean seasonal) {
                this.color = color;
                this.seasonal = seasonal;
        }

        public abstract void prepare();

        public String getColor() {
                return color;
        }

        public boolean isSeasonal() {
                return seasonal;
        }

}
```

Abstract Classes & Constructors (cont.)

```
public class Mango extends Fruit {

        public Mango(String color, boolean seasonal) {
                super(color, seasonal);
        }
        @Override
        public void prepare() {
                System.out.println("Cut the Mango");
        }
}


public class Banana extends Fruit {

        public Banana(String color, boolean seasonal) {
                super(color, seasonal);
        }
        @Override
        public void prepare() {
                System.out.println("Peel the Banana");
        }
}
```

Abstract Classes & Constructors

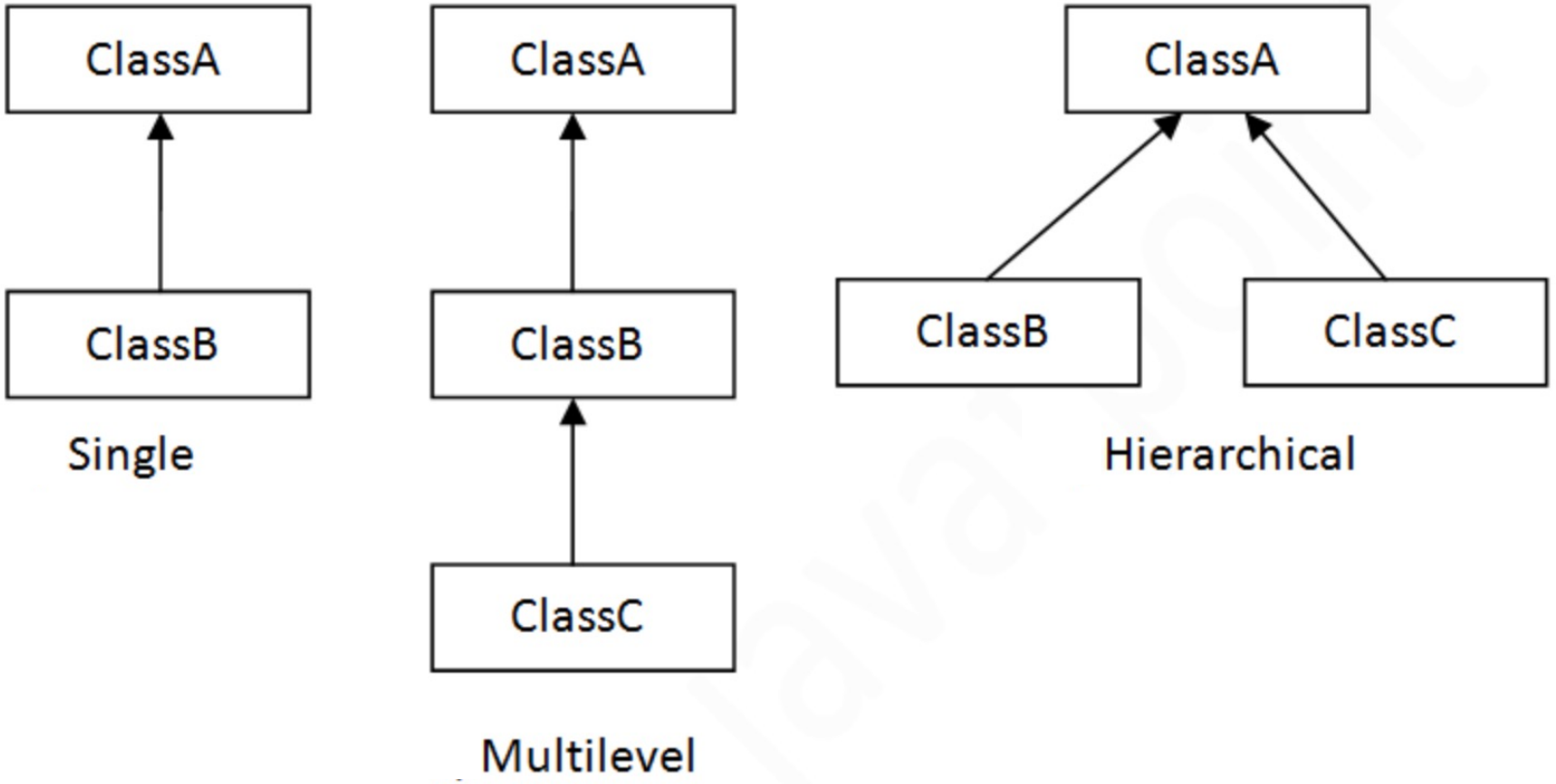
- You may define more than one constructor (with different arguments)
- You should define all your constructors `protected`
  - Making them public is pointless anyway

Abstract Classes in summary

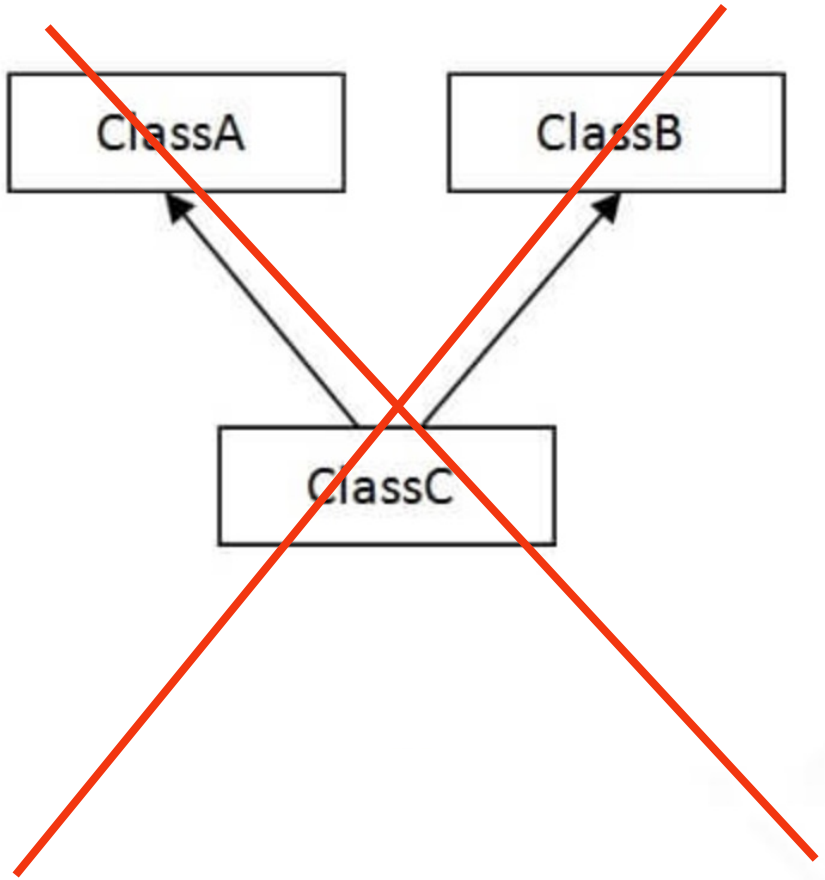
- The use of abstract classes is a design decision
- An abstract class must be declared with an `abstract` keyword
- It can have abstract and non-abstract methods
- It cannot be instantiated
- It can have constructors

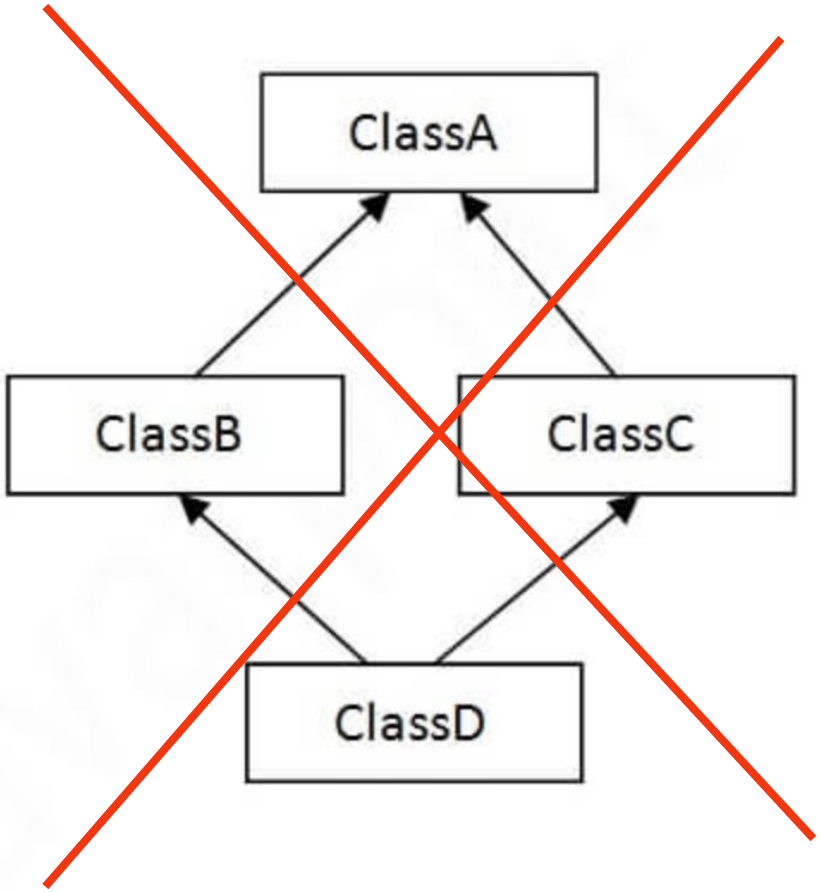
Interfaces

Types of inheritance in Java



Types of inheritance in Java





Multiple Inheritance

- Java supports single inheritance, meaning that a derived class can have only one parent class
- Multiple inheritance allows a class to be derived from two or more classes, inheriting the members of all parents
- **Java does not support multiple inheritance** because of possible conflicts
  - Which class should `super` refer when child class has multiple parents?
- Alternative: **Interface**
  - Looks like a class
  - It describes what a class does
- You can have a class that **extends** one class and **implements** one or more interfaces

What's an Interface?

- An Interface looks like a class, but it is not a class
  - Contains a list of methods that classes can promise to implement
- An interface can have methods just like the class, but the methods declared in interface are by default `abstract`
- A Java class can implement multiple Java Interfaces

- Inheritance gives you an is-a relationship and code-sharing
- Interfaces give you an is-a relationship without code sharing

Why Interfaces?

- They are used for full abstraction
  - Methods in interfaces do not have body, they must be implemented by the class before you can access them
  - The class that implements interface must implement all the methods of that interface

Interface Syntax

- Interface declaration, general syntax:

```
public interface <name> {
    public <type> <name>(<type> <name>, ..., <type> <name>);
    public <type> <name>(<type> <name>, ..., <type> <name>);
    ...
    public <type> <name>(<type> <name>, ..., <type> <name>);
}
```

Abstract methods

Interface Definition

FORM:

```
public interface interfaceName {
    abstract method headings
    constant declarations
}
```

EXAMPLE:

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final
           double DEDUCTIONS = 25.5;
}
```

- The keyword `abstract` is implicit in each *abstract method* definition
- And keywords `static final` are implicit in each *constant* declaration
- As such, they may be omitted

Shape Interface

- An interface for shapes:

```
// A general interface for shape classes
public interface Shape {
    public double area();
    public double perimeter();
}
```

- This interface describes the features common to all shapes (every shape has an area and perimeter)

Implementing an Interface

- **A class can declare that it implements an interface**
  - This means the class contains an implementation for each of the abstract methods in that interface
- Implementing an interface, general syntax:

```
public class <name> implements <interface name> {
          ...
}
```

- Example

```
public class Triangle implements Shape {
          ...
}
```

Interface Requirements

- If we write a class that claims to be a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile
- Example:

```
public class Banana implements Shape {

}
```

- The compiler error message:

```
Banana.java:1: Banana is not abstract and does not override abstract method area()
in Shape

public class Banana implements Shape {

           ^
```