Advanced Programming Techniques in Java

Polymorphism & Abstract Clas

Lecture 14

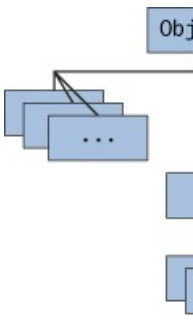
Class Objectives

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

- Interfaces (section 9.5)

Review: Inheritance and Polyr

- **Inheritance**: A way to form new classes based on e
attributes/behavior
- **Polymorphism**: Ability for an object to be used as i



Review: The `Object` class

- The `Object` class is the parent class of all the
- All classes are derived from the `Object` class (i.e.
- It defines and implement the behavior common to al
- It is defined in the `java.lang` package

- If a class is not explicitly defined to be the child
  be the child of the `Object` class

Review: Object Casting

- Casting allows the use of an object of one type in pl
- It applies among the objects permitted by inheritance

- **Upcasting**: an object of a subclass type can be trea
  - Upcasting is automatic in Java (**implicit casting**)

- **Downcasting**: treating a superclass object as its re
- Downcasting must be specified (**explicit casting**)

```
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();      }
}
```

Review: Polymorphism and para

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

Review: Polymorphism and arra

- Arrays of superclass types can store any subtype a

```
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();
        }
    }
}
```

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

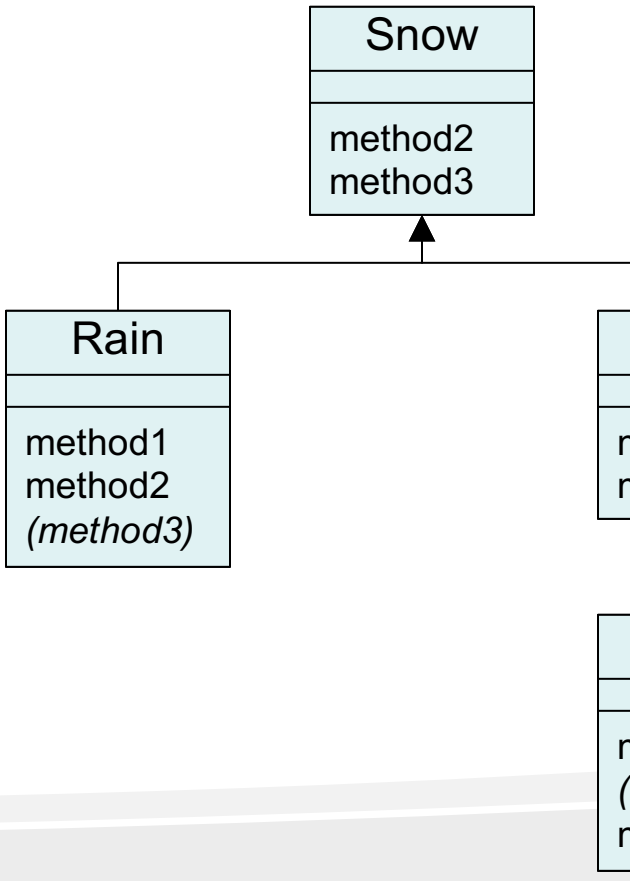
- 4-5 classes with inheritance relationships are show
- A client program calls methods on objects of each c
- You must read the code and determine the client's

```
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");
        }
}
```

Technique 1: diagram

- Diagram the classes from top (superclass) to bottom



```
public class
      void me
          Sys
      }

      public
          Sys
      }
}
public class
      public
          Sys
      }

      public
          Sys
      }
}
public class
    public vo
        Syste
        2");
        metho
      }

      public
          Sys
      }
}
public class
      public
          Sys
```

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
            Sys
        }
}
```

Problem 2

- Example 1: `Snow var1 = new Sleet(); var1`

- Output:

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

- Example 2:

Problem 2

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

- Output:

  None!

  There is a (syntax) error, because `Snow` does n
- Example 3:

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

Problem 2

- Output:

  None!

  There is a (runtime) error because a `Rain` is `Sleet`.

- Example 4 `Fog var4 = new Fog(); var4.me`

- Output:

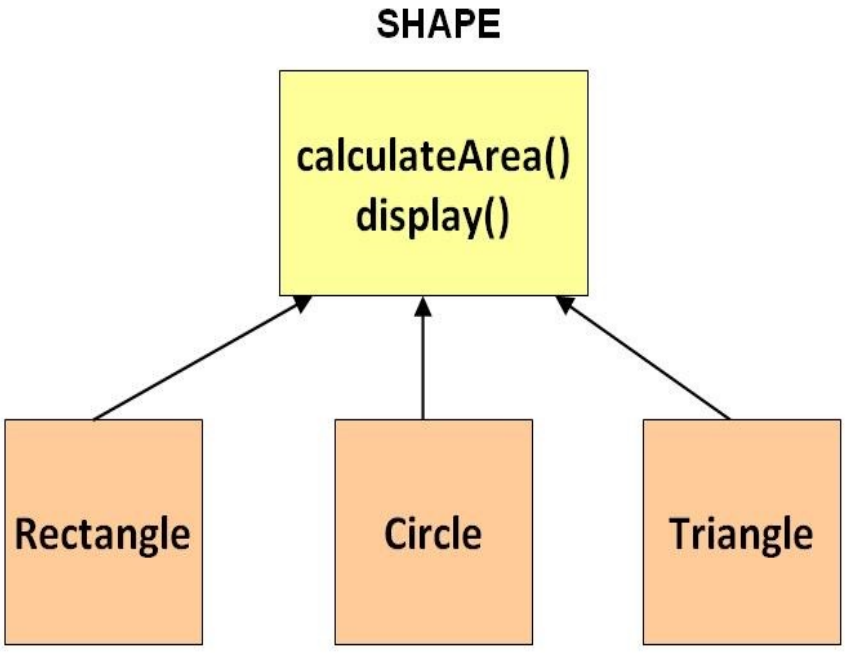
  `Sleet 2`

Problem 2

```
Snow 2
Fog 3
```

Abstract Classes

- Consider the following class hierarchy



- The `Shape` class is created to save on comm
by the `Rectangle`, `Circle`, and `Triangle` class

Abstract Classes

- Assume now that you write code to create objects f

```
Rectangle obj = new Rectangle();

Triangle obj = new Triangle();
```

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

?

Abstract Classes

- Assume now that you write code to create objects fo

```
Rectangle obj = new Rectangle();

Triangle obj = new Triangle();

Shape obj = new Shape();
```

????

Abstract Classes

- The Shape class serves in **achieving inherita**
not built to be instantiated
- **Abstract classes**
- An abstract class is a placeholder in a class hierarc

An abstract class cannot be instantiated

- Why?
- The use of abstract classes is a design decision; it helps
that are too general to instantiate
- To declare a class as abstract we use the modifier a
- **Syntax**

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

Abstract Classes

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

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

`Cannot instantiate the type Shape`

- Abstract classes can choose to provide implementa
- Abstract classes can choose **not** to implement meth subclasses
- How to choose not to implement a method?

abstract methods

Abstract Method

- An abstract method is a method that has just the sig

  implementation `public abstract <type> <nar`

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

Why Abstract Method?

- The formula for calculating the area of a rectang
- The `calculateArea()` methods in the `Shap` inheriting classes, thus it makes no sense writing it i
- But we need to make sure that all the inheriting clas labeled `abstract`

Abstract Classes

```
public abstract class Shape{

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

    public abstract double calculateArea();
}
```

- An `abstract` method cannot be defined as `fina`
- The child of an `abstract` class must override the
- Methods can call `abstract` methods

Abstract Classes and Inheritance

- `abstract` classes can be inherited
- Subclass of abstract class inherits `abstract` met
- Subclasses must provide implementation for inherited `ab`

```java
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 eith
class or if you don't, the compiler will add a default con

- Why can an abstract class have a constructor?
- When a class extends an abstract class, the constructo
super class either implicitly or explicitly

Abstract Classes & Constructors

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

        protected Fruit(String color, b
                this.color = color;
                this.seasonal = season
        } public abstract void

        prepare();

        public String getColor() {
                return color;
        }
```

```
        public boolean isSeasonal() {
                return seasonal;
        }

}
```

Abstract Classes & Constructors (c

```
public class Mango extends Fruit {

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


public class Banana extends Fruit {

        public Banana(String color, boolean
                super(color, seasonal);
        }
        @Override
        public void prepare() {
```

```
                    System.out.println("Peel
        } }
```

Abstract Classes & Constructors

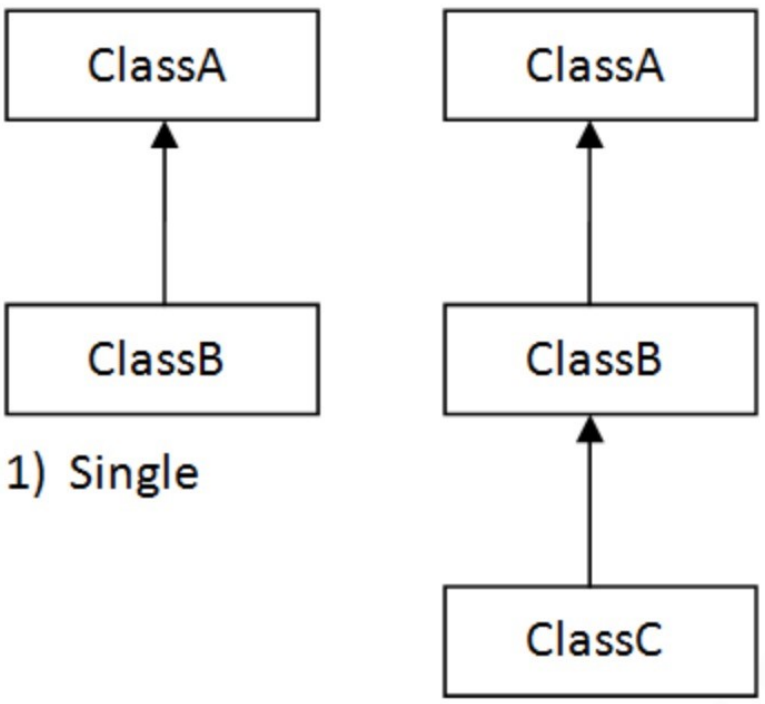
- You may define more than one constructor (with c
  should define all your constructors `protected`
- Making them public is pointless anyway

Abstract Classes in summary

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

Interfaces

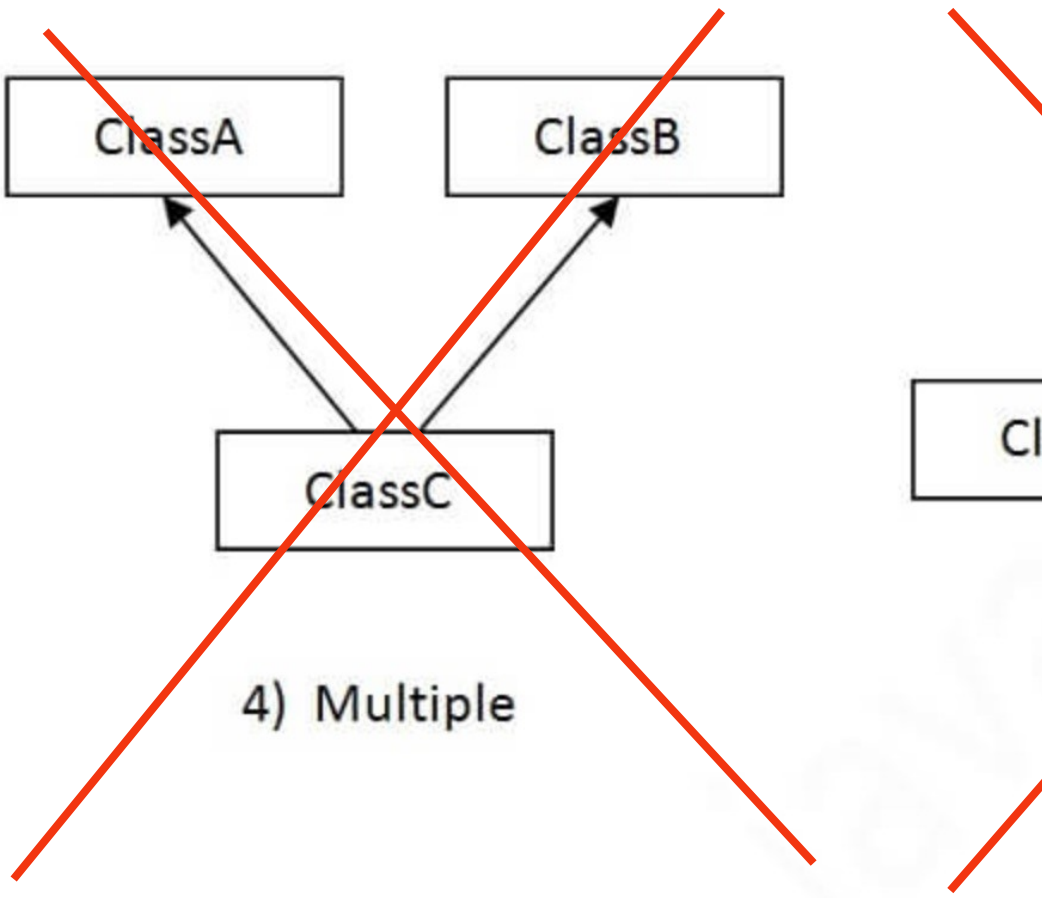
Types of inheritance in Java



1) Single

2) Multilevel

Types of inheritance in Java



4) Multiple

Multiple Inheritance

- Java supports single inheritance, meaning that a de
class
- Multiple inheritance allows a class to be derived fro
members of all parents
- **Java does not support multiple inheritance** bec

conflicts
- Which class should `super` refer when child cla
- Alternative: **Interface**
- Looks like a class
- It describes what a class does

- You can have a class that **extends** one class and i

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 impleme
- An interface can have methods just like the class, l by default `abstract`
- A Java class can implement multiple Java Interface
- Inheritance gives you an is-a relationship and code

- Interfaces give you an is-a relationship without cod

Why Interfaces?

- They are used for full abstraction
- Methods in interfaces do not have body, they must be im
  them
- The class that implements interface must implement all t

Interface Syntax

- Interface declaration, general syntax:

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

Interface Definition

FORM:

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

EXAMPLE:

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

constant declaration

- As such, they may be omitted

Shape Interface

- An interface for shapes:

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

- This interface describes the features common to all
perimeter)

Implementing an Interface

- **A class can declare that it implements an i**
- This means the class contains an implementation f

interface

- Implementing an interface, general syntax

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

- Example

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

Interface Requirements

- If we write a class that claims to be a `Shape` but do
  `perimeter` methods, it will not compile
- Examp

```
public class Banana implements

}
```

- The compiler error message:

```
Banana.java:1: Banana is not abstract and does
Shape

public class Banana implements Shape {

            ^
```