Lecture 5 Outline

- Review: Creating classes from scratch
- Class relationship 2: Inheritance
- Creating sub/child/inherited/derived classes
import java.awt.Graphics;

public class BallSimple {
    private int dia;
    private int[] pos = {0,0};
    private int[] vel = {10, 10};

    public BallSimple(int diameter){
        this.dia = diameter;
    }
    public void draw(Graphics g){
        g.drawOval(pos[0], pos[1], dia, dia);
        g.fillOval(pos[0], pos[1], dia, dia);
    }
    public void update(){
        pos[0] += vel[0];
        pos[1] += vel[1];
        vel[1] += 9.8f;
    }
}

Creating Class From Scratch
import java.awt.Graphics;

public class BallSimple {
    private int dia;
    private int[] pos = {0,0};
    private int[] vel = {10, 10};

    public BallSimple(int diameter){
        this.dia = diameter;
    }

    public void draw(Graphics g){
        g.drawOval(pos[0], pos[1], dia, dia);
        g.fillOval(pos[0], pos[1], dia, dia);
    }

    public void update(){
        pos[0] += vel[0];
        pos[1] += vel[1];
        vel[1] += 9.8f;
    }
}

Creating Class From Scratch

• Member data field
• Constructor method
• Accessor method
• Mutator method
Creating Class From Scratch

import java.awt.Graphics;

public class BallSimple {
    private int dia;
    private int[] pos = {0,0};
    private int[] vel = {10, 10};

    public BallSimple(int diameter){
        this.dia = diameter;
    }

    public void draw(Graphics g){
        g.drawOval(pos[0], pos[1], dia, dia);
        g.fillOval(pos[0], pos[1], dia, dia);
    }

    public void update(){
        pos[0] += vel[0];
        pos[1] += vel[1];
        vel[1] += 9.8f;
    }
}

• Member data field
• Constructor method
• Accessor method
• Mutator method

• Encapsulate object by making instance member data fields private
import java.awt.Graphics;

public class BallSimple {
    private int dia;
    private int[] pos = {0,0};
    private int[] vel = {10, 10};

    public BallSimple(int diameter){
        this.dia = diameter;
    }

    public void draw(Graphics g){
        g.drawOval(pos[0], pos[1], dia, dia);
        g.fillOval(pos[0], pos[1], dia, dia);
    }

    public void update(){
        pos[0] += vel[0];
        pos[1] += vel[1];
        vel[1] += 9.8f;
    }
}

• Member data field
• Constructor method
• Accessor method
• Mutator method

• Encapsulate object by making instance member data fields private
• Use static member when class-wise behavior is needed
import java.awt.Graphics;
public class BallSimple {
    private int dia;
    private int[] pos = {0,0};
    private int[] vel = {10, 10};

    public BallSimple(int diameter){
        this.dia = diameter;
    }
    public void draw(Graphics g){
        g.drawOval(pos[0], pos[1], dia, dia);
        g.fillOval(pos[0], pos[1], dia, dia);
    }
    public void update(){
        pos[0] += vel[0];
        pos[1] += vel[1];
        vel[1] += 9.8f;
    }
}

• Member data field
• Constructor method
• Accessor method
• Mutator method
• Encapsulate object by making instance member data fields
  private
• Use static member when class-wise behavior is needed
• Overloading methods
import java.awt.*;
import javax.swing.*;

public class BallViewerSimple extends JPanel {
    private BallSimple ball;

    public BallViewerSimple() {
        ball = new BallSimple(50);
        this.setPreferredSize(new Dimension(100, 100));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ball.draw(g);
    }

    public static void main(String[] args) {
        BallViewerSimple viewer = new BallViewerSimple();
        JFrame frame = new JFrame("Simple Ball Viewer");
        frame.getContentPane().add(viewer);
        frame.pack();
        frame.setVisible(true);
    }
}

awt: abstract window toolkit, java’s original platform-independent windowing, graphics, and ui widget toolkit.
javax.swing: extended from awt package to provide more sophisticated gui component
From Java 6 update 12, awt and swing widgets can be mixed without z-order problem.
import java.awt.*;
import javax.swing.*;

public class BallViewerSimple extends JPanel {
    private BallSimple ball;

    public BallViewerSimple() {
        ball = new BallSimple(50);
        this.setPreferredSize(new Dimension(100, 100));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ball.draw(g);
    }

    public static void main(String[] args) {
        BallViewerSimple viewer = new BallViewerSimple();
        JFrame frame = new JFrame("Simple Ball Viewer");
        frame.getContentPane().add(viewer);
        frame.pack();
        frame.setVisible(true);
    }
}

import java.awt.*;
import javax.swing.*;

public class BallViewerSimple extends JPanel {
    private BallSimple ball;

    public BallViewerSimple() {
        ball = new BallSimple(50);
        this.setPreferredSize(new Dimension(100, 100));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ball.draw(g);
    }

    public static void main(String[] args) {
        BallViewerSimple viewer = new BallViewerSimple();
        JFrame frame = new JFrame("Simple Ball Viewer");
        frame.getContentPane().add(viewer);
        frame.pack();
        frame.setVisible(true);
    }
}
import java.awt.*;
import javax.swing.*;

public class BallViewerSimple extends JPanel {
    private BallSimple ball;

    public BallViewerSimple() {
        ball = new BallSimple(50);
        this.setPreferredSize(new Dimension(100, 100));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ball.draw(g);
    }

    public static void main(String[] args) {
        BallViewerSimple viewer = new BallViewerSimple();
        JFrame frame = new JFrame("Simple Ball Viewer");
        frame.getContentPane().add(viewer);
        frame.pack();
        frame.setVisible(true);
    }
}
import java.awt.*;
import javax.swing.*;

public class BallViewerSimple extends JPanel {
    private BallSimple ball;

    public BallViewerSimple() {
        ball = new BallSimple(50);
        this.setPreferredSize(new Dimension(100,100));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ball.draw(g);
    }

    public static void main(String[] args) {
        BallViewerSimple viewer = new BallViewerSimple();
        JFrame frame = new JFrame("Simple Ball Viewer");
        frame.getContentPane().add(viewer);
        frame.pack();
        frame.setVisible(true);
    }
}
Class Relationship 2: Inheritance

![Class Relationship Diagram]

- **Parent**
- **Child**
- **Mammal**
- **Horse**
Class Relationship 2: Inheritance

UML symbol for inheritance

Parent

Child

Mammal

Horse
Class Relationship 2: Inheritance

- The Child "is a" more specific version of the Parent
Class Relationship 2: Inheritance

• The Child “is a” more specific version of the Parent

• The child class inherits every member its ancestor classes hold (accumulation of members)
Class Relationship 2: Inheritance

- The Child “is a” more specific version of the Parent
- The child class inherits every member its ancestor classes hold (accumulation of members)
- One can add more members to the child class, or override the same method from its ancestor with a new version
Class Relationship 2:
Inheritance

- The Child “is a” more specific version of the Parent
- The child class inherits every member its ancestor classes hold (accumulation of members)
- One can add more members to the child class, or **override** the same method from its ancestor with a new version
- Inheritance promotes code reuse!
Creating Sub/Child/Inherited/Derived classes

```java
public class Parent {
    private int field;
    public Parent(){ field = 0; }
    public String toString(){
        return "field="+field;
    }
}
```

```java
public class Child extends Parent {
}
```

Note: make it clear that private-ized member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
Creating Sub/Child/Inherited/Derived classes

- The **extends** keyword sets up the relationship

```java
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "field=\"+field;
    }
}

public class Child extends Parent {
}
```

Note: make it clear that private-ized member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
Creating Sub/Child/Inherited/Derived classes

• The **extends** keyword sets up the relationship

```java
public class Parent {
    private int field;
    public Parent(){ field = 0; }
    public String toString(){
        return "field="+field;
    }
}
```

```java
public class Child extends Parent {
}
```

Now Child has everything defined in Parent

Note: make it clear that private-IZED member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
Creating Sub/Child/Inherited/Derived classes

- The **extends** keyword sets up the relationship

```java
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "field=\"+field;
    }
}
```

```java
public class Child extends Parent {
    private double newField;
    public Child() {
        super();
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() + "newField=\"+newField;
    }
}
```

Note: make it clear that private-ized member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
Creating Sub/Child/Inherited/Derived classes

- The `extends` keyword sets up the relationship
- The `super` reference gets hold of members from the parent class

Note: make it clear that private-ized member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
Creating Sub/Child/Inherited/Derived classes

- The `extends` keyword sets up the relationship
- The `super` reference gets hold of members from the parent class

Note: make it clear that private-ized member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
Creating Sub/Child/Inherited/Derived classes

- The `extends` keyword sets up the relationship
- The `super` reference gets hold of members from the parent class

Note: make it clear that private-ized member from parent are invisible in child, but they are existing in child, child object holds memory spaces for those entities, although child class cannot directly use them. Invisible does not imply non-existing!!
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "Parent:field=\""+field;
    }
}

public class Child extends Parent {
    private double newField;
    public Child() {
        super();
        newField = 9.8f;
    }
    public String toString() {
        return super.toString()
                +"Child:newField=\""+newField;
    }
}
Method Overriding

- Child.toString() **overrides** Parent.toString() with same method signature but different procedure

```java
public class Parent {
    private int field;
    public Parent(){ field = 0; }
    public String toString(){
        return "Parent:field=\"+field;
    }
}

public class Child extends Parent {
    private double newField;
    public Child() {
        super.();
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() + "\nChild:newField=\"+newField;
    }
}
```
• Child.toString() **overrides** Parent.toString() with same method signature but different procedure

• Which one is used? The one defined within the closest scope is used.
Method Overriding

- Child.toString() **overrides** Parent.toString() with same method signature but different procedure

- Which one is used? The one defined within the closest scope is used.

```java
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "Parent:field="+field;
    }
}

public class Child extends Parent {
    private double newField;
    public Child() {
        super();
        newField = 9.8f;
    }
    public String toString() {
        return super.toString()
                + "\nChild:newField="+newField;
    }
}
```

Child james = new Child();
System.out.println(james);
Method Overriding

- Child.toString() **overrides** Parent.toString() with same method signature but different procedure

- Which one is used? The one defined within the closest scope is used.

- Use **final** modifier to forbid overriding in descendants
Java Don’t Like Multiple Inheritance
Java Don’t Like Multiple Inheritance
Java Don’t Like Multiple Inheritance

- One parent class can have many child classes - Multiple derivation is OK.
Java Don’t Like Multiple Inheritance

- One parent class can have many child classes - Multiple derivation is OK.
Java Don’t Like Multiple Inheritance

- One parent class can have many child classes - Multiple derivation is OK.
- But one child class can only have one parent class - Multiple inheritance is NOT OK.
Preserve Encapsulation Between Super and Sub classes

```java
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "Parent:field=\n" + field;
    }
}

public class Child extends Parent {
    private double newField;
    public Child() {
        field = 10;
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() + "\nChild:newField=\n" + newField;
    }
}
```
Preserve Encapsulation Between Super and Sub classes

```java
public class Parent {
    private int field;
    public Parent(){ field = 0; }
    public String toString(){
        return "Parent:field="+field;
    }
}
```

```java
public class Child extends Parent {
    private double newField;
    public Child() {
        field = 10;
        newField = 9.8f;
    }
    public String toString() {
        return super.toString()
                + "Child:newField="+newField;
    }
}
```
Preserve Encapsulation Between Super and Sub classes

```java
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "Parent:field=" + field;
    }
}
```

```java
public class Child extends Parent {
    private double newField;
    public Child() {
        field = 10;
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() + "Child:newField=" + newField;
    }
}
```

Because it's not visible from outside Parent.

Monday, February 7, 2011
Preserve Encapsulation Between Super and Sub classes

- **Protected** visibility modifier: makes the modified member visible to current class and all its descends

```java
public class Parent {
    private int field;
    public Parent(){ field = 0; }
    public String toString(){
        return "Parent:field="+field;
    }
}
```

```java
public class Child extends Parent {
    private double newField;
    public Child() {
        field = 10;
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() +
        "Child:newField="+newField;
    }
}
```

Because it's not visible from outside Parent
Preserve Encapsulation Between Super and Sub classes

- **Protected** visibility modifier: makes the modified member visible to current class and all its descends

```java
class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "Parent:field=":field;
    }
}

class Child extends Parent {
    private double newField;
    public Child() {
        field = 10;
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() + "Child:newField=":newField;
    }
}
```

Fix by changing visibility from private to protected
Preserve Encapsulation Between Super and Sub classes

- **Protected** visibility modifier: makes the modified member visible to current class and all its descends

- So child classes can still access/mutate/invoke that member

- Keep encapsulation to non-descending classes

```java
public class Parent {
    private int field;
    public Parent() { field = 0; }
    public String toString() {
        return "Parent:field=" + field;
    }
}
```

```java
public class Child extends Parent {
    private double newField;
    public Child() {
        field = 10;
        newField = 9.8f;
    }
    public String toString() {
        return super.toString() + "Child:newField=" + newField;
    }
}
```
Demo and Summary

- Use `extends` to set up the relationship in child classes
- Use `super` to refer to members defined in the parent class
- Use `protected` to allow child classes manipulate members
- Use `final` to disallow child classes overriding the member method
- Derive a new child class if you want to use an existing class with some changes

Demonstrate how to derive BallInBox from Ball, then use the new class in BallViewer.
Lecture 6 Outline

• Review: Deriving classes from a base class
• Class hierarchy
• Abstract classes
• Design with inheritance
Deriving Classes From A Base Class
Deriving Classes From A Base Class

- Child holds everything from parent by `extends`
- Child sees only `protected` or public members
- Child class can hold more member fields/methods, or override parent class methods that are not `final`
- Child class can refer to its parent as `super`
- Inheritance promote code reuse!!!
Class Hierarchy - Multiple Generations of Inheritance

Demonstrate multiple generation of inheritance.
Demonstrate multiple generation of inheritance.
Class Hierarchy - Multiple Generations of Inheritance

• Snake includes everything defined in Reptile and in Animal -- Child includes everything defined in all its ancestors

• Reduce code duplication by pulling common behaviors between sibling classes into their parent class

• Remember a child makes a difference from its parent
Show in eclipse the feature to get all available methods.
Java Class Hierarchy

- **Object** - The root class of everything in Java

- A class without explicit `extends` has the implicit `extends Object`

- `toString` is defined in Object

- Object class also holds other useful methods such as `equals`, `getClass`, `hashCode`, etc.

Show in eclipse the feature to get all available methods.
Abstract Class

Other examples, Graphic Object -> Rectangle, Circle, Line, Curve
Emphasize on the idea of something in common, but cannot be defined with a concrete body
Abstract Class

- Vehicle is just a general term for all transportations. We know every vehicle consume fuel, but we can only tell how much it consume given a specific type of Vehicle.

- An abstract class serve for defining a general form of a set of sibling classes, but the class by itself is not instantiable

- Often, an abstract class holds methods with only the headers. Because the method body can not be determined at this level of abstraction

Other examples, Graphic Object -> Rectangle, Circle, Line, Curve
Emphasize on the idea of something in common, but cannot be defined with a concrete body
Abstract Class - Syntax

- Methods that cannot have a defined body need **abstract** modifier in the header

- Classes with an abstract method need **abstract** modifier in the header

- A concrete class (class that can be instantiated) derived from an abstract class must overrides all abstract methods
Abstract Class - Syntax

- Methods that cannot have a defined body need `abstract` modifier in the header
- Classes with an abstract method need `abstract` modifier in the header
- A concrete class (class that can be instantiated) derived from an abstract class must overrides all abstract methods

```java
public abstract class GraphicObject {
    // declare fields
    // declare non-abstract methods
    abstract void draw();
    abstract void resize();
}
```
Abstract Class - Syntax

- Methods that cannot have a defined body need `abstract` modifier in the header
- Classes with an abstract method need `abstract` modifier in the header

```java
public abstract class GraphicObject {
    // declare fields
    // declare non-abstract methods
    abstract void draw();
    abstract void resize();
}
```
Abstract Class - Syntax

- Methods that cannot have a defined body need `abstract` modifier in the header

- Classes with an abstract method need `abstract` modifier in the header

- A concrete class (class that can be instantiated) derived from an abstract class must overrides all abstract methods

```java
public abstract class GraphicObject {
    // declare fields
    // declare non-abstract methods
    abstract void draw();
    abstract void resize();
}

class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```
Abstract Class - Design
Abstract Class - Design

• We will use abstract class and its derivatives next week to show its significance in Polymorphism
Abstract Class - Design

• We will use abstract class and its derivatives next week to show its significance in Polymorphism

• We will compare Abstract and Interface next week to show their specialties in Polymorphism
Design with Inheritance
Design with Inheritance

- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible
Design with Inheritance

• Think about the potential future of a class hierarchy, and design classes to be reusable and flexible

• Commonalities -- push to parent
Design with Inheritance

• Think about the potential future of a class hierarchy, and design classes to be reusable and flexible

• Commonalities -- push to parent

• Variations -- add new fields/methods, or override methods into child
Design with Inheritance

- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible

- Commonalities -- push to parent

- Variations -- add new fields/methods, or override methods into child

- Every child “is an” extension of the parent class
Design with Inheritance  - continued
Design with Inheritance - continued

- Try your best to let each class manage its own claimed data.
Design with Inheritance - continued

- Try your best to let each class manage its own claimed data.

- Use super.parentMethod() whenever it applies - stay away from reinventing the wheel as much as you can
Design with Inheritance - continued

• Try your best to let each class manage its own claimed data.

• Use super.parentMethod() whenever it applies - stay away from reinventing the wheel as much as you can

• Play with private/protect to control the visibility in descendants
Design with Inheritance

- Try your best to let each class manage its own claimed data.

- Use super.parentMethod() whenever it applies - stay away from reinventing the wheel as much as you can

- Play with private/protect to control the visibility in descendants

- Play with final/abstract to control overriding in descendants
Homework and Summary

- **Know** - Inheritance sets the “is-a” relationship
Homework and Summary

• **Know** - Inheritance sets the “is-a” relationship

• **How** - Important keywords: extends, super, protected, final, abstract
Homework and Summary

- **Know** - Inheritance sets the “is-a” relationship

- **How** - Important keywords: extends, super, protected, final, abstract

- **Apply** - Design system with object abstraction in hierarchical inheritance