

Inheritance and Polymorphism

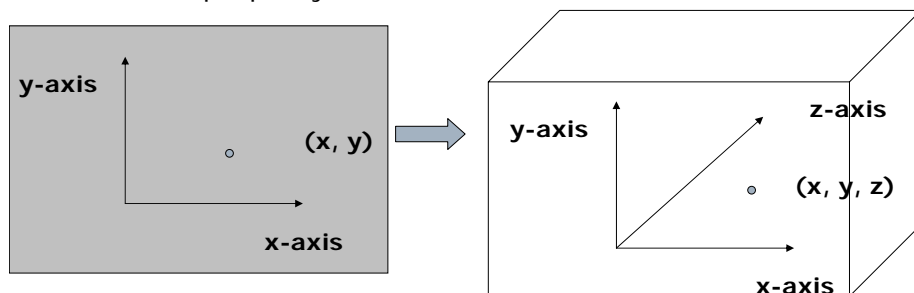
Yao, Zhiyang

Inheritance

- Organizes objects in a top-down fashion from most general to least general
- Inheritance defines a “is-a” relationship
 - A mountain bike “is a” kind of bicycle
 - A SUV “is a” kind of automobile
 - A border collie “is a” kind of dog
 - A laptop “is a” kind of computer

ThreeDimensionalPoint

- Build a new class `ThreeDimensionalPoint` using inheritance
 - `ThreeDimensionalPoint` extends the awt class `Point`
 - `Point` is the **superclass** (base class)
 - `ThreeDimensionalPoint` is the **subclass** (derived class)
 - `ThreeDimensionalPoint` extends `Point` by adding a new property to `Point`—a z-coordinate



Class ThreeDimensionalPoint

```
package geometry;

import java.awt.*;

public class ThreeDimensionalPoint extends Point {
    // private class constant
    private final static int DEFAULT_Z = 0;

    // private instance variable
    private int z = DEFAULT_Z;
```

See next slide

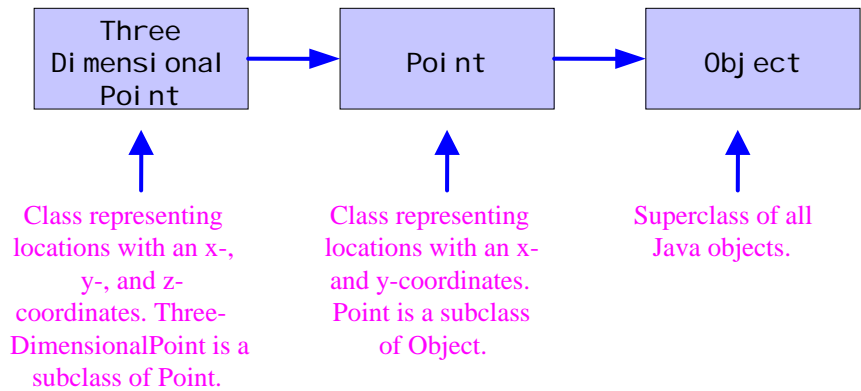
Keyword extends indicates that ThreeDimensionalPoint is a subclass of Point

New instance variable

Packages

- Allow definitions to be collected together into a single entity—a package
- ThreeDimensionalPoint will be added to the geometry package
 - Classes and names in the same package are stored in the same folder
 - Classes in a package go into their own namespace and therefore the names in a particular package do not conflict with other names in other packages
 - For example, a package called Graph might have a different definition of ThreeDimensionalPoint
 - When defining members of a class or interface, Java does not require an explicit access specification. The implicit specification is known as *default access*. Members of a class with default access can be accessed only by members of the package.

Java's Mother-of-all-objects— Class Object



ThreeDimensionalPoint

```
ThreeDimensionalPoint a =
    new ThreeDimensionalPoint(6, 21, 54);
a.translate(1, 1); // invocation of superclass translate()
a.translate(2, 2, 2); // invocation of subclass translate()
```

- Java determines which method to use based on the number of parameters in the invocation
- After the first call to translate, what is the value of a?
- After the second call to translate, what is the value of a?

ThreeDimensionalPoint

- Methods toString(), equals(), and clone() should not have different signatures from the Point versions

```
ThreeDimensionalPoint c = new ThreeDimensionalPoint(1, 4, 9);
```

```
ThreeDimensionalPoint d = (ThreeDimensionalPoint)
    c.clone();
```

Cast is necessary as return type of subclass method clone() is Object

```
String s = c.toString();
```

Invocation of subclass toString() method

```
boolean b = c.equals(d);
```

Invocation of subclass equals() method

ThreeDimensionalPoint

□ Constructors

// ThreeDimensionalPoint(): default constructor

```
public ThreeDimensionalPoint() {  
    super();  
    setZ(DEFAULT_Z);  
}
```

// ThreeDimensionalPoint(): specific constructor

```
public ThreeDimensionalPoint(int a, int b, int c) {  
    super(a, b);  
    setZ(c);  
}
```

ThreeDimensionalPoint

□ Accessors and mutators

// getZ(): z-coordinate accessor

```
public double getZ() {  
    return z;  
}
```

// setZ(): y-coordinate mutator

```
public void setZ(int value) {  
    z = value;  
}
```

ThreeDimensionalPoint

□ Facilitators

// translate(): shifting facilitator

```
public void translate(int dx, int dy, int dz) {  
    translate(dx, dy);
```

```
    int zValue = (int) getZ();
```

```
    setZ(zValue + dz);
```

```
}
```

ThreeDimensionalPoint

□ Facilitators

// toString(): conversion facilitator

```
public String toString() {  
    int a = (int) getX();  
    int b = (int) getY();  
    int c = (int) getZ();  
    return getClass() +  
        "[" + a + ", " + b + ", " + c + "];"  
}
```

ThreeDimensionalPoint

Facilitators

```
// equals(): equality facilitator
public boolean equals(Object v) {
    if (v instanceof ThreeDimensionalPoint) {
        ThreeDimensionalPoint p =
            (ThreeDimensionalPoint) v;
        int z1 = (int) getZ();
        int z2 = (int) p.getZ();

        return super.equals(p) && (z1 == z2);
    }
    else {
        return false;
    }
}
```

ThreeDimensionalPoint

Facilitators

```
// clone(): clone facilitator
public Object clone() {
    int a = (int) getX();
    int b = (int) getY();
    int c = (int) getZ();

    return new ThreeDimensionalPoint(a, b, c);
}
```

ColoredPoint

- Suppose an application calls for the use of colored points.
- We can naturally extend class Point to create ColoredPoint
- Class ColoredPoint will be added to **package** geometry

```
package geometry;
```

```
import java.awt.*;
```

```
public class ColoredPoint extends Point {
    // instance variable
    Color color;
    ...
}
```

ColoredPoint

Constructors

```
// ColoredPoint(): default constructor
```

```
public ColoredPoint() {
    super();
    setColor(Color.BLUE);
}
```

```
// ColoredPoint(): specific constructor
```

```
public ColoredPoint(int x, int y, Color c) {
    super(x, y);
    setColor(c);
}
```

ColoredPoint

Facilitators

```
// equals(): equal facilitator
public boolean equals(Object v) {
    if (v instanceof ColoredPoint) {
        Color c1 = getColor();
        Color c2 = ((ColoredPoint) v).getColor();
        return super.equals(v) && c1.equals(c2);
    }
    else {
        return false;
    }
}
```

Colored3DPoint

- Suppose an application needs a colored, three-dimensional point.
- Can we create such a class by extending both ThreeDimensionalPoint and ColoredPoint?
 - No. Java does not support multiple inheritance
 - Java only supports single inheritance

```
package Geometry;
import java.awt.*;
```

```
public class Colored3DPoint extends
    ThreeDimensionalPoint {
    // instance variable
    Color color;
```

Colored3DPoint

Constructors

```
// Colored3DPoint(): default constructor
public Colored3DPoint() {
    setColor(Color.BLUE);
}

// Colored3DPoint(): specific constructor
public Colored3DPoint(int a, int b, int c, Color d)
{
    super(a, b, c);
    setColor(d);
}
```

Colored3DPoint

Facilitators

```
// equals(): equal facilitator
public boolean equals(Object v) {
    if (v instanceof Colored3DPoint) {
        Color c1 = getColor();
        Color c2 = ((Colored3DPoint)
v).getColor();
        return super.equals(v) && c1.equals(c2);
    }
    else {
        return false;
    }
}
```

Polymorphism

- A code expression can invoke different methods depending on the types of objects being manipulated
- Example: function overloading like method `min()` from `java.lang.Math`
 - The method invoked depends on the types of the actual arguments

Example

```
int a, b, c;
double x, y, z;
...
c = min(a, b);    // invokes integer min()
z = min(x, y);   // invokes double min
```

Polymorphism

- Two types of polymorphism
 - Syntactic polymorphism—Java can determine which method to invoke at compile time
 - Efficient
 - Easy to understand and analyze
 - Also known as primitive polymorphism
 - Pure polymorphism—the method to invoke can only be determined at execution time

Polymorphism

- Pure polymorphism example

```
public class PolymorphismDemo {
    // main(): application entry point
    public static void main(String[] args) {
        Point[] p = new Point[4];

        p[0] = new Colored3DPoint(4, 4, 4, Color.BLACK);
        p[1] = new ThreeDimensionalPoint(2, 2, 2);
        p[2] = new ColoredPoint(3, 3, Color.RED);
        p[3] = new Point(4, 4);

        for (int i = 0; i < p.length; ++i) {
            String s = p[i].toString();
            System.out.println("p[" + i + "]: " + s);
        }

        return;
    }
}
```

Inheritance nuances

- When a new object that is a subclass is constructed, the constructor for the superclass is always called.
 - Constructor invocation may be implicit or explicit

Example

```
public class B {
    // B(): default constructor
    public B() {
        System.out.println("Using B's default constructor");
    }

    // B(): specific constructor
    public B(int i) {
        System.out.println("Using B's int constructor");
    }
}
```

Inheritance nuances

```
public class C extends B {
    // C(): default constructor
    public C() {
        System.out.println("Using C's default
        constructor");
        System.out.println();
    }

    // C(int a): specific constructor
    public C(int a) {
        System.out.println("Using C's int
        constructor");
        System.out.println();
    }
}
```

Inheritance nuances

```
// C(int a, int b): specific constructor
public C(int a, int b) {
    super(a + b);
    System.out.println("Using C's int-int
    constructor");
    System.out.println();
}

// main(): application entry point
public static void main(String[] args) {
    C c1 = new C();
    C c2 = new C(2);
    C c3 = new C(2, 4);
    return;
}
```

Inheritance nuances

Output

Using B's default constructor
Using C's default constructor

Using B's default constructor
Using C's int constructor

Using B's int constructor
Using C's int-int constructor

1. Construction of C is a two-step process, the construction of each C object begins with the construction of its superclass attributes, i.e., construct B first, then C.
2. **Unless explicitly invoking a superclass's constructor, superclass's default constructor is invoked.**

Controlling access

□ Class access rights

Member Restriction	this	Subclass	Package	General
public	✓	✓	✓	✓
protected	✓	✓	✓	—
default	✓	—	✓	—
private	✓	—	—	—

Controlling access

Example

```
package demo;

public class P {
    // instance variable
    private int data;

    // P(): default constructor
    public P() {
        setData(0);
    }

    // getData(): accessor
    public int getData() {
        return data;
    }
}
```

Controlling access

Example (continued)

```
// setData(): mutator
protected void setData(int v) {
    data = v;
}

// print(): facilitator
void print() {
    System.out.println();
}
}
```

Controlling access

Example

```
import demo.P;

public class Q extends P {
    // Q(): default constructor
    public Q() {
        super();
    }

    // Q(): specific constructor
    public Q(int v) {
        setData(v);
    }
}
```

```
package demo;
public class P {
    // instance variable
    private int data;
    // P(): default constructor
    public P() {
        setData(0);
    }
    // getData(): accessor
    public int getData() {
        return data;
    }
    // setData(): mutator
    protected void setData(int v) {
        data = v;
    }
    // print(): facilitator
    void print() {
        System.out.println();
    }
}
```

Q can access superclass's public default constructor

Q can access superclass's protected mutator

Controlling access

Example

```
// toString(): string facilitator
public String toString() {
    int v = getData();
    return String.valueOf(v);
}

// invalid1(): illegal method
public void invalid1() {
    data = 12;
}

// invalid2(): illegal method
public void invalid2() {
    print();
}
}
```

```
package demo;
public class P {
    // instance variable
    private int data;
    // P(): default constructor
    public P() {
        setData(0);
    }
    // getData(): accessor
    public int getData() {
        return data;
    }
    // setData(): mutator
    protected void setData(int v) {
        data = v;
    }
    // print(): facilitator
    void print() {
        System.out.println();
    }
}
```

Q can access superclass's public accessor

Q cannot access superclass's private data field

Q cannot directly access superclass's default access method print()

Controlling access

Example

```
package demo;
```

```
public class R {  
    // instance variable  
    private P p;  
  
    // R(): default constructor  
    public R() {  
        p = new P();  
    }  
    // set(): mutator  
    public void set(int v) {  
        p.setData(v);  
    }  
}
```

R can access P's public default constructor

R can access P's protected mutator

```
package demo;  
public class P {  
    // instance variable  
    private int data;  
    // P(): default constructor  
    public P() {  
        setData(0);  
    }  
    // getData(): accessor  
    public int getData() {  
        return data;  
    }  
    // setData(): mutator  
    protected void setData(int v) {  
        data = v;  
    }  
    // print(): facilitator  
    void print() {  
        System.out.println();  
    }  
}
```

Controlling access

Example

```
// get(): accessor  
public int get() {  
    return p.getData();  
}  
// use(): facilitator  
public void use() {  
    p.print();  
}  
// invalid(): illegal method  
public void invalid() {  
    p.data = 12;  
}
```

R can access P's public accessor

R can access P's default access method

R cannot directly access P's private data

```
package demo;  
public class P {  
    // instance variable  
    private int data;  
    // P(): default constructor  
    public P() {  
        setData(0);  
    }  
    // getData(): accessor  
    public int getData() {  
        return data;  
    }  
    // setData(): mutator  
    protected void setData(int v) {  
        data = v;  
    }  
    // print(): facilitator  
    void print() {  
        System.out.println();  
    }  
}
```

Controlling access

Example

```
import demo.P;
```

```
public class S {  
    // instance variable  
    private P p;  
  
    // S(): default constructor  
    public S() {  
        p = new P();  
    }  
    // get(): inspector  
    public int get() {  
        return p.getData();  
    }  
}
```

S can access P's public default constructor

S can access P's public accessor

```
package demo;  
public class P {  
    // instance variable  
    private int data;  
    // P(): default constructor  
    public P() {  
        setData(0);  
    }  
    // getData(): accessor  
    public int getData() {  
        return data;  
    }  
    // setData(): mutator  
    protected void setData(int v) {  
        data = v;  
    }  
    // print(): facilitator  
    void print() {  
        System.out.println();  
    }  
}
```

Controlling access

Example

```
// illegal1(): illegal method  
public void illegal1(int v) {  
    p.setData(v);  
}  
// illegal2(): illegal method  
public void illegal2() {  
    p.data = 12;  
}  
// illegal3(): illegal method  
public void illegal3() {  
    p.print();  
}
```

S cannot access P's protected mutator

S cannot access directly P's private data field

S cannot access directly P's default access method print()

```
package demo;  
public class P {  
    // instance variable  
    private int data;  
    // P(): default constructor  
    public P() {  
        setData(0);  
    }  
    // getData(): accessor  
    public int getData() {  
        return data;  
    }  
    // setData(): mutator  
    protected void setData(int v) {  
        data = v;  
    }  
    // print(): facilitator  
    void print() {  
        System.out.println();  
    }  
}
```

Data fields

- A superclass's instance variable can be hidden by a subclass's definition of an instance variable with the same name

Example

```
public class D {
    // D instance variable
    protected int d;

    // D(): default constructor
    public D() {
        d = 0;
    }
    // D(): specific constructor
    public D(int v) {
        d = v;
    }
}
```

Data fields

Class D (continued)

```
// printD(): facilitator
public void printD() {
    System.out.println("D's d: " + d);
    System.out.println();
}
}
```

Data fields

- Class F extends D and introduces a new instance variable named d. F's definition of d hides D's definition.

```
public class D {
    // D instance variable
    protected int d;

    // D(): default constructor
    public D() {
        d = 0;
    }
    // D(): specific constructor
    public D(int v) {
        d = v;
    }
    // printD(): facilitator
    public void printD() {
        System.out.println("D's d: " + d);
        System.out.println();
    }
}
```

```
public class F extends D {
    // F instance variable
    int d;
```

```
// F(): specific constructor
```

```
public F(int v) {
```

```
    d = v;
```

```
    super.d = v*100;
```

```
}
```

Modification of this's d

Modification of superclass's d

Data fields

Class F (continued)

```
// printF(): facilitator
public void printF() {
    System.out.println("D's d: " + super.d);
    System.out.println("F's d: " + this.d);
    System.out.println();
}
}
```

```
public class D {
    // D instance variable
    protected int d;

    // D(): default constructor
    public D() {
        d = 0;
    }
    // D(): specific constructor
    public D(int v) {
        d = v;
    }
    // printD(): facilitator
    public void printD() {
        System.out.println("D's d: " + d);
        System.out.println();
    }
}
```

Polymorphism and late binding

Example

```
public class L {
    // L(): default constructor
    public L() {
    }
    // f(): facilitator
    public void f() {
        System.out.println("Using L's f()");
        g();
    }
    // g(): facilitator
    public void g() {
        System.out.println("using L's g()");
    }
}
```

Polymorphism and late binding

Example

```
public class M extends L {
    // M(): default constructor
    public M() {
        // no body needed
    }
    // g(): facilitator
    public void g() {
        System.out.println("Using M's g()");
    }
}
```

Polymorphism and late binding

Example

```
// main(): application entry point
public static void main(String[] args) {
    L l = new L();
    M m = new M();
    l.f();
    m.f();
    return;
}
}
```

Outputs

```
Using L's f()
using L's g()
Using L's f()
Using M's g()
```

```
public class L {
    // L(): default constructor
    public L() {
    }
    // f(): facilitator
    public void f() {
        System.out.println("Using L's f()");
        g();
    }
    // g(): facilitator
    public void g() {
        System.out.println("using L's g()");
    }
}
```

```
public class M extends L {
    // M(): default constructor
    public M() {
        // no body needed
    }
    // g(): facilitator
    public void g() {
        System.out.println("Using M's g()");
    }
}
```

Finality

- ❑ A final class is a class that cannot be extended.
 - Developers may not want users extending certain classes
 - Makes tampering via overriding more difficult

Example

```
final public class U {
    // U(): default constructor
    public U() {
    }

    // f(): facilitator
    public void f() {
        System.out.println("f() can't be overridden: "
            + "U is final");
    }
}
```

Finality

- A final method is a method that cannot be overridden.

Example

```
public class V {
    // V(): default constructor
    public V() {
    }

    // f(): facilitator
    final public void f() {
        System.out.println("Final method f() can't be "
            + "overridden");
    }
}
```

Abstract base classes

- Allows creation of classes with methods that correspond to an abstract concept (i.e., there is not an implementation)
- Suppose we wanted to create a class GeometricObject
 - Reasonable concrete methods include
 - getPosition()
 - setPosition()
 - getColor()
 - setColor()
 - paint()
 - For all but paint(), we can create implementations.
 - For paint(), we must know what kind of object is to be painted. Is it a square, a triangle, etc.
 - Method paint() should be an abstract method

Abstract base classes

Example

```
import java.awt.*;

abstract public class GeometricObject {
    // instance variables
    Point position;
    Color color;

    // getPosition(): return object position
    public Point getPosition() {
        return position;
    }

    // setPosition(): update object position
    public void setPosition(Point p) {
        position = p;
    }
}
```

Makes GeometricObject an abstract class

Abstract base classes

Example (continued)

```
// getColor(): return object color
public Color getColor() {
    return color;
}

// setColor(): update object color
public void setColor(Color c) {
    color = c;
}

// paint(): render the shape to graphics context g
abstract public void paint(Graphics g);
}
```

Indicates that an implementation of method paint() will not be supplied

Interfaces

- An interface is a template that specifies what must be in a class that implements the interface
 - An interface cannot specify any method implementations
 - All the methods of an interface are public
 - All the variables defined in an interface are **public**, **final**, and **static**

Interfaces

- An interface for a colorable object

```
public interface Colorable {  
    // getColor(): return the color of the object  
    public Color getColor();  
    // setColor(): set the color of the object  
    public void setColor(Color c);  
}
```

- Now the interface can be used to create classes that implement the interface

Interfaces

- ColorablePoint

```
import java.awt.*;
```

```
public class ColorablePoint extends Point implements
```

```
Colorable {
```

```
    // instance variable
```

```
    Color color;
```

Class ColorablePoint must provide implementations of getColor() and setColor()

```
    // ColorablePoint(): default constructor
```

```
    public ColorablePoint() {
```

```
        super();
```

```
        setColor(Color.BLUE);
```

```
    }
```

```
...
```

Assignment: due April 17, Tuesday before noon

- Question and answers:

- (P.509) S9.2 and S9.4
- 9.6
- 9.8
- 9.15

- For the following two programming problems, study the following three programs first: Fish.java, FishDemo.java, FishCanvas.java
 - 9.17
 - 9.20 (this is an extra practical question, if you have time, try this, and you may get bonus points if you can finish it)

Something regarding the second
midterm

□ Schedule of 2nd Midterm:

**on Tuesday, April 17, from 4:30 to
5:10am, 40 minutes, close book,
close note, in ERB 803.**