

1.00 Lecture 12

October 4, 2005

Inheritance

Inheritance

- **Inheritance allows you to write new classes based on existing (super) classes**
 - Inherit super class methods and data
 - Add new methods and data
- **This allows substantial reuse of Java code**
 - When extending software, we often write new code that invokes old code (libraries, etc.)
 - We sometimes need to have old code invoke new code (even code that wasn't imagined when the old code was written), without changing (or even having) the old code!
 - E.g. A drawing program must manage a new shape
 - Inheritance allows us to do this!

Review: Member Access

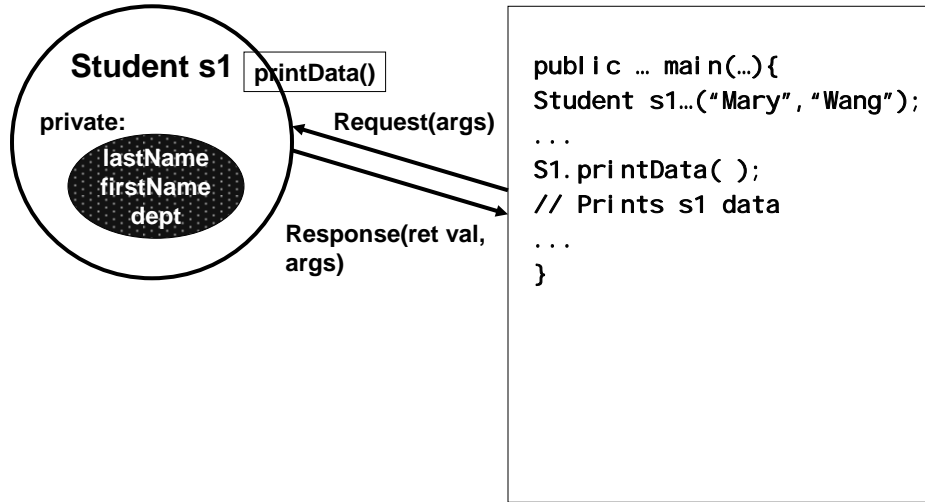
- **Class may contain members (methods or data) of type:**
 - **Private:**
 - Access only by class's methods
 - **Protected (rarely used in Java; it's pretty unsafe)**
 - Access by:
 - Class's methods
 - Methods of inherited classes, called subclasses
 - Classes in same package
 - **Package:**
 - Access by methods of classes in same package
 - **Public:**
 - Access to all classes everywhere

A Programming Project

- **Department has system with Student class**
 - Has extensive data (name, ID, courses, year, ...) for all students that you need to use/display
 - Dept wants to manage research projects better
 - Undergrads and grads have very different roles
 - Positions, credit/grading, pay, ...
 - You want to reuse the Student class but need to add very different data and methods by grad/undergrad
 - Suppose Student was written 5 years ago by someone else without any knowledge that it might be used to manage research projects

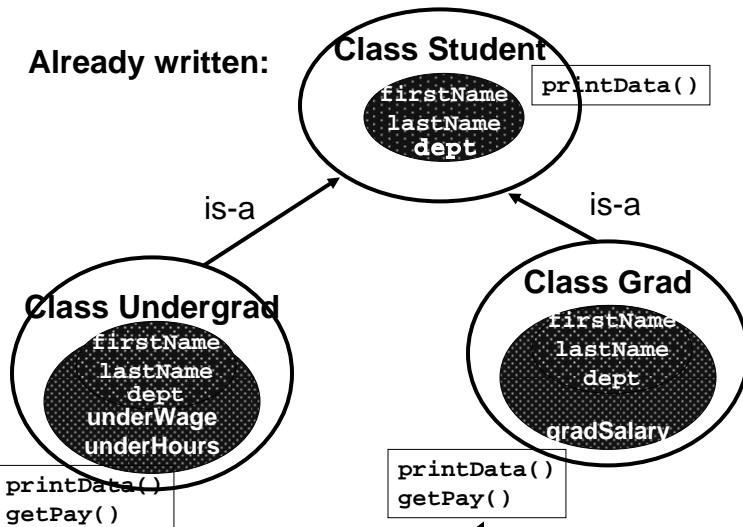
Classes and Objects

Encapsulation Message passing "Main event loop"



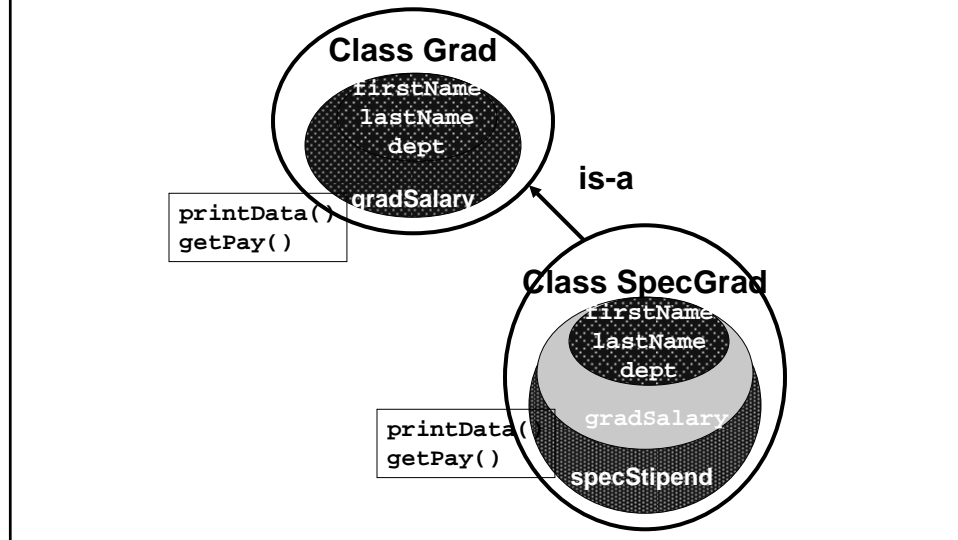
Inheritance

Already written:



You next write:

Inheritance, p.2



Example: Student class

```
class Student {  
    // Constructor  
    public Student(String fName, String lName) {  
        firstName= fName; lastName= lName;}  
    // Method  
    public void printData() {  
        System.out.println(firstName + " " + lastName);}  
    // Data  
    private String firstName;  
    private String lastName;  
}
```

This is the super (or base) class.

Undergrad student class

```
class Undergrad extends Student { // 'extends' keyword
    // Constructor, calls superclass constructor in 1st line
    public Undergrad(String fName, String lName,
        double rate, double hours) {
        super(fName, lName); // invokes Student constructor
        underWage= rate;
        underHours= hours; }

    public double getPay() {
        return underWage * underHours; }

    public void printData() {
        super.printData(); // Student getData() method
        System.out.println("Weekly pay: $" + underWage *
            underHours); }

    private double underWage;
    private double underHours;
}
```

Subclass or (directly) derived class

Grad student class

```
class Grad extends Student {
    public Grad(String fName, String lName, double salary) {
        super(fName, lName);
        gradSalary= salary; }

    public double getPay() {
        return gradSalary; }

    public void printData() {
        super.printData();
        System.out.println("Monthly salary: $" + gradSalary); }

    private double gradSalary;
}
```

Subclass or (directly) derived class

Special grad student class

```
class SpecGrad extends Grad {
    public SpecGrad(String fName, String lName, double stipend) {
        super(fName, lName, 0.0); // Zero monthly salary
        specStipend= stipend; }

    public double getPay() {
        return specStipend; }

    public void printData() {
        super.printData();
        System.out.println("Semester stipend: $" + getPay()); }
    // Using getPay is dangerous!! If a subclass is defined,
    // it will invoke its own getPay, not SpecGrad's!!

    private double specStipend;
}
```

Derived class from derived class

Main method

```
public class Student2 {
    public static void main(String[] args) {
        Undergrad ferd= new Undergrad("Ferd", "Smith", 12.00, 8.0);
        ferd.printData();
        Grad ann= new Grad("Ann", "Brown", 1500.00);
        ann.printData();
        SpecGrad mary= new SpecGrad("Mary", "Barrett", 2000.00);
        mary.printData();
        System.out.println("\n");

        // Polymorphism, or late binding
        Student[] team= new Student[3];
        team[0]= ferd;
        team[1]= ann;
        team[2]= mary;
        for (int i=0; i < 3; i++)
            team[i].printData();
        }
}
```

**Java knows the
Object's class and
chooses the
appropriate method
at run time**

Output from main method

Ferd Smith
Weekly pay: \$96.0
Ann Brown
Monthly salary: \$1500.0
Mary Barrett
Monthly salary: \$0.0
Semester stipend: \$2000.0

Note that we could not write:
`team[i].getPay();`
because `getPay()` is not a method of the superclass `Student`. In contrast, `printData()` is a method of `Student`, so Java can find the appropriate version.

We'd have similar problems with a method like `isUR0P` that would only be defined for undergrads and not in `Student`

Type and Class

Every object belongs to exactly one class, but it generally has more than one type.

- **Example: If we had a class called `Point`, the the object referred to by `p` in**

`Point p = new Point(4,5);`

is a member of the `Point` class but is also of type `Object`. (All objects in Java automatically inherit from the `Object` class.

Why type matters?

- The arguments to a method declare the type of the argument expected, not the class.
- An array can consist of references to objects of the same type, not necessarily of the same class.
- Given a reference to any object, a method called on that object refers to the method of the class of the specific object, not just its type.

Abstract Classes

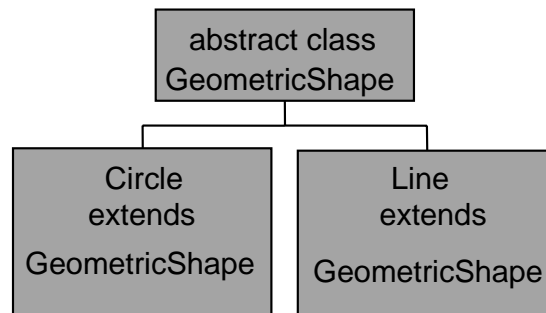
A class declared abstract, as in

```
public abstract class AClass {  
    // ... body here  
}
```

Can only be used as a base class.

- no instances can be created
- can have both abstract and non-abstract methods
- non-abstract derived classes must provide non-abstract methods

Example of Inheritance



Abstract Class

```
// This class is an abstract base class for all
// geometric shapes.
// It has a required method called output( ).

public abstract class GeometricShape extends
    Object
{
    public abstract void output();
}
```

Circle-A Concrete Class

```
import java.awt.*;
public class Circle extends GeometricShape {
    int x,y;      // coordinates of center of circle
    int radius;   // integer radius of circle
    // constructor takes 2 points: center and point on circle
    public Circle(Point p1, Point p2) {
        x = p1.x;
        y = p1.y;
        radius = (int)Math.sqrt( (x-p2.x)*(x-p2.x) +
            (y-p2.y)*(y-p2.y)); }
    public void output() {
        System.out.println("Circle from (x,y)=( " + x + " , "
            + y + " ) with r=" + radius );
    }
}
```

Line - A Concrete Class

```
import java.awt.*;
public class Line extends GeometricShape {
    int x1,y1;
    int x2,y2;
    public Line(Point p1, Point p2) {
        x1 = p1.x;
        y1 = p1.y;
        x2 = p2.x;
        y2 = p2.y; }
    public void output() {
        System.out.println("Line from (x,y)=( " + x1 + " , "
            + y1 + " ) to ( " + x2 + " , " + y2 + " )" );
    }
}
```

Example of Concrete Class

```
import java.awt.*;
public class TestPoly {
    public static void main(String[ ] argv) {
        GeometricShape[] ar = new GeometricShape[4];
        ar[0] = new Line(new Point(1, 1), new Point(2, 3) );
        ar[1] = new Circle(new Point(1, 1), new Point(9, 8) );
        ar[2] = new Line(new Point(3, 6), new Point(12, 9) );
        ar[3] = new Circle(new Point(1, 1), new Point(6, 12) );
        for(int i=0; i<ar.length; i++)
            ar[i].output();
    }
}
```

Exercise

- **Design a hierarchy for:**
 - **Plants:** photosynthetic, multicellular, kingdom Plantae
 - **Trees:** woody perennial with trunk and crown
 - **Flowers:** seed-bearing plant w/stamen, pistil, blossom
 - **Roses:** genus Rosa, pinnate leaves, prickly stems
 - **Pines:** genus Pinus, needle-shaped leaves, cones
- **Steps:**
 - Draw the inheritance tree first.
 - Define some (2-4) data fields and a constructor for each class
 - What does every plant have in common? Put these in the Plant super class. Don't just use the plant definition above.
 - What does each subclass have in common? Put these in the appropriate subclass; don't just use definitions from above.
 - You may need to reorganize what goes where as you go.
 - Don't define any methods other than the constructor
 - Draw an 'egg' picture as you go if that is helpful.

Exercise

```
class Plant {  
    _____  
    _____  
    _____  
    _____  
    public Plant(_____ ) {  
        _____  
        _____  
        _____  
    }  
}  
class Tree extends _____ {  
    _____  
    _____  
    _____  
    public Tree(_____ ) {  
        _____  
        _____  
        _____  
    }  
}
```

Exercise

```
class Flower extends _____ {  
    _____  
    _____  
    public Flower(_____ ) {  
        _____  
    }  
}  
class Rose extends _____ {  
    _____  
    public Rose(_____ ) {  
        _____  
        _____  
    }  
}  
class Pine extends _____ {  
    _____  
    public Rose(_____ ) {  
        _____  
        _____  
    }  
}
```

Sample Solution

```
class Plant {
    private String kingdom;
    private String genus;
    private String species;
    private boolean annual;
    public Plant(String g, String s, boolean a) {
        kingdom= "Plantae";
        genus= g;
        species= s;
        annual = a;
    }
}
class Tree extends Plant {
    private double crownSize;
    private double trunkSize;
    public Tree(String g, String s, double cs, double ts) {
        super(g, s, false);
        crownSize= cs;
        trunkSize= ts;
    }
}
```

Sample Solution, p.2

```
class Flower extends Plant {
    private String blossomColor;
    public Flower(String g, String s, String bc, boolean a) {
        super(g, s, a);
        blossomColor= bc;
    }
}
class Rose extends Flower {
    private double thornDensity;
    public Rose(double td, String g, String s, String bc) {
        super(g, s, bc, false);
        thornDensity= td;
    }
}
class Pine extends Tree {
    private String needleType;
    private String coneType;
    public Pine(String g, String s, double cs, double ts,
        String nt, String ct) {
        super(g, s, cs, ts);
        needleType= nt;
        coneType= ct;
    }
}
```

Sample Solution, p.3

```
public class PlantTest {
    public static void main(String[] args) {
        Plant p= new Plant("PGenus", "PSpecies", false);
        Tree t= new Tree("TGenus", "TSpecies", 15.0, 2.0);
        Flower f= new Flower("FGenus", "FSpecies", "red", true);
        Rose r= new Rose(1.0, "RGenus", "RSpecies", "yellow");
        Pine pi= new Pine("PI Genus", "PI Species", 10.0, 1.0,
            "thin", "fat");
        System.exit(0);
    }
}

// Step through in debugger to show how constructors are called
```

Constructors

- **Subclass inherits constructors of super class**
 - **Constructors invoked in order of inheritance**

```
class Base{
    public Base() {
        System.out.println("Base"); } }
class Derived extends Base {
    public Derived() {
        System.out.println("Derived"); } }
class DerivedAgain extends Derived {
    public DerivedAgain() {
        System.out.println("Derived Again"); } }
public class Constructor1 {
    public static void main(String[] args) {
        DerivedAgain object1= new DerivedAgain();}}}
```

Output:

Base
Derived
DerivedAgain

Default constructor invoked unless
another constructor explicitly called.
Some constructor must be invoked.

Example of Class and Type

Suppose we have a class Tutorial that contains all students in a tutorial

```
import java.util.*;
public class Tutorial {
    ArrayList sArray = new ArrayList();
    String taName;
    public Tutorial(String ta) {
        taName = ta;}
    public void addToTutorial(Student s) {
        sArray.add(s);
    }
    //other methods here
}
```

Using Tutorial class

```
public class TestTutorial {
    public static void main(String[] args) {
        Undergrad u1 = new Undergrad(...);
        Grad g1 = new Grad(...);
        Tutorial t = new Tutorial( );
        t.addToTutorial(u1);
        t.addToTutorial(g1);

        // ...
    }
}
```