Advanced Programming Techniques in Java

Inheritance

Lecture 12

Class Objectives

- Inheritance & Constructors (Section 9.2)

- Overloading Vs Overriding
- `super`
- `protected`

Review: Inheritance

- Inheritance is an important concept of OOP and
  It allows a software developer to derive a new cl
- One class acquires the properties of another class
- Like a child inherits the traits of the parents

Review: Is-a relationships

- **Is-a relationship** is a hierarchical connection wh
  specialized version of another
- Every marketer **is-an** employee
- Every legal secretary **is-a** secretary

- **Inheritance hierarchy** is a set of classes connec
  share common code

Review: Inheritance

- **Syntax**

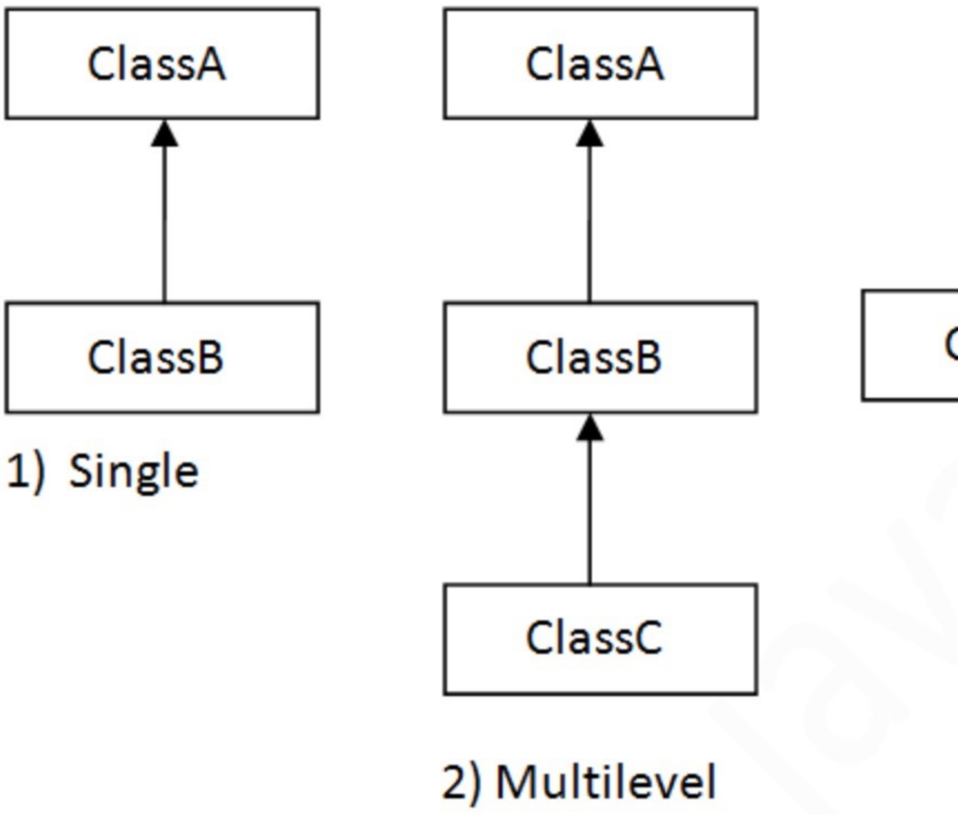
```
public class <subclass name> extends <superc
```

- **Example**

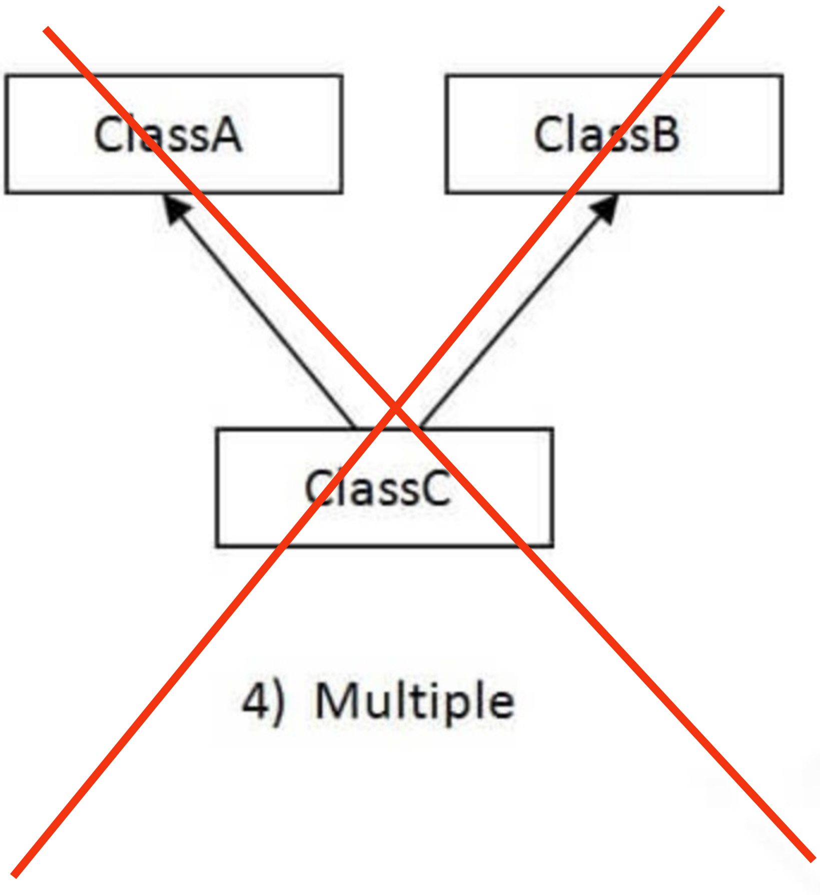
```
public class Secretary extends Employee
{ ... }
```

- By extending `Employee`, each `Secretary` ob
- Receives a `getHours`, `getSalary`, `getVacation` **automatically**
- Can be treated as an `Employee` by client code

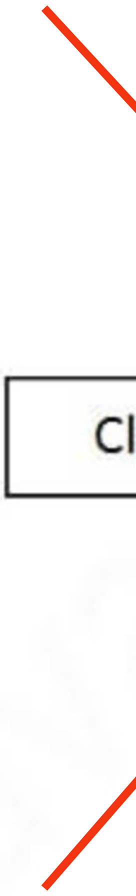
Types of inheritance in Java


ClassA
ClassB
1) Single
ClassA
ClassB
ClassC
2) Multilevel

Types of inheritance in Java



4) Multiple



`Employee` class (so far)

```
// A class to represent employees

public class Employee { public int
    return 40;              // works
    }

    public double getSalary() { re
        40000.0;       // $40,000.0
    }

    public int getVacationDays() {
        // 2 weeks' paid vacation
    }

    public String getVacationForm(
    "yellow";      // use the yello
}
```

Changes to common behavior

- Imagine a company-wide change affecting all em
- Everyone is given a $10,000 raise due to inflation
- The base employee salary is now $50,000
- Legal secretary now makes $55,000
- Marketer now
- We must modify our code to reflect this policy cha

Modifying the superclass

```
// A class to represent employees public class
Employee { public int getHours() { return 40;
// works 40 hours/week
    }

    public double getSalary() { return 50000.0;
    // $50,000.00/year }    ...
}
```

- Are we finished?

Modifying the superclass

```
// A class to represent employees public class
Employee { public int getHours() { return 40;
// works 40 hours/week
    }

    public double getSalary() { return 50000.0;
    // $50,000.00/year }     ...
}
```

- Are we finished?
- The Employee subclasses are still incorrect
- They have overridden `getSalary` to return other value

Marketer/Legal Secretar

```
// A class to represent marketers
public class Marketer extends Employee
{ public void advertise() {
        System.out.println("Act now whi
    }
     // overrides getSalary from Employ
    class public double getSalary() { r
    50000.0;
    }
}
```

```
// A class to represent legal secretari
public class LegalSecretary extends Sec
{ public void fileLegalBriefs() {
          System.out.println("I could fil
     }
     // overrides getSalary from Employe
     class public double getSalary() { r
     45000.0;
     }
}
```

An unsatisfactory solution

```
public class LegalSecretary extends Secretar
    public double getSalary() { return 55000
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() { return
    60000.0; }
    ...
}
```

- The subclasses' salaries are based on the Emplo
code does not reflect this

Calling overridden methods

- Subclasses can call overridden methods with the s
  keyword
- **Syntax** super.**method**(**parameters**)

```
public class LegalSecretary extends Sec
{ public double getSalary() { double
baseSalary = super.getSalary(); return
baseSalary + 5000.0; } ... }
```

More on Overriding

- A parent method can be invoked explicitly using th
- **If a method is declared with the `final` modifie**
- The concept of overriding can be applied to data a

Overloading vs. Overriding

- What is the difference between method **overloa**
- **Overloading**: one class contains multiple methods w signatures
- **Overriding**: a subclass substitutes its own version of same name and the same parameters
- Overloading lets you define a similar operation in diff
- Overriding lets you define a similar operation in differ

Improved subclasses

- Modify `Marketer` to use `super`

```
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now whil
    }

    public double getSalary() { return
    super.getSalary() + 10000.0; }
}
```

Inheritance and constructors

- Imagine that we want to give employees more vacat
company

- For each year worked, we'll award 2 additional vacation da
- When an `Employee` object is constructed, we'll pass in t
with the company
- This will require us to modify our `Employee` class and ac

Old `Employee` class

```
// A class to represent employees

public class Employee { public int
    return 40;            // works
    }

    public double getSalary() { re
        40000.0;       // $40,000.0
    }

    public int getVacationDays() {
        // 2 weeks' paid vacation
    }

    public String getVacationForm(
    "yellow";      // use the yello
}
```

New `Employee` class

```
public class Employee {

    private int years;

    public Employee(int initialYears)
        years = initialYears;
    }
    public int getHours() {
        return 40;
    }
    public double getSalary() {
        return 50000.0;
    }
    public int getVacationDays() {
        return 10 + 2 * years;
    }
    public String getVacationForm()
        return "yellow";
```

```
            }
  }
```

Problem with constructors

- Now that we've added the constructor to the `Emplo`
compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

- The short explanation: Once we write a constructor
superclass, we must now write constructors for our

Problem with constructors

- Constructors are not inherited
- Subclasses don't inherit the `Employee(int)` constr
  constructor that contains:

```
public Lawyer() { super();   //
callsEmployee()constructor }
```

- But `Employee(int)` replaces the default `Empl`
- The subclasses' default constructors are now trying to
  constructor

Calling superclass constructor

- **Syntax** `super(`**`parameters`**`);`

```
public class Lawyer extends Employee
Lawyer(int years) { super(years);  //
Employee constructor } ... }
```

- The `super` call must be the first statement in the constructor

The `super` reference

- Constructors are not inherited, even though they hav
- Yet we often want to use the parent's constructor to
- The `super` reference can be used to refer to the pa
  the parent's constructor
- A child’s constructor is responsible for calling the pa
- The first line of a child’s constructor should use the s
  constructor
- The `super` reference can also be used to reference
  in the parent’s class

Modified `Marketer` class

```
// A class to represent marketers public
class Marketer extends Employee { public
Marketer(int years) { super(years);
    }

    public void advertise() {
        System.out.println("Act now while
    }

    public double getSalary() { return
        super.getSalary() + 10000.0;
    }
}
```

- Modify the `Secretary` subclass
- Secretaries' years of employment are not tracked. The
worked

Modified `Secretary` class

- Modify the `Secretary` subclass
- Secretaries' years of employment are not tracked. The
  worked

```
// A class to represent secretaries
public class Secretary extends Employee
public Secretary() { super(0);
    }

    public void takeDictation(String te
        System.out.println("Taking dict
    }
}
```

- Since `Secretary` doesn't require any paramet
  `LegalSecretary` compiles without a construc

- Its default constructor calls the `Secretary()` constr

Which constructors are called?

```
public Employee(int initialYears) {
        years = initialYears;
}

public Lawyer(int years) {
        super(years);
}

public Secretary() {
        super(0);
}

public LegalSecretary() {
        super(); // implicit call
}
```

C

N
Ja

Constructors and Inheritance

```
Employee emp = new Employee();
```

- Creates an `Employee` object by calling its constru

- When you instantiate an object of a subclass
- Constructor of the super class (executes first)
- The constructor of the subclass (executes second

```
Secretary sec = new Secretary();
```

- This calls `Employee()` and then `Secretary()`

Things to remember

- You cannot directly call the constructor of the sup
  - Java allows you to use only super()

```
public Employee(int initialYears) {
    years = initialYears;
}
```

Things to remember (cont.)

- The super() statement must be the first statem
  that uses it. Not even data field definitions can pre

```
public Lawyer(int years) {
 int x; super(years);
}
```

It is not legal

```
public Lawyer(int years) { super(years); in
```

It is legal

```
}
```

Em

```
public class Employee { private int
     years;    public Employee(int
     initialYears) { years =
     initialYears;
     } public int getHours()
     { return 40;
     } public double getSalary()
     { return 50000.0;
     } public int getVacationDays()
     { return 10 + 2 * years;
     } public String getVacationForm
     { return "yellow";
     }
}
```

Ma

```
public class Marketer extends Empl
    public Marketer(int years) {
    super(years);
    }

    public void advertise() {
        System.out.println("Act no
    }

    public double getSalary() { re
    super.getSalary() + 10000.0; }
}
```

Se

```
public class Secretary extends Employe
     { public Secretary() { super(0);
     }

     public void takeDictation(String t
          System.out.println("Taking dic
     }
}
```

Inheritance and fields

- Try to give lawyers $5000 for each year

```
public class Lawyer extends Emplo
    ... public double getSalary()
    super.getSalary() + 5000 * yea
    ...
}
```

- Does not work; the error is the following:

```
Lawyer.java:7: years has private acc
            return super.getSalary() + 5
```

- `private` fields cannot be directly acce
- One reason: So that subclass can't break en around this limitation?

Inheritance & Information

- When class serves as superclass
- Subclasses inherit all data and methods
- Except `private` members of parent cla
  class's methods
- `private` fields can be accessible throu

`Employee` class with “getter”

```
public class Employee {
    private int years;

    public Employee(int initial
        years = initialYears;
    }

    public int getYears()
    { return years; }
    ...
}

public class Lawyer extends Emp
    public Lawyer(int years) {
    super(years);
    }

    public double getSalary() {
    super.getSalary() + 5000 *
    ...
}
```

One more level of information hic

- Keyword `protected`
- Provides intermediate level of security b access
- Allows a member of a superclass to be i
- Can be used within own class or in any
- Cannot be used by “outside” classes

- When might you need it? (RARELY
- If you want your fields to be `private` b accessor method
- `public` methods can be used by EVEI

So far ... `Secretary` class

```
// A class to represent secretaries
public class Secretary extends Employee
public Secretary() { super(0);
    }

    public void takeDictation(String tex
        System.out.println("Taking dicta
    }
}
```

Revisiting `Secretary`

- The `Secretary` class currently has

- We set all Secretaries to 0 years becaus
their service
- If we call `getYears` on a `Secretary`
- This isn't a good solution; what if we wan
employees based on years of service?

- Redesign our `Employee` class to all

Employee class so far

```
public class Employee {
    private int years;

    public Employee(int initialYear
        years = initialYears;
    }
    public int getHours() {
        return 40;
    }
    public double getSalary() {
        return 50000.0;
    }
    public int getVacationDays() {
        return 10 + 2 * years;
    }
    public String getVacationForm()
        return "yellow";
    }
}
```

Improved `Employee` class

- Let's separate the standard 10 vacation d
  based on seniority

```
public class Employee {
    private int years;
    public Employee(int initialYears)
        years = initialYears;
    }
    public int getVacationDays() {
        return 10 + getSeniorityBonus
    }

    public int getSeniorityBonus() {
    return 2 * years; }
    ...
}
```

- How does this help us improve the `Secr`

Improved `Secretary` class

- `Secretary` can selectively override ge
`getVacationDays` runs, it will use the n
- Choosing a method at runtime is called **dyna**

```
public class Secretary extends Emplo
    public Secretary(int years) {
    super(years);
    }

    // Secretaries don't get a bonus
    public int getSeniorityBonus() {
    }

    public void takeDictation(String
        System.out.println("Taking d
    }
}
```

```
public class Employee {

   private int years;
   public Employee(int initialYears) {
        years = initialYears;
        System.out.println("Creating a r
   }
   public Employee() {
        System.out.println("Default cons
   }
   public int getHours() { return 40;
           40 hours / week
   }
   public double getSalary() { return 40
           // $40,000.00 / year
   }
   public int getVacationDays() { return
           2 weeks' paid vacation
   }
   public int getSeniorityBonus(){
           return 2*years;
   }
   public String getVacationForm() { ret
           // use the yellow form
        }
}
```

Fina

Final `Secretary` class

```
public class Secretary extends Employee

   public Secretary(int years) { super(
           calls Employee constructor
        System.out.println("Creating a
   }

   public int getSeniorityBonus(){
          return 0;
   }

   public void takeDictation(String tex
          System.out.println("Taking di
   }
}
```

Final Lawyer class

```
public class Lawyer extends Employee

    public Lawyer(int years) { super
    calls Employee constructor }

    // overrides getVacationForm fro
    public String getVacationForm()
    }

    // overrides getVacationDays fro
    int getVacationDays() { return s
    // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see
    }
}
```

Final LegalSecretary class

```
public class LegalSecretary extends Sec

    public LegalSecretary(int years){
        super(years);
    }
    public void fileLegalBriefs() {
        System.out.println("I could fil
    }

    public double getSalary() { return
        super.getSalary()+5000.0;   // S
    }
}
```

Packages and

Packages

- A Java *package* is a group of *cooperating cla*
- The Java API is organized as packages
- Indicate the package of a class at the top of t

```
package classPackage;
```

- Classes in the *same package* should be in th
- The folder must have the same name as the
- Classes in the *same folder* must be in the *sa*

Packages and Visibility

- Classes *not* part of a package can only members of classes in the package
- If a class is not part of the package, the public classes by their complet would be
  `packagename.className`
- For example, `x = Java.awt.Color`
- If the package is imported, the `packag` required.
  ```
  import java.awt.Color;
  ```

```
... x =
Color.GREEN;
```

The Default Package

- Files which do not specify a package
  package
- If you do not declare packages, all your
  the default package
- The default package is intended for
  stages of implementation or for smal

- When you develop an application, declare i
  the same package

Visibility

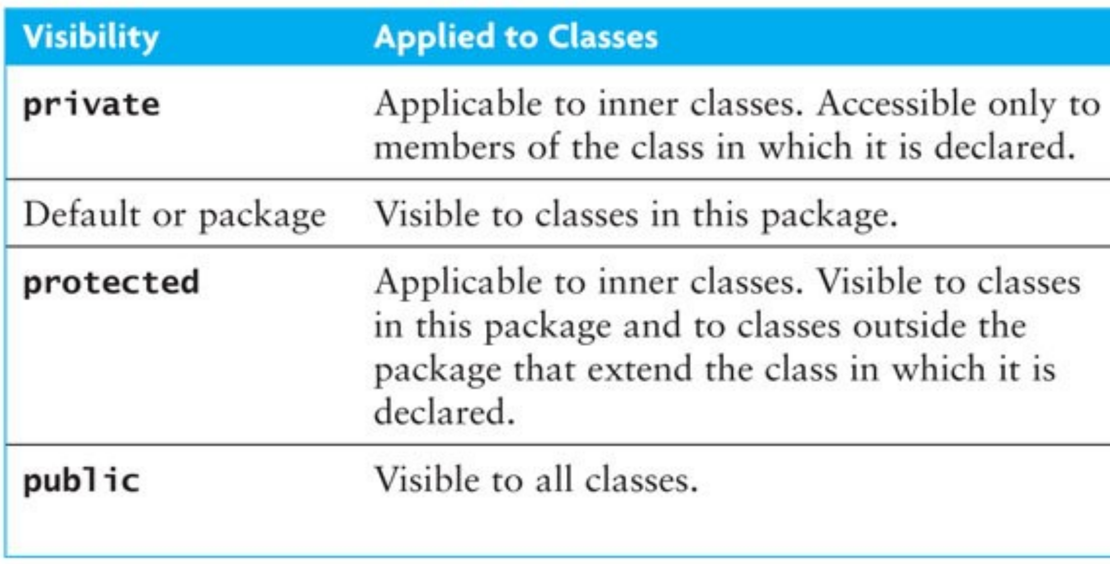
- We have seen three visibility l
  `protected`, `private`
- A fourth layer, *package visibility*, lies l
  `protected`

- Classes, data fields, and methods
  are accessible to all other methods
  package, but are not accessible to
  package
- Classes, data fields, and methods
  protected are visible within subclasse
  outside the package (in addition to
  members *inside* the package)
- There is no keyword to indicate pack
- Package visibility is the default in
  `protected`, `private` are not used

Visibility Supports Encapsulation

- Visibility rules enforce encapsulation i
- `private`:for members thatshould be i
  subclasses
- package: shields classes and members f
  the package
- `protected`: provides visibility to c
  in the package
- `public`: provides v

Visibility Supports Encapsulation



Visibility	Applied to Classes
`private`	Applicable to inner classes. Accessible only to members of the class in which it is declared.
Default or package	Visible to classes in this package.
`protected`	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared.
`public`	Visible to all classes.

Visibility Supports Encapsulation

- Encapsulation insulates against change
- Greatervisibility means less encapsulatio
- *So*... use the most restrictive visibility
  job done!