

Advanced Programming Techniques in Java



COSI 12B

Inheritance



Lecture 12



Class Objectives

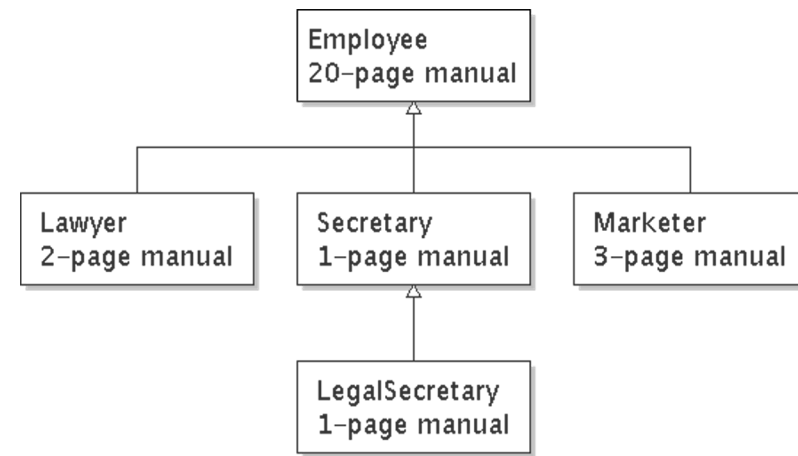
- Inheritance & Constructors (Section 9.2)
- Overloading Vs Overriding
- `super`
- `protected`



Review: Inheritance

- Inheritance is an important concept of OOP and **promotes software reusability**
- It allows a software developer to derive a new class from an existing one
 - 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 where one category can be treated as a specialized version of another
 - Every marketer **is-an** employee
 - Every legal secretary **is-a** secretary
- **Inheritance hierarchy** is a set of classes connected by is-a relationships that can share common code



Review: Inheritance

- **Syntax**

```
public class <subclass name> extends <superclass name> {
```

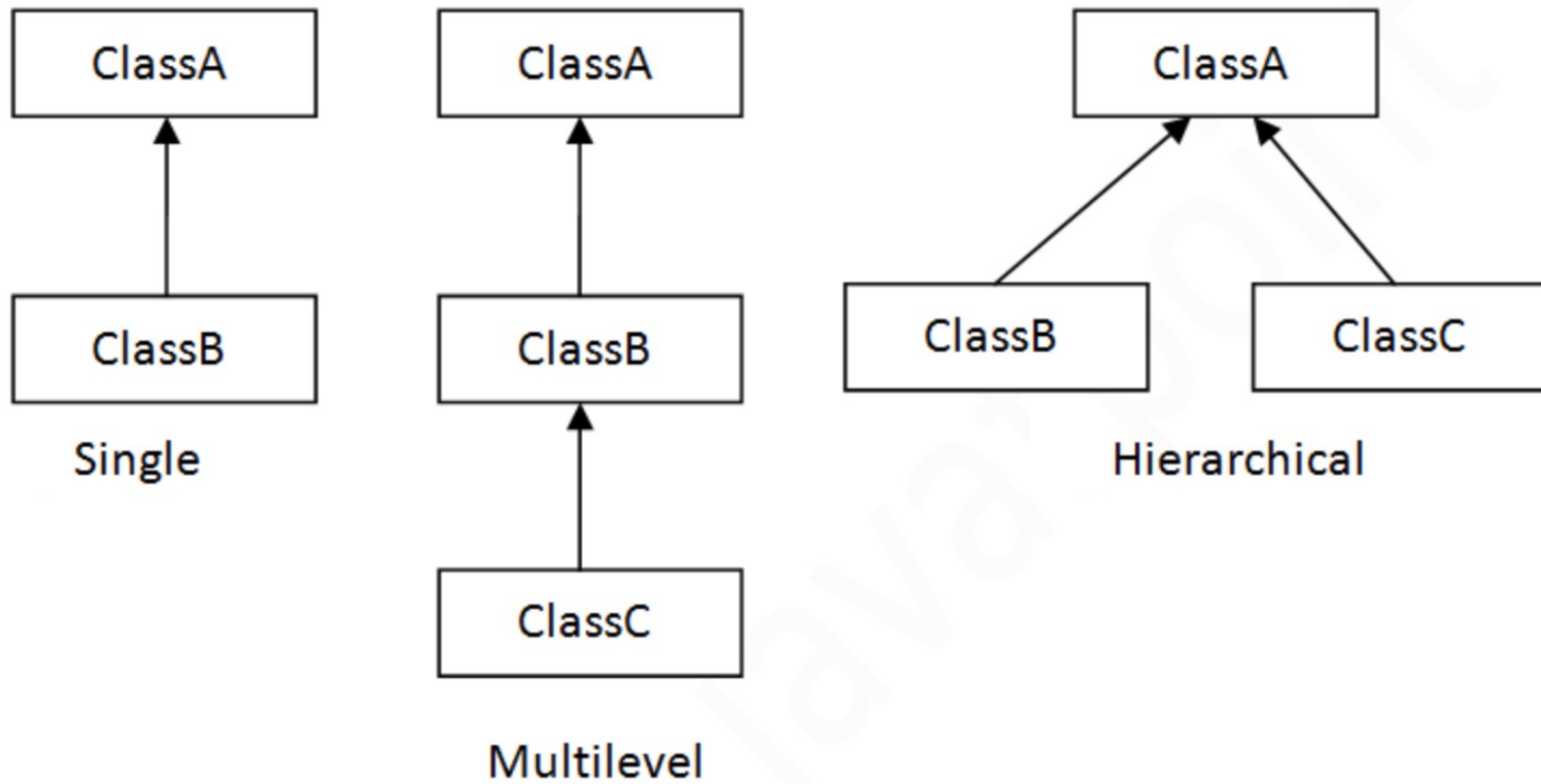
- **Example**

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

- By extending `Employee`, **each** `Secretary` **object** now:
 - Receives a `getHours`, `getSalary`, `getVacationDays`, **and** `getVacationForm` method **automatically**
 - Can be treated as an `Employee` by client code

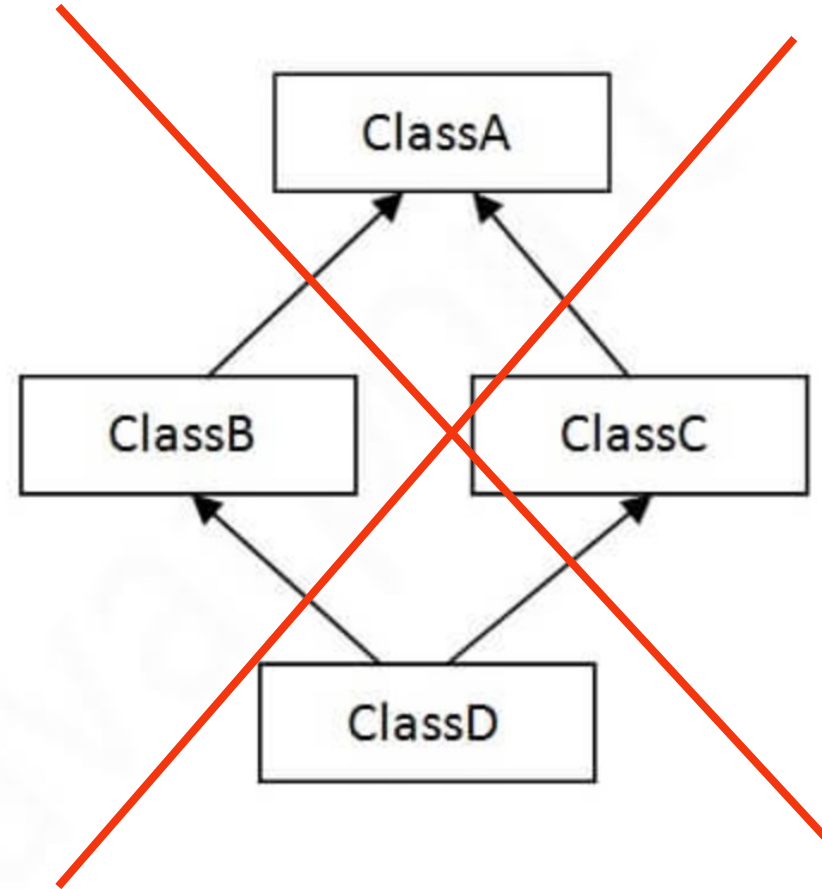
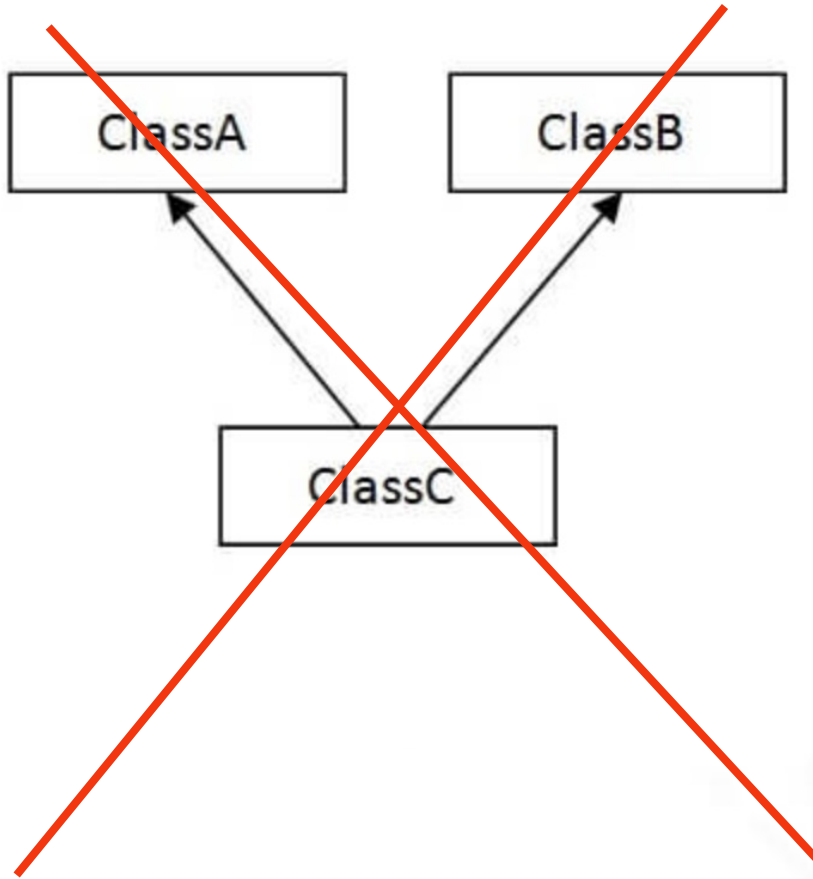


Types of inheritance in Java





Types of inheritance in Java





Employee class (so far)

Employee.java

```
// A class to represent employees in general

public class Employee {
    public int getHours() {
        return 40;           // works 40 hours/week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00/year
    }

    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}
```



Changes to common behavior

- Imagine a company-wide change affecting all employees
- 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 makes \$60,000
- We must modify our code to reflect this policy change



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 values



Marketer/Legal Secretary class

```
// A class to represent marketers
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    // overrides getSalary from Employee class
    public double getSalary() {
        return 50000.0;
    }
}
```

```
// A class to represent legal secretaries
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }
    // overrides getSalary from Employee class
    public double getSalary() {
        return 45000.0;
    }
}
```



An unsatisfactory solution

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        return 55000.0;  
    }  
    ...  
}  
  
public class Marketer extends Employee {  
    public double getSalary() {  
        return 60000.0;  
    }  
    ...  
}
```

- The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this



Calling overridden methods

- Subclasses can call overridden methods with the `super` keyword
- **Syntax**

`super.method(parameters)`

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



More on Overriding

- A parent method can be invoked explicitly using the `super` reference
- **If a method is declared with the `final` modifier, it cannot be overridden**
- The concept of overriding can be applied to data and is called *shadowing variables*



Overloading vs. Overriding

- What is the difference between method **overloading** and method **overriding**?
 - **Overloading**: one class contains multiple methods with the same name but different parameter signatures
 - **Overriding**: a subclass substitutes its own version of an otherwise inherited method, with the same name and the same parameters
 - Overloading lets you define a similar operation in different ways for different data
 - Overriding lets you define a similar operation in different ways for different object types



Improved subclasses

- Modify Marketer to use super

```
public class Marketer extends Employee {  
    public void advertise() {  
        System.out.println("Act now while supplies last!");  
    }  
  
    public double getSalary() {  
        return super.getSalary() + 10000.0;  
    }  
}
```



Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they've been with the company
 - For each year worked, we'll award 2 additional vacation days
 - When an `Employee` object is constructed, we'll pass in the number of `years` the person has been with the company
 - This will require us to modify our `Employee` class and add some new state and behavior



Old Employee class

```
// A class to represent employees in general

public class Employee {
    public int getHours() {
        return 40;           // works 40 hours/week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00/year
    }

    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}
```



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 `Employee` class, our subclasses do not 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 (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well



Problem with constructors

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

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

- But `Employee(int)` replaces the default `Employee()`
 - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor



Calling superclass constructor

- **Syntax**

`super(parameters);`


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

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



The `super` reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined 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 supplies last!");
    }

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

- Modify the `Secretary` subclass
 - Secretaries' years of employment are not tracked. They do not earn extra vacation for years worked



Modified Secretary class

- Modify the `Secretary` subclass
 - Secretaries' years of employment are not tracked. They do not earn extra vacation for years worked

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

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

- Since `Secretary` doesn't require any parameters to its constructor, `LegalSecretary` compiles without a constructor
 - Its default constructor calls the `Secretary()` constructor

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  
}
```

Calls the constructor of the superclass

NOTE: You cannot call `Employee(years)`
Java allows to use only `super(years)`

Calls the constructor of the superclass



Constructors and Inheritance

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

- Creates an `Employee` object by calling its constructor
- When you instantiate an object of a subclass you call at least two constructors:
 - 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 superclass
 - Java allows you to use only `super()`

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

```
public Lawyer(int years) {  
    Employee(years);  
}
```

It is not legal, you should call
`super(year)`



Things to remember (cont.)

- The `super()` statement must be the first statement in any subclass constructor that uses it. Not even data field definitions can precede it

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

It is not legal

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

It is legal




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";  
    }  
}
```




Marketer class

```
public class Marketer extends Employee {  
    public Marketer(int years) {  
        super(years);  
    }  
  
    public void advertise() {  
        System.out.println("Act now while supplies last!");  
    }  
  
    public double getSalary() {  
        return super.getSalary() + 10000.0;  
    }  
}
```



Secretary class

```
public class Secretary extends Employee {  
    public Secretary() {  
        super(0);  
    }  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```



Inheritance and fields

- Try to give lawyers \$5000 for each year at the company:

```
public class Lawyer extends Employee {  
    ...  
    public double getSalary() {  
        return super.getSalary() + 5000 * years;  
    }  
    ...  
}
```

- Does not work; the error is the following:

```
Lawyer.java:7: years has private access in Employee  
        return super.getSalary() + 5000 * years;  
                                   ^
```

- `private` fields cannot be directly accessed from subclasses
 - One reason: So that subclass can't break encapsulation
 - How can we get around this limitation?



Inheritance & Information Hiding

- When class serves as superclass
 - Subclasses inherit all data and methods of superclass
 - Except `private` members of parent class not accessible within child class's methods
 - `private` fields can be accessible through accessor methods



Employee class with "getter"

```
public class Employee {
    private int years;

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

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

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

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



One more level of information hiding

- Keyword `protected`
 - Provides intermediate level of security between `public` and `private` access
 - Allows a member of a superclass to be inherited into a subclass
 - Can be used within own class or in any classes extended from that class
 - Cannot be used by “outside” classes
- When might you need it? (RARELY USED)
 - If you want your fields to be `private` but you don't want to have a public accessor method
 - `public` methods can be used by EVERY CLASS not only the subclasses



So far ... Secretary class

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

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```



Revisiting Secretary

- The `Secretary` class currently has a poor solution
 - We set all Secretaries to 0 years because they do not get a vacation bonus for their service
 - If we call `getYears` on a `Secretary` object, we'll always get 0
 - This isn't a good solution; what if we wanted to give some other reward to all employees based on years of service?
- Redesign our `Employee` class to allow for a better solution



Employee class so far

```
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";  
    }  
}
```



Improved Employee class

- Let's separate the standard 10 vacation days from those that are awarded 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 Secretary?



Improved Secretary class

- Secretary **can selectively override** `getSeniorityBonus`; when `getVacationDays` runs, it will use the new version
 - Choosing a method at runtime is called **dynamic binding**

```
public class Secretary extends Employee {  
    public Secretary(int years) {  
        super(years);  
    }  
  
    // Secretaries don't get a bonus for their years of service  
    public int getSeniorityBonus() {  
        return 0;  
    }  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

Final Employee class

```
public class Employee {

    private int years;
    public Employee(int initialYears) {
        years = initialYears;
        System.out.println("Creating a new employee.");
    }
    public Employee() {
        System.out.println("Default constructor.");
    }
    public int getHours() {
        return 40;           // works 40 hours / week
    }
    public double getSalary() {
        return 40000.00;      // $40,000.00 / year
    }
    public int getVacationDays() {
        return 10+getSeniorityBonus(); // 2 weeks' paid vacation
    }
    public int getSeniorityBonus(){
        return 2*years;
    }
    public String getVacationForm() {
        return "yellow";      // use the yellow form
    }
}
```



Final Secretary class

```
public class Secretary extends Employee {  
  
    public Secretary(int years) {  
        super(years); // calls Employee constructor  
        System.out.println("Creating a new Secretary.");  
    }  
  
    public int getSeniorityBonus(){  
        return 0;  
    }  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```



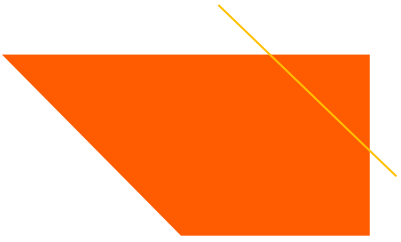
Final Lawyer class

```
public class Lawyer extends Employee {  
  
    public Lawyer(int years) {  
        super(years); // calls Employee constructor  
    }  
  
    // overrides getVacationForm from Employee class  
    public String getVacationForm() {  
        return "pink";  
    }  
  
    // overrides getVacationDays from Employee class  
    public int getVacationDays() {  
        return super.getVacationDays()+5; // 3 weeks vacation  
    }  
  
    public void sue() {  
        System.out.println("I'll see you in court!");  
    }  
}
```



Final LegalSecretary class

```
public class LegalSecretary extends Secretary {  
  
    public LegalSecretary(int years){  
        super(years);  
    }  
    public void fileLegalBriefs() {  
        System.out.println("I could file all day!");  
    }  
  
    public double getSalary() {  
        return super.getSalary()+5000.0;    // $45,000.00 / year  
    }  
}
```



Packages and Visibility



Packages

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

```
package classPackage;
```
- Classes in the *same package* should be in the *same directory* (folder)
- The folder must have the same name as the package
- Classes in the *same folder* must be in the *same package*



Packages and Visibility

- Classes *not* part of a package can only access `public` members of classes in the package
- If a class is not part of the package, it must access the public classes by their complete name, which would be `packageName.className`
- For example,

```
x = Java.awt.Color.GREEN;
```
- If the package is imported, the `packageName` prefix is not required.

```
import java.awt.Color;  
...  
x = Color.GREEN;
```



The Default Package

- Files which do not specify a package are part of the default package
- If you do not declare packages, all your classes belong to the default package
- The default package is intended for use during the early stages of implementation or for small prototypes
- When you develop an application, declare its classes to be in the same package



Visibility

- We have seen three visibility layers, `public`, `protected`, `private`
- A fourth layer, *package visibility*, lies between `private` and `protected`
- Classes, data fields, and methods with package visibility are accessible to all other methods of the same package, but are not accessible to methods outside the package
- Classes, data fields, and methods that are declared `protected` are visible within subclasses that are declared *outside* the package (in addition to being visible to all members *inside* the package)
- There is no keyword to indicate package visibility
- Package visibility is the default in a package if `public`, `protected`, `private` are not used



Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in Java
- `private`: for members that should be invisible even in subclasses
- `package`: shields classes and members from classes outside the package
- `protected`: provides visibility to extenders or classes in the package
- `public`: provides visibility to all



Visibility Supports Encapsulation (cont.)

Visibility	Applied to Classes	Applied to Class Members
private	Applicable to inner classes. Accessible only to members of the class in which it is declared.	Visible only within this class.
Default or package	Visible to classes in this 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.	Visible to classes in this package and to classes outside the package that extend this class.
public	Visible to all classes.	Visible to all classes. The class defining the member must also be public.



Visibility Supports Encapsulation (cont.)

- Encapsulation insulates against change
- Greater visibility means less encapsulation
- So... use the most restrictive visibility possible to get the job done!