Employee class

```
// A class to represent employees
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
    }
}
```


Shape classes

- Write a class called Rectangle with a width, a length, and a method for calculating the area. Include a constructor.
- Write a Square class with a width, a method for calculating the area, and a constructor.
- Write a Triangle class with lengths for each side. Include a constructor, and a method for calculating the angle between two sides.
- Write an EquilateralTriangle class …

Some Object-Oriented Programming (OOP) Review

Let's practice writing some classes

- Write an `Employee` class with methods that return values for the following properties of employees at a particular company:
  - Work week: 40 hours
  - Annual salary: $40,000
  - Paid time off: 2 weeks
  - Leave of absence form: Yellow form

Inheritance

Writing more classes

- Write a `Secretary` class with methods that return values for the following properties of secretaries at a particular company:
  - Work week: 40 hours
  - Annual salary: $40,000
  - Paid time off: 2 weeks
  - Leave of absence form: Yellow form
- Add a method `takeDictation` that takes a string as a parameter and prints out the string prefixed by "Taking dictation of text: ".

`Secretary` class

```
// A class to represent secretaries
public class Secretary {
    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
    }

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


How are they similar?

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

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public String getVacationForm() {
        return "yellow";
    }
}
```

```
// A class to represent secretaries
public class Secretary {
    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public String getVacationForm() {
        return "yellow";
    }

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

Inheritance

- **inheritance**: A way to specify a relationship between two classes where one class *inherits* the state and behavior of another.
- The *child* class (also called subclass) inherits from the *parent* class (also called superclass).
- The subclass receives a copy of every field and method from the superclass.

Inheritance syntax

- Creating a subclass, general syntax:

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

- Example:

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

- By extending `Employee`, each `Secretary` object automatically has a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method.

Is-a relationship

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
- Examples:
  - Every secretary is an employee.
  - Every square is a rectangle.
  - Every dog is a mammal.

Reusing code: why re-invent the wheel?

- **code reuse**: The practice of writing program code once and using it in many contexts.
- We'd like to be able to say the following:

```
// A class to represent secretaries
public class Secretary {
    <copy all the contents from Employee class>

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

- That way we would be ***reusing*** the `Employee` code.

Overriding methods

- **override**: To write a new version of a method in a subclass to replace the superclass's version.
- To override a superclass method, just write a new version of it in the subclass. This will replace the inherited version.

Marketer class

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

    public double getSalary() {
      //Employee e = new Employee();

        return 50000.0
  //  + e.getSalary();          // $50,000.00 / year
    }
}
```

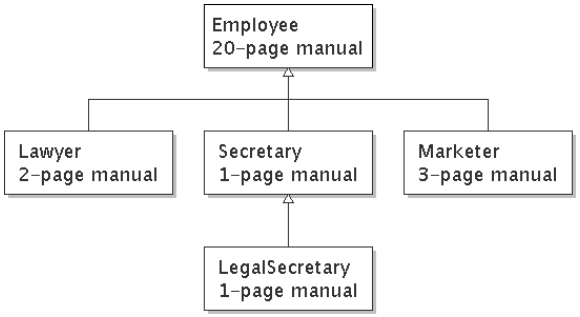

Improved Secretary class

```
// A class to represent secretaries
public class Secretary extends Employee
{
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: "
                           + text);
    }
}
```


Writing even more classes

- Write a Marketer class that represents marketers who have the same properties as general employees, but instead of making only a paltry $40,000, marketers make $50,000!
- Can we still leverage the Employee class or do we have to re-write everything, because one method (getSalary) is different?
- If only Marketer could write a new version of the getSalary method, but inherit everything else…

Why bother with separate manuals?



- Why not just have a 22-page manual for lawyers, 21-page manual for secretaries, 23-page manual for marketers, etc…?

Advantages of separate manuals

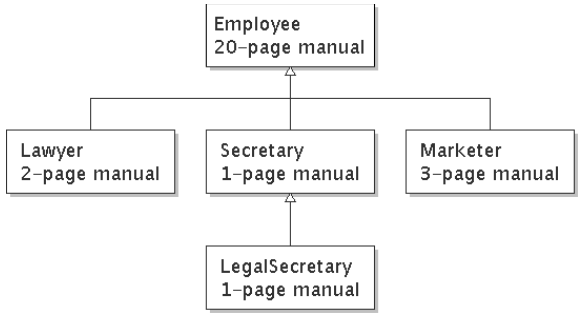
- maintenance: If a common rule changes, only the common manual needs to be updated.
- locality: A person can look at the manual for lawyers and quickly discover all rules that are specific to lawyers.

Based in reality or too convenient?

- At many companies, all new employees attend a common orientation to learn general rules (e.g., what forms to fill out when).
- Each person receives a big manual of these rules.
- Each employee also attends a subdivision-specific orientation to learn rules specific to their subdivision (e.g., marketing department).
- Everyone receives a smaller manual of these rules.

Rules, rules, everywhere

- The smaller manual adds some rules and also changes (read: overrides) some rules from the large manual (e.g., "use the pink form instead of the yellow form")



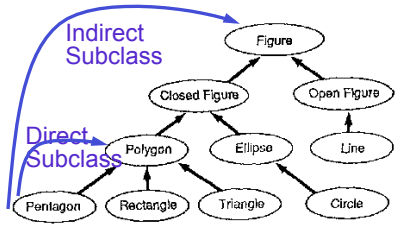
Solution: `LegalSecretary`

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

    public double getSalary() {
        return 45000.0;          // $45,000.00 / year
    }
}
```


Inheritance hierarchies

- Deep hierarchies can be created by multiple levels of subclassing.
- **inheritance hierarchy**: A set of classes connected by is-a relationships that can share common code.



Key ideas

- It is useful to be able to specify general rules that will apply to many groups (the 20-page manual).
- It is also useful to specify a smaller set of rules for a particular group, including being able to replace rules from the overall set (e.g., "use the pink form instead of the yellow form").

Exercise: `LegalSecretary`

- Write a `LegalSecretary` class that represents legal secretaries—a special type of secretary that can file legal briefs. Legal secretaries also earn more money ($45,000).

Object

Exercise: `Lawyer`

- Lawyers are employees that know how to sue. They get an extra week of paid vacation (a total of 3) and have to use the pink form when applying for vacation leave. Write the `Lawyer` class.

Some Questions about Inheritance

- All classes in Java are direct or indirect subclasses of one class. What is it?
  - The `Object` class
- Explain why the designers of Java decided this was a good idea.
  - ***All*** classes share functionality! What is it?

Solution: `Lawyer`

```
// A class to represent lawyers
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;          // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

Example: Course

```
public class Course {
   String [] students;
   public Course(String [] students)
   { this.students = students; }

   public static void main(String [] args)
   throws CloneNotSupportedException {
        String [] s1 = {"Jill","Jim","Joe"};
        String [] s2 = {"Jill","Jim","Joe"};
        Course c1 = new Course(s1);
        Course c2 = new Course(s2);
        System.out.println(c1.toString());
        System.out.println("c1 equals c2:  " + c1.equals(c2));
        Course c3 = c1.clone();
        System.out.println("c1 == c3:  " + (c1==c3));
        System.out.println("c1 equals c3:  " + c1.equals(c3));
   }
}
```

Output of Course

```
> java Course
Course@187aeca
c1 equals c2:  false
Exception in thread "main"
  java.lang.CloneNotSupportedExcept
  ion:  Course
…
```

The `Object` Class

Method	Description
String toString()	Returns a String description of the object.
boolean equals(Object other)	Returns true if **this** and **other** point to the same address.
Object clone()	Returns a copy of **this**.
Class getClass()	Returns a Class object representing the data type (Class) of **this**.
Several others	See Java API documentation …

Because every class automatically *extends* Object, and because Object defines these methods,

→ **every class automatically has these methods!**

A closer look at these methods

- The Object class already defines toString()
- And every class inherits from Object
- *So why do we ever define our own toString()?*
  - Because the default Object methods are basically placeholders!
  - They don't do anything very useful.

Overriding the equals method, Take 2

- Ok, how about this?

```
public boolean equals(Course other)
{
    if(other==null) {
        return false;
    }
    return this.students == other.students;
}
```

Overriding the equals method, Take 3

- Ok, how about this?

```
public boolean equals(Course other)
{
    if(other==null) {
        return false;
    }
    if(this.students==null && other.students==null) {
        return true;
    }
    if(this.students!=null) {
        return this.students.equals(other.students);
    }
    return false;
}
```

Overriding

- It's a good idea to override Object methods for any new class!
- Example: overriding toString()

```
import java.util.Arrays;
…
public String toString()
{
    return "Student list:  " + Arrays.toString(students);
}
```

Overriding the equals method

- toString() is the easy case. Let's take a look at equals()
- Will this work?

```
public boolean equals(Course other)
{
    return this==other;
}
```

So super

Constructor for superclass

```
public class Employee {
    private double salary;
    public Employee(double initialSalary)
    {
        salary = initialSalary;
    }
    public int getHours() {
        return 40;     // 40 hours per week
    }
    public double getSalary() {
        return salary;
    }
    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }
    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}
```


Deep vs. Shallow Comparisons

- A shallow comparison
  - checks to see if two objects have fields with the same ***references***:

```
this.students == other.students
```

- A deep comparison
  - checks to see if two objects have fields with the same ***contents***:

```
this.students.equals(other.students)
```

Exercise: Overriding clone()

- First, override the clone() method for Course so that if we call Course c2 = c1.clone(), then c2.students == c1.students (shallow clone).
- Now, override the clone() method for Course so that if we call Course c2 = c1.clone(), then c2.equals(c1), but c2.students != c1.students (deep clone).

protected

Do you have protection?

- Recall: there are four access specifiers:
  `public private protected` <default>
- Question: If a method is declared private, does a subclass inherit it?
  - Actually, yes. Subclasses inherit everything that they don't override.
- If a method is declared private, can a subclass *call* it?
  - NO! Only code inside the same class can call a private method.
- What if you want a subclass to be able to use it?
  - Use the `protected` access level

Constructors of subclasses

Use the `super()` method to call the superclass's constructor

```
public class Marketer extends Employee {
    // inherits double salary

    public Marketer(double initialSalary)
    {
        //construct superclass
        super(initialSalary);
    }
}
```

- For every constructor of a subclass, the call to super() must be the first statement in the subclass's constructor.

- Make sure to give the same number of arguments as there are parameters in the definition of the superclass's constructor.

Exercise: Subclass constructors

- Write a new version of the Secretary subclass that extends the new Employee class. All Secretaries should make $40,000 per year, and they should all have a takeDictation() method.

- Write a new version of the Square class that extends the Rectangle class. Make sure that the length and width of a Square are always equal!

Access Specifier Example, continued

- If we want subclasses (and nothing else) to see something, make it `protected`:

```
public class Employee {
    protected double salary = 40000.00;

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

    public double getSalary() {
        return salary;
    }

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

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


Access Specifier Example, continued

- Subclasses can see `protected` variables and methods just fine.

```
public class CEO extends Employee {
    public void giveMyselfRaise() {
        salary += 1000000.00;    // No longer an error
    }

    public static void main(String [] args)
    {
        CEO c = new CEO();
        // This is fine, no error here
        // Access to salary field is indirect
        // We're accessing the public getSalary() method
        System.out.println("My salary is " + c.getSalary());
    }
}
```


Access Specifier Example

- Recall our new Employee class

```
public class Employee {
    private double salary = 40000.00;

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

    public double getSalary() {
        return salary;
    }

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

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


Access Specifier Example, continued

- Subclasses cannot see `salary` directly!

```
public class CEO extends Employee {

    public void giveMyselfRaise() {

        salary += 1000000.00;    // Compile-time Error!

    }

    public static void main(String [] args)
    {
        CEO c = new CEO();
        // This is fine, no error here
        // Access to salary field is indirect
        // We're accessing the public getSalary() method
        System.out.println("My salary is " + c.getSalary());
    }
}
```

What would happen if ....

```
public class Employee {
    private double salary = 40000.00;

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

    public double getSalary() {
        return salary;
    }

    public void addToSalary(double raise) {
        salary += raise;
    }

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

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


What would happen if ... (continued)

```
public class CEO extends Employee {

    public void giveMyselfRaise() {

        addToSalary(1000000.00); // Error??

    }
}
```

- CEO still has its own copy of the salary field, and this code will change the value of it appropriately.
- The fact that salary is private simply means that CEO can't access it *directly*. It can still call public (or protected) superclass methods that can access it.