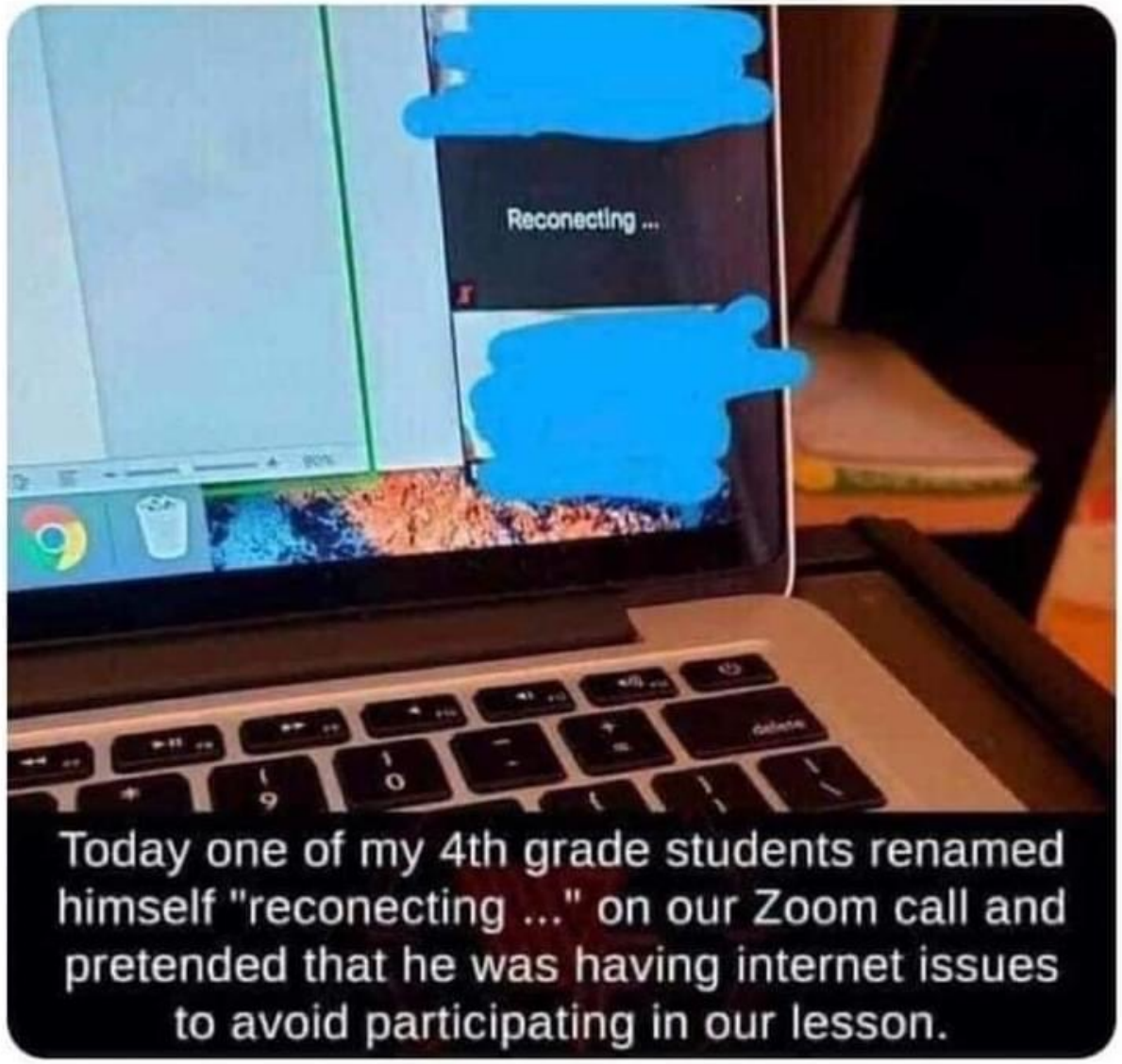
# Advanced Programming Techniques in Java

COSI 12B

# The future of IT is in good hands.




Zoom Classes

# Object Oriented Programming V

Lecture 11

# Class Objectives

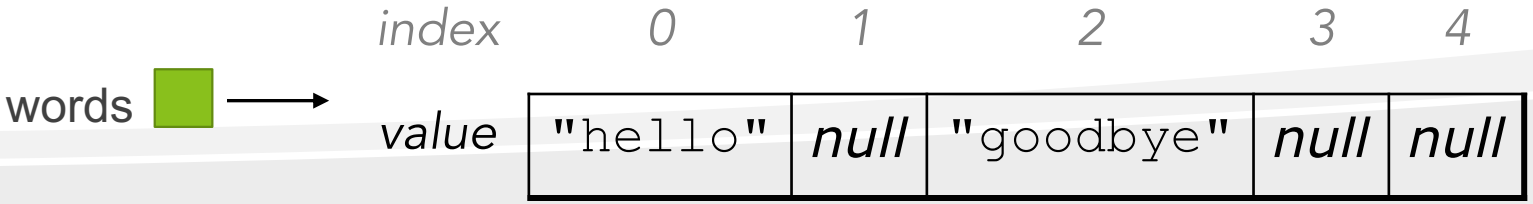
- OOD
- Inheritance Basics (9.1)

# Review: Looking before you leap

- You can check for `null` before calling an object's methods

```
String[] words = new String[5];
words[0] = "hello";
words[2] = "goodbye";    // words[1], [3], [4] are null
```

```
for (int i = 0; i < words.length; i++) {
    if (words[i] != null) {
        words[i] = words[i].toUpperCase();
    }
}
```



# Review: Method Overloading

- There are three ways to overload a method
  - Number of parameters

    ```
    add (int, int)
    add (int, int, int)
    ```

  - Data type of parameters

    ```
    add (int, int)
    add (int, double)
    ```

  - Sequence of data type of parameters

    ```
    add(int, double)
    add(double, int)
    ```

# Review: Encapsulation

- Encapsulation is a principle of wrapping data (variables) and code together as a single unit
- It is one of the four OOP concepts
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

# Review: Encapsulation example 1

```
public class Account {
   private int account_number;
   private int account_balance;

   public void showData() {
      //code to show data
   }

   public void deposit(int a) {
       if (a < 0){
          //show error
       } else {
          account_balance = account_balance + a;
       }
}
```

- Approach 1 and Approach 2 fail
- You never expose your data to an external party (which makes your  application secure)
- The entire code can be thought as capsule

# Review: Point class

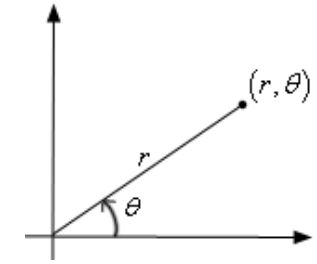
```
public class Point{
     private int x;
     private int y;

     public Point(){
          this(0, 0);
     }

     public Point(int x, int y){
          setLocation(x, y);
     }

     public double distanceFromOrigin(){
          return Math.sqrt(x * x + y * y);
     }

     public int getX(){
          return x;
     }
…
```

```
…
     public int getY(){
          return y;
     }

     public void setLocation(int x, int y){
          this.x = x;
          this.y = y;
     }

     public String toString(){
          return "(" + x + "," + y + ")";
     }

     public void translate (int dx, int dy){
          setLocation(x + dx, y + dy);
     }

…
}
```

# Benefits of encapsulation

- Abstraction between object and clients
- Protects object from unwanted access
- Example: Can't fraudulently change the points coordinates
  - `getX()`, `getY()` return just  a copy of the coordinates

# Benefits of encapsulation (cont.)




- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates ($r$, $\theta$) (radius, angle) with the same methods
  - Client calls to `getX`() and `getY`() do not need to change
  - We just change their internal implementation
- Can constrain objects' state (**invariants**)
  - Example: Only allow `Points` with non-negative coordinates

# Class invariants

- **Class invariant** is an assertion about an object's state that is true throughout the lifetime of the object
  - An invariant can be thought of as a postcondition on every constructor and mutator method of a class
  - e.g.: "No BankAccount object's balance can be negative”

- **Example**: Suppose we want to ensure that all `Point` objects' `x` and `y` coordinates are never negative
  - We must ensure that a client cannot construct a `Point` object with a negative `x` or `y` value
  - We must ensure that a client cannot move an existing `Point` object to a negative (x, y) location

# Pre/postconditions

- **Precondition**: Something that you assume to be true when your method is called
- **Postcondition**: Something you promise to be true when your method exits
  - Pre/postconditions are often documented as comments

```
// Sets this Point's location to be the given (x, y)
// Precondition: newX >= 0 && newY >= 0
// Postcondition: x >= 0 && y >= 0
public void setLocation(int newX, int newY) {
    x = newX;
    y = newY;
}
```

# Violated preconditions

- What if your precondition is not met?
- Sometimes the client passes an invalid value to your method

```
Point pt = new Point(5, 17);
Scanner console = new Scanner(System.in);
System.out.print("Type the coordinates: ");
int x = console.nextInt();  // what if the user types a negative number?
int y = console.nextInt();
pt.setLocation(x, y);
```

- How can we prevent the client from misusing our object?

# Dealing with violations

- One way to deal with this problem would be to return out of the method if negative values are encountered
  - However, it is not possible to do something similar in the constructor
- A more common solution is to have your object ***throw an exception***
- **Exception** is a Java object that represents an error
  - When a precondition of your method has been violated, you can generate ("`throw`") an exception in your code
  - This will cause the client program to halt

# Throwing exceptions example

- Throwing an exception, general syntax:

  **`throw new <exception type> ();`**

  or **`throw new <exception type> ("<message>");`**

  - The ***<message>*** will be shown on the console when the program crashes

```
// Sets this Point's location to be the given (x, y).
// Throws an exception if newX or newY is negative.
// Postcondition: x >= 0 && y >= 0
public void setLocation(int x, int y) throws
IllegalArgumentException{
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    }
    this.x = x;
    this.y = y;
}
```

# `Point` class and invariants

- Ensure that no `Point` is constructed with negative `x` or `y`:

```
public Point(int x, int y) throws
IllegalArgumentException{
     if (x < 0 || y < 0) {
          throw new IllegalArgumentException();
     }
     this.x = x;
     this.y = y;
}
```

- Ensure that no `Point` can be moved to a negative `x` or `y`:

```
public void translate(int dx, int dy) throws
IllegalArgumentException {
     if (x + dx < 0 || y + dy < 0) {
          throw new IllegalArgumentException();
     }
     x += dx;
     y += dy;
}
```

# Eliminating Redundancy

```
public class Point {
   private int x;
   private int y;
   public Point(int initialX, int initialY) {
        setLocation(initialX, initialY);
    }
   ....
   public void translate (int dx, int dy){
        setLocation(x + dx, y + dy);
   }
   public void setLocation throws IllegalArgumentException(int
   x, int y){
         if (x < 0 || y < 0) {
               throw new IllegalArgumentException()
         }
         this.x = x;
         this.y = y;
   }
}
```

Add the invariants only in one location

# Final `Point` class

```
public class Point {
   private int x;
   private int y;

   public Point() {
         this(0, 0);    // calls Point(int, int) constructor
   }

   public Point(int x, int y) {
         setLocation(x, y);
   }

  public double distanceFromOrigin() {
          return Math.sqrt(x * x + y * y);
   }
   public void translate(int dx, int dy) {
         setLocation(x + dx, y + dy);
   }
  …
}
```

# Final `Point` class (cont.)

```
public class Point {
   …
   public boolean equals(Object o) {
         if (o instanceof Point) {
             Point other = (Point) o;
             return x == other.x && y == other.y;
         } else {  // not a Point object
             return false;
         }
   }
   public int getX() {
         return x;
   }

   public int getY() {
         return y;
   }
   ...
}
```

# Final `Point` class (cont.)

```
public class Point {
   …

   public void setLocation(int x, int y) throws
   IllegalArgumentException {
         if (x < 0 || y < 0) {
             throw new IllegalArgumentException();
         }
         this.x = x;
         this.y = y;
   }

   public String toString() {
         return "(" + x + ", " + y + ")";
   }


}
```

# Inheritance

# The importance of code reuse

- **Software engineering** is the practice of designing, developing, documenting, testing and maintaining large computer programs
- Large-scale projects face many issues:
  - Getting many programmers to work together
  - Avoiding redundant code
  - Finding and fixing bugs
  - Maintaining, improving, and reusing existing code

- **Code reuse** is the practice of writing program code once and using it in many contexts

# Inheritance

- Inheritance is an important concept of OOP that **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

# Inheritance

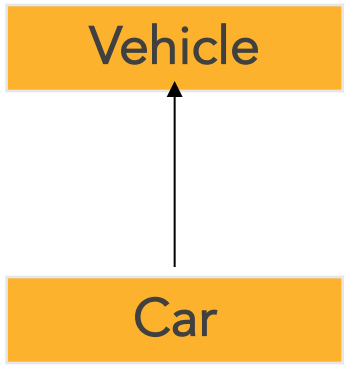
- The existing class is called the ***parent class***, or ***superclass***, or ***base class***
- The derived class is called the ***child class*** or ***subclass***
  - The child class inherits the methods and data defined for the parent class

# Inheritance

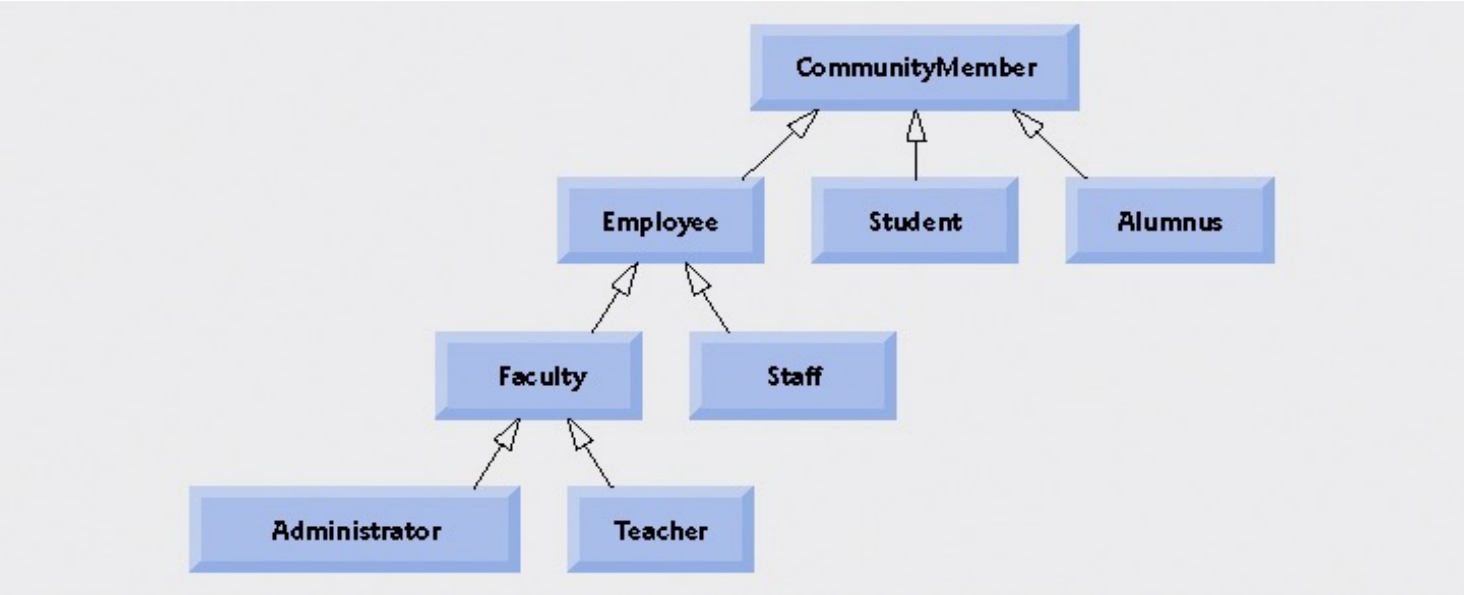
- *Software reuse* is at the heart of inheritance
- We are using existing software components to create new ones
  - We capitalize on all the effort that went into the design, implementation, and testing of the existing software
- The programmer can add new variables or methods to a subclass or can modify the inherited ones

# Inheritance

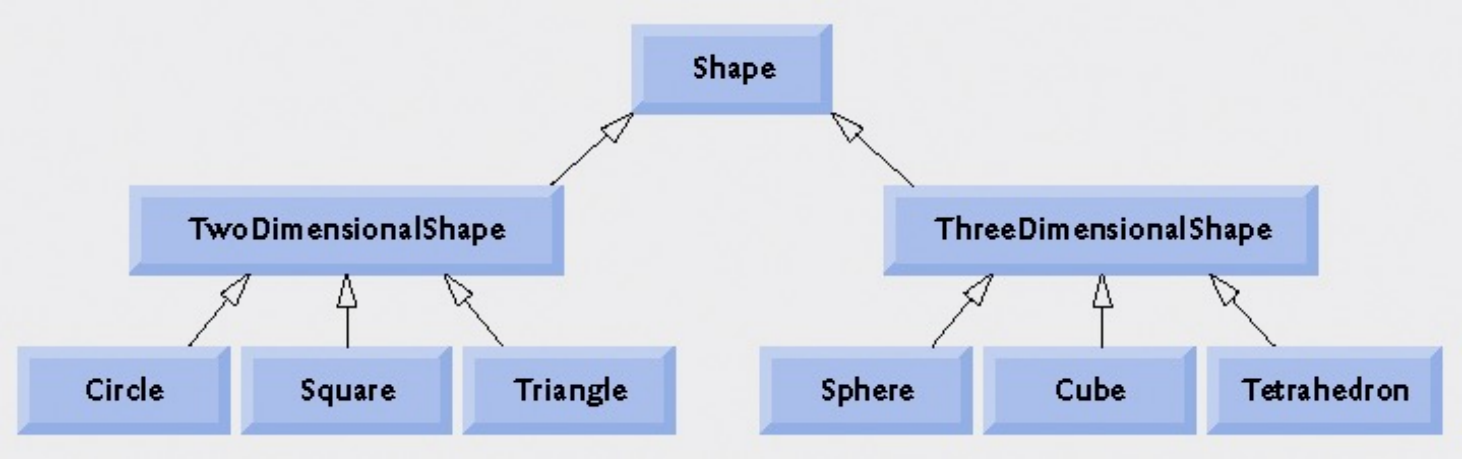
- Inheritance relationships often are shown graphically in a **UML class diagram**, with an open arrowhead pointing to the parent class



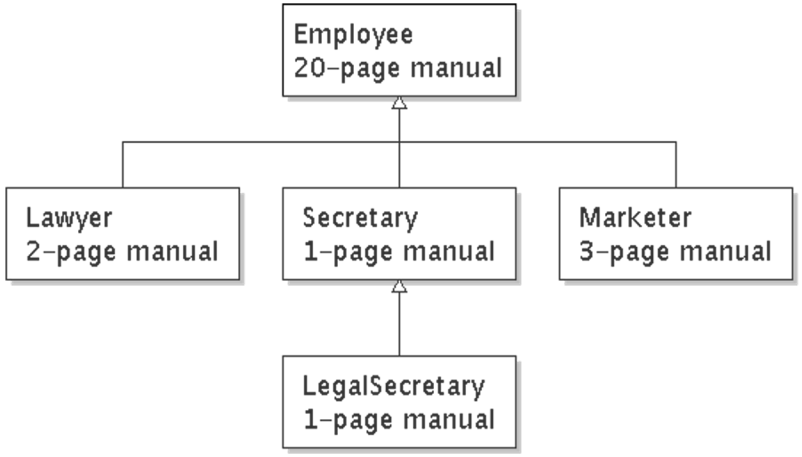
# Inheritance example



# Inheritance example

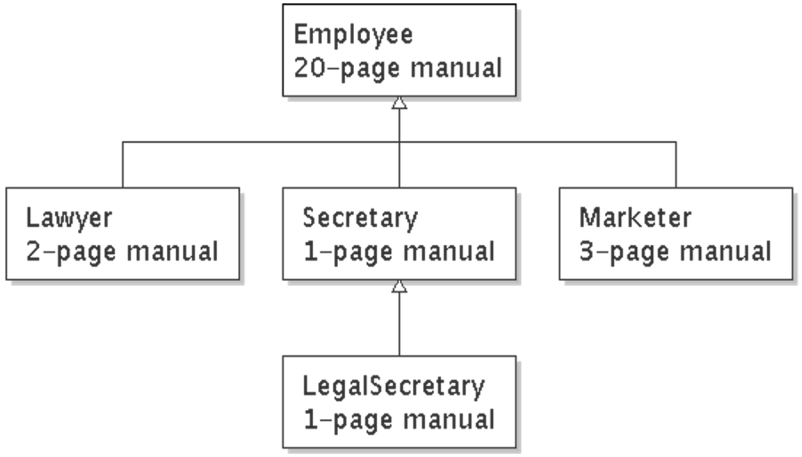


# Law firm employee hierarchy




- Common rules: hours, vacation, benefits, regulations ...
  - All employees attend a common orientation to learn general company rules
- Each subdivision also has specific rules
- We can have a 22-page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?

# Law firm employee hierarchy



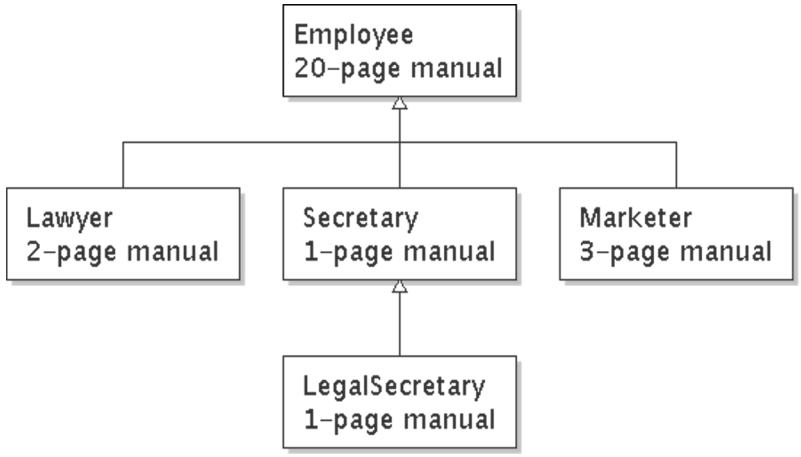

- Common rules: hours, vacation, benefits, regulations ...
  - All employees attend a common orientation to learn general company rules
  - Each employee receives a 20-page manual of common rules
- Each subdivision also has specific rules:
  - Employee receives a smaller (1-3 page) manual of these rules
  - Smaller manual adds some new rules and also changes some rules from the large manual

# Separating behavior

- Why not just have a 22-page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?
- Some advantages of the separate manuals:
  - **Maintenance**: Only one update if a common rule changes
  - **Locality**: Quick discovery of all rules specific to lawyers

- Some key ideas from this example:
  - General rules are useful (the 20-page manual)
  - Specific rules that may override general ones are also useful

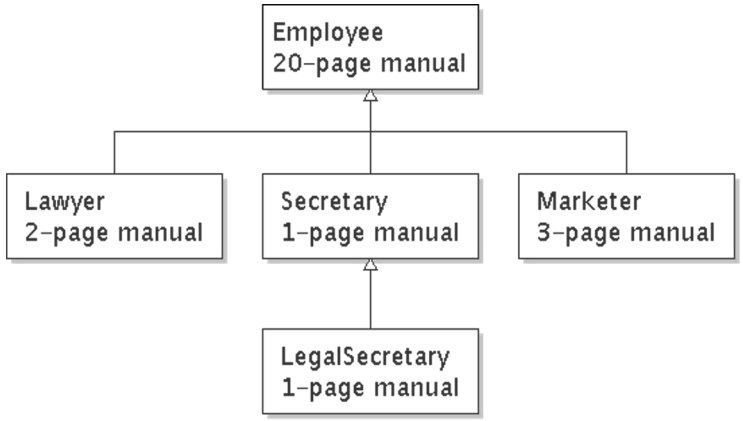
# Is-a relationships




- **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

# Employee regulations




- Employee regulations:
  - **Employee** works 40 hours / week
  - **Employee** makes $40,000 per year, except **Legal Secretary** who makes $5,000 extra per year ($45,000 total), and **Marketer** who makes $10,000 extra per year ($50,000 total)
  - **Employee** have 2 weeks of paid vacation leave per year, except **Lawyer** who get an extra week (a total of 3)
  - **Employee** should use a yellow form to apply for leave, except for **Lawyer** who uses a pink form

- Each type of employee has some unique behavior:
  - **Lawyer** knows how to sue
  - **Marketer** knows how to advertise
  - **Secretary** knows how to take dictation
  - **Legal Secretary** knows how to prepare legal documents

# Employee class

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
    }
}
```

# Secretary class

**Redundant**

Secretary.java

```
// A redundant 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);
    }
}
```

# Desire for code-sharing

- `takeDictation` is the only unique behavior in `Secretary`
- We'd like to be able to say:

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

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

This is the only added behaviour

# 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 (seen later)

# 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);
    }

}
```

- We only write the parts unique to each type of employee
  - `Secretary` inherits `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` methods from Employee
  - Secretary adds the `takeDictation` method

# Client program example

```
public class EmployeeMain {

   public static void main(String[] args) {
      System.out.println("Employee:");
      Employee employee1 = new Employee();
      System.out.print(employee1.getHours() + ", ");
      System.out.printf("$%.2f, ", employee1.getSalary());
      System.out.print(employee1.getVacationDays() + ", ");
      System.out.println(employee1.getVacationForm());

      System.out.print("Secretary: ");
      Secretary employee2 = new Secretary();
      System.out.print(employee2.getHours() + ", ");
      System.out.printf("$%.2f, ", employee2.getSalary());
      System.out.print(employee2.getVacationDays() + ", ");
      System.out.println(employee2.getVacationForm());
      employee2.takeDictation("CS12b example");
   }
}
```

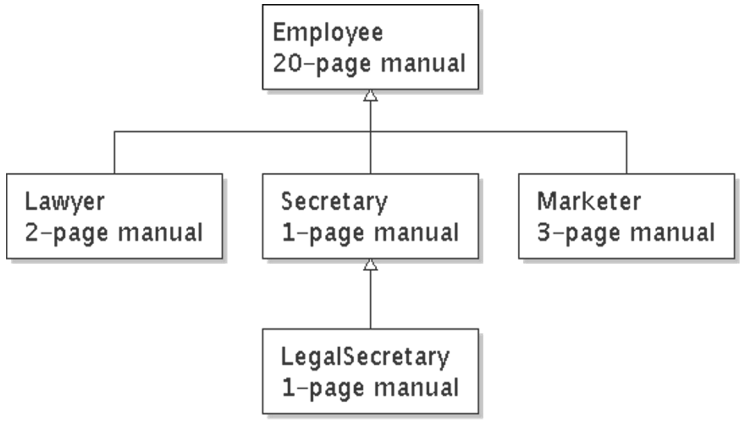
You call the methods of the Employee class (superclass)

You call the inherited methods of the Employee class (superclass)

The only method declared separately for Secretary

```
Employee: 40, $40000.00, 10, yellow
Secretary: 40, $40000.00, 10, yellow
Taking dictation of text: CS12b example
```

# Implementing `Lawyer`




- Lawyer regulations:
  - Gets an extra week of paid vacation (a total of 3)
  - Uses a pink form when applying for vacation leave
  - Has some unique behavior: they know how to sue

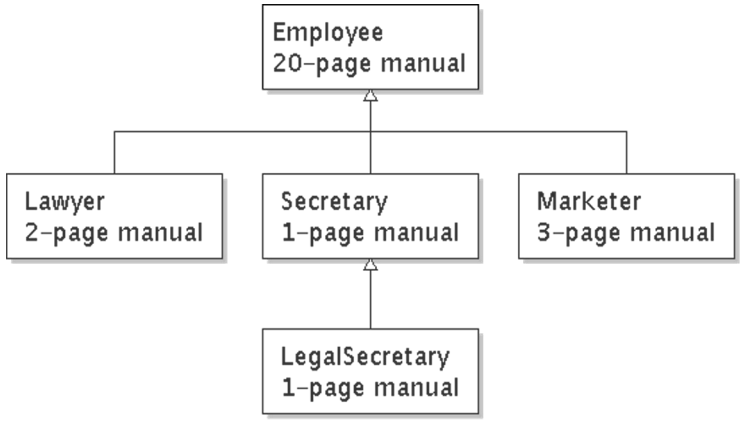
- We want lawyer to inherit most behavior from employee, but we want to replace parts with new behavior

# Overriding methods

- **Override**: To write a new version of a method in a subclass that replaces the superclass's version
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass

```
public class Lawyer extends Employee {
// overrides getVacationForm method in Employee class
        public String getVacationForm() {
                return "pink";
        }
        ...
}
```

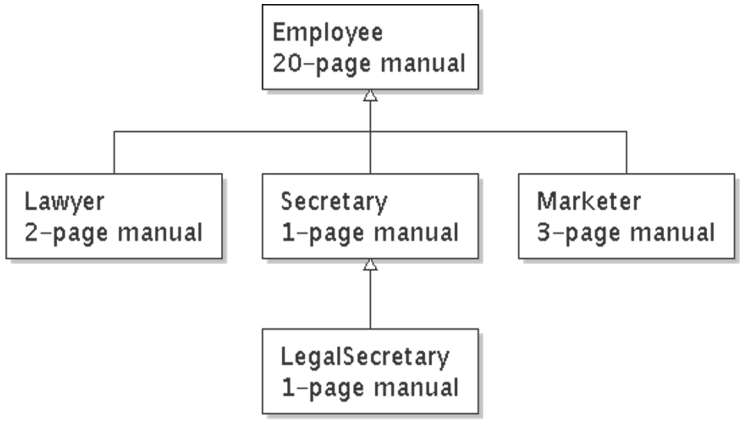
# Employee regulations




- Employee regulations:
  - **Employee** works 40 hours / week
  - **Employee** makes $40,000 per year, except **Legal Secretary** who makes $5,000 extra per year ($45,000 total), and **Marketer** who makes $10,000 extra per year ($50,000 total)
  - **Employee** have 2 weeks of paid vacation leave per year, except **Lawyer** who get an extra week (a total of 3)
  - **Employee** should use a yellow form to apply for leave, except for **Lawyer** who uses a pink form

- Each type of employee has some unique behavior:
  - **Lawyer** knows how to sue
  - **Marketer** knows how to advertise
  - **Secretary** knows how to take dictation
  - **Legal Secretary** knows how to prepare legal documents

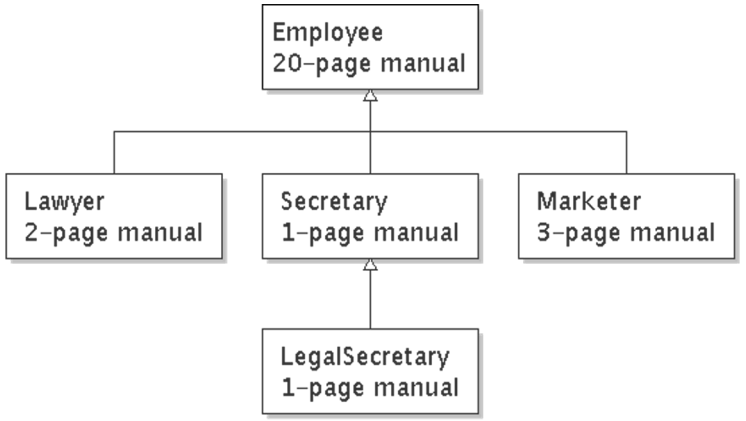
# Marketer class




- Marketer makes $10,000 extra ($50,000 total) and knows how to advertise

```
// 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;      // $50,000 / year
    }
}
```

# LegalSecretary class




- Legal secretary makes $5,000 extra per year ($45,000 total) and knows how to prepare legal documents

```
// 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
    }
}
```

# Client program

```
public class EmployeeMain2 {

  public static void main(String[] args) {
     System.out.println("Lawyer:");
     Lawyer employee3 = new Lawyer();
     System.out.print(employee3.getHours() + ", ");
     System.out.printf("$%.2f, ", employee3.getSalary());
     System.out.print(employee3.getVacationDays() + ", ");
     System.out.println(employee3.getVacationForm());
     employee3.sue();

     System.out.print("Legal Secretary: ");
     LegalSecretary employee4 = new LegalSecretary();
     System.out.print(employee4.getHours() + ", ");
     System.out.printf("$%.2f, ", employee4.getSalary());
     System.out.print(employee4.getVacationDays() + ", ");
     System.out.println(employee4.getVacationForm());
     employee4.takeDictation("CS12b example");
     employee4.fileLegalBriefs();
  }
}
```