

# CS 142

## Lecture 10: Inheritance and Polymorphism



Thanks to Marty Stepp and Stuart Reges for parts of these slides

# 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.
  - Exercise: Make necessary modifications to the `Employee` class.

# Modified 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.
- The long explanation: (next slide)

# The detailed explanation

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

```
public Lawyer() {  
    super();           // calls Employee() constructor  
}
```

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

# Calling superclass constructor

```
super (parameters) ;
```

- Example:

```
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.
- Exercise: Make a similar modification to the `Marketer` 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;
    }
}
```

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

# Modified 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);  
    }  
}
```

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

# 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 subclassing can't break encapsulation.
  - How can we get around this limitation?

# Improved Employee code

Add an accessor for any field needed by the subclass.

```
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();
    }
    ...
}
```

# 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.

# Improved Employee code

- 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();
    }

    // vacation days given for each year in the company
    public int getSeniorityBonus() {
        return 2 * years;
    }
    ...
}
```

- How does this help us improve the Secretary?

# Improved Secretary code

- 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);
    }
}
```

# Polymorphism

- **polymorphism:** Ability for the same code to be used with different types of objects and behave differently with each.
  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.

# Coding with polymorphism

- A variable of type  $T$  can hold an object of any subclass of  $T$ .

```
Employee ed = new Lawyer();
```

- You can call any methods from the `Employee` class on `ed`.

- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary());           // 50000.0  
System.out.println(ed.getVacationForm());    // pink
```

# Polymorphism examples

- You can use the object's extra functionality by casting.

```
Employee ed = new Lawyer();  
ed.getVacationDays(); // ok  
ed.sue(); // compiler error  
(Lawyer ed).sue(); // ok
```

- You can't cast an object into something that it is not.

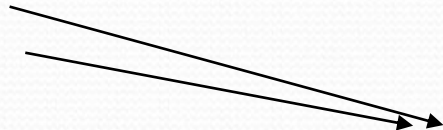
```
Object otto = new Secretary();  
System.out.println(otto.toString()); // ok  
otto.getVacationDays(); // compiler error  
(Employee otto).getVacationDays(); // ok  
(Lawyer otto).sue(); // runtime error
```

# Polymorphism and parameters

- You can pass any subtype of a parameter's type.

```
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer();
        Secretary steve = new Secretary();
        printInfo(lisa);
        printInfo(steve);
    }

    public static void printInfo(Employee empl) {
        System.out.println("salary: " + empl.getSalary());
        System.out.println("v.days: " + empl.getVacationDays());
        System.out.println("v.form: " + empl.getVacationForm());
        System.out.println();
    }
}
```



## OUTPUT:

|                 |                 |
|-----------------|-----------------|
| salary: 50000.0 | salary: 50000.0 |
| v.days: 15      | v.days: 10      |
| v.form: pink    | v.form: yellow  |

# Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] e = { new Lawyer(), new Secretary(),
                       new Marketer(), new LegalSecretary() };

        for (int i = 0; i < e.length; i++) {
            System.out.println("salary: " + e[i].getSalary());
            System.out.println("v.days: " + e[i].getVacationDays());
            System.out.println();
        }
    }
}
```

## Output:

```
salary: 50000.0
v.days: 15

salary: 50000.0
v.days: 10

salary: 60000.0
v.days: 10

salary: 55000.0
v.days: 10
```

# "Polymorphism mystery"

- Figure out the output from all methods of these classes:

```
public class Snow {  
    public void method2() {  
        System.out.println("Snow 2");  
    }  
  
    public void method3() {  
        System.out.println("Snow 3");  
    }  
}
```

```
public class Rain extends Snow {  
    public void method1() {  
        System.out.println("Rain 1");  
    }  
  
    public void method2() {  
        System.out.println("Rain 2");  
    }  
}
```

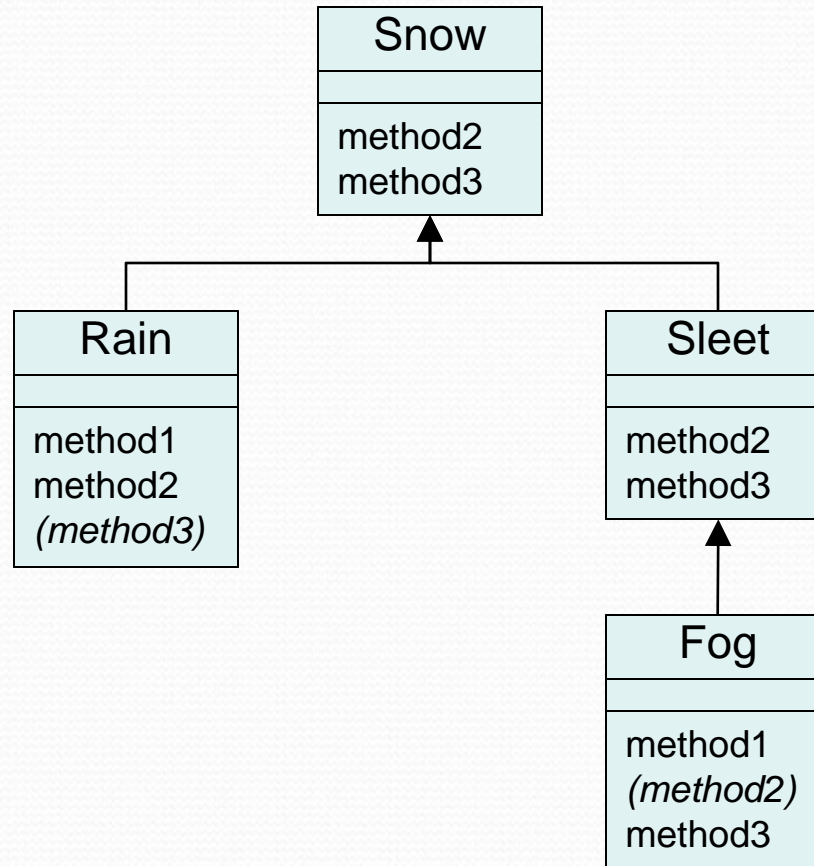
# "Polymorphism mystery"

```
public class Sleet extends Snow {  
    public void method2() {  
        System.out.println("Sleet 2");  
        super.method2();  
        method3();  
    }  
    public void method3() {  
        System.out.println("Sleet 3");  
    }  
}
```

```
public class Fog extends Sleet {  
    public void method1() {  
        System.out.println("Fog 1");  
    }  
    public void method3() {  
        System.out.println("Fog 3");  
    }  
}
```

# Technique 1: diagram

- Diagram the classes from top (superclass) to bottom.



# Technique 2: table

| <b>method</b> | <b>Snow</b> | <b>Rain</b>   | <b>Sleet</b>                           | <b>Fog</b>  |
|---------------|-------------|---------------|--|---|
| method1       |             | Rain 1        |  | Fog 1   |
| method2       | Snow 2      | Rain 2        | Sleet 2<br>Snow 2<br><b>method3 ()</b> | <i>Sleet 2</i><br><i>Snow 2</i><br><b><i>method3 ()</i></b> |
| method3       | Snow 3      | <i>Snow 3</i> | Sleet 3                                | Fog 3   |

*Italic* - inherited behavior

**Bold** - dynamic method call

# Mystery problem, no cast

```
Snow var3 = new Rain ();  
var3.method2 ();           // What's the output?
```

- If the problem does *not* have any casting, then:
  1. Look at the variable's type.  
If that type does not have the method: ERROR.
  2. Execute the method, behaving like the object's type.  
(The variable type no longer matters in this step.)

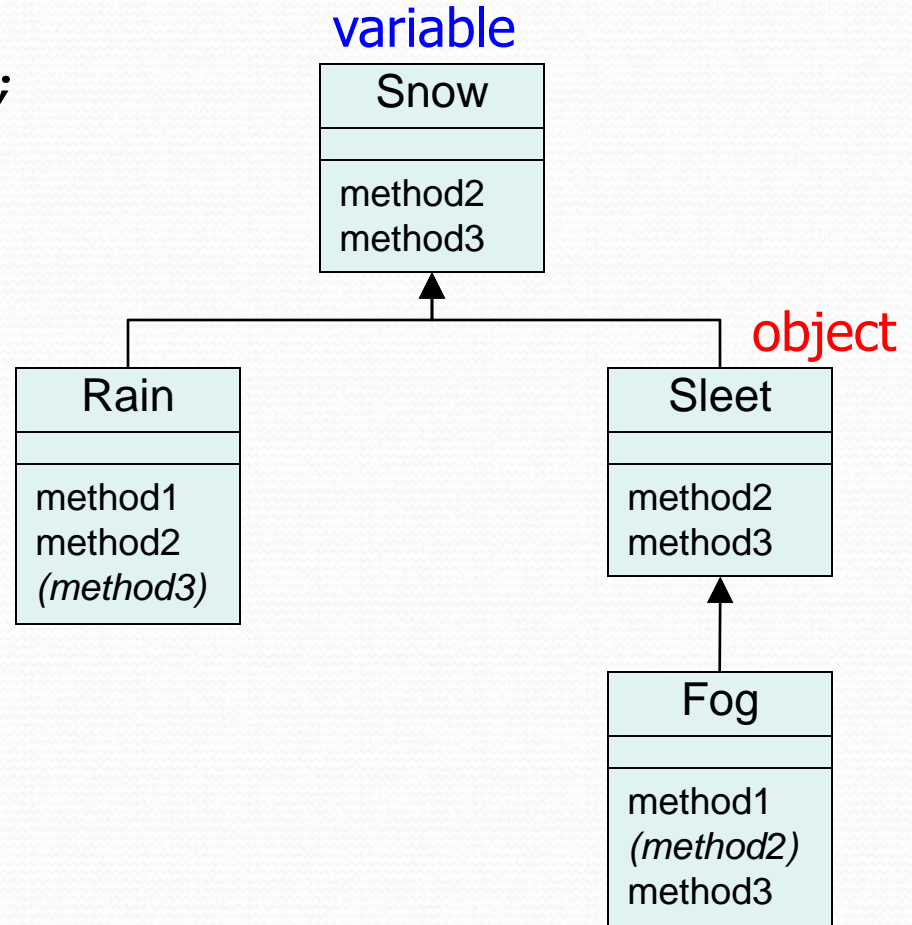
# Example 1

- What is the output of the following call?

```
Snow var1 = new Sleet ();  
var1.method2 ();
```

- Answer:

```
Sleet 2  
Snow 2  
Sleet 3
```



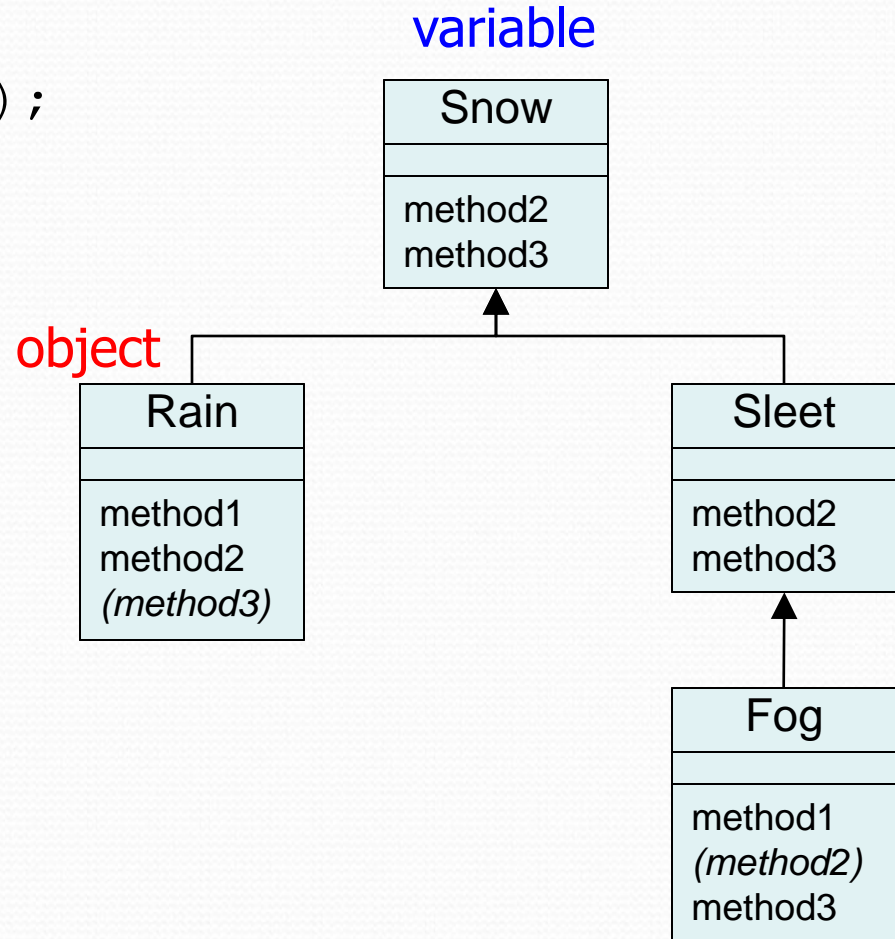
# Example 2

- What is the output of the following call?

```
Snow var2 = new Rain ();  
var2.method1 ();
```

- Answer:

ERROR  
(because `Snow` does not  
have a `method1`)



# Mystery problem with cast

```
Snow var2 = new Rain();  
((Sleet) var2).method2(); // What's the output?
```

- If the problem *does* have a type cast, then:
  1. Look at the cast type.  
If that type does not have the method: ERROR.
  2. Make sure the object's type is the cast type or is a subclass of the cast type. If not: ERROR. (No sideways casts!)
  3. Execute the method, behaving like the object's type.  
(The variable / cast types no longer matter in this step.)

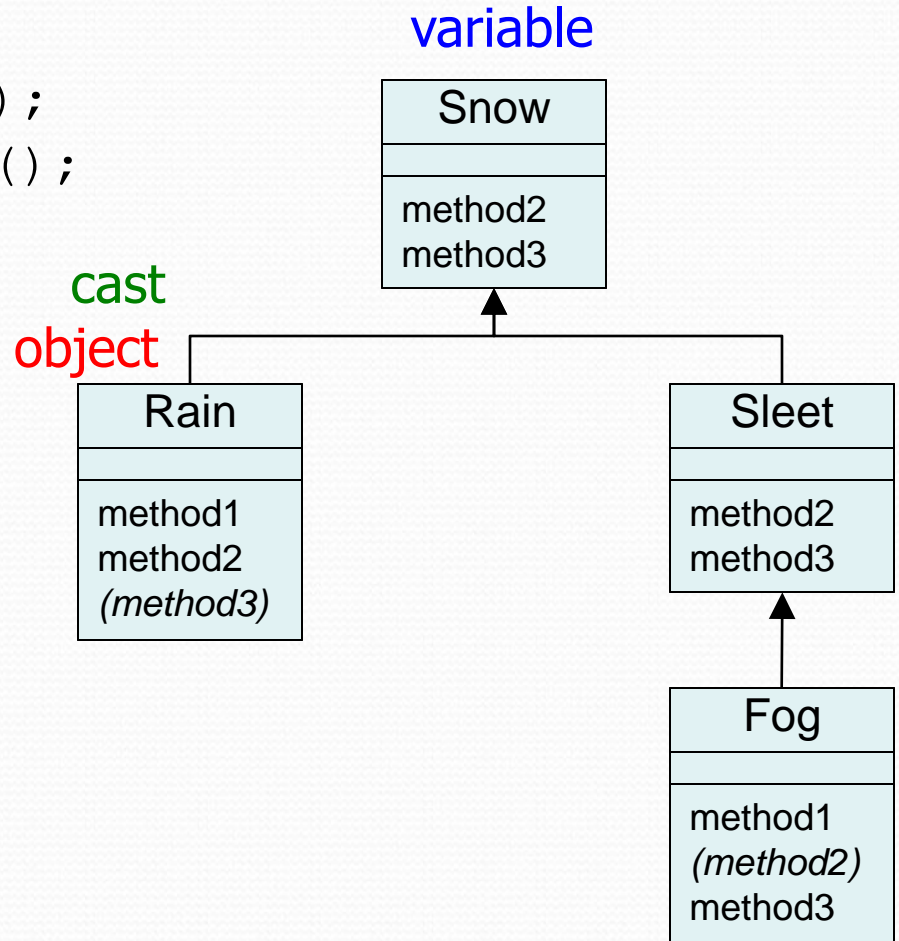
# Example 3

- What is the output of the following call?

```
Snow var2 = new Rain();  
(Rain) var2.method1();
```

- Answer:

```
Rain 1
```



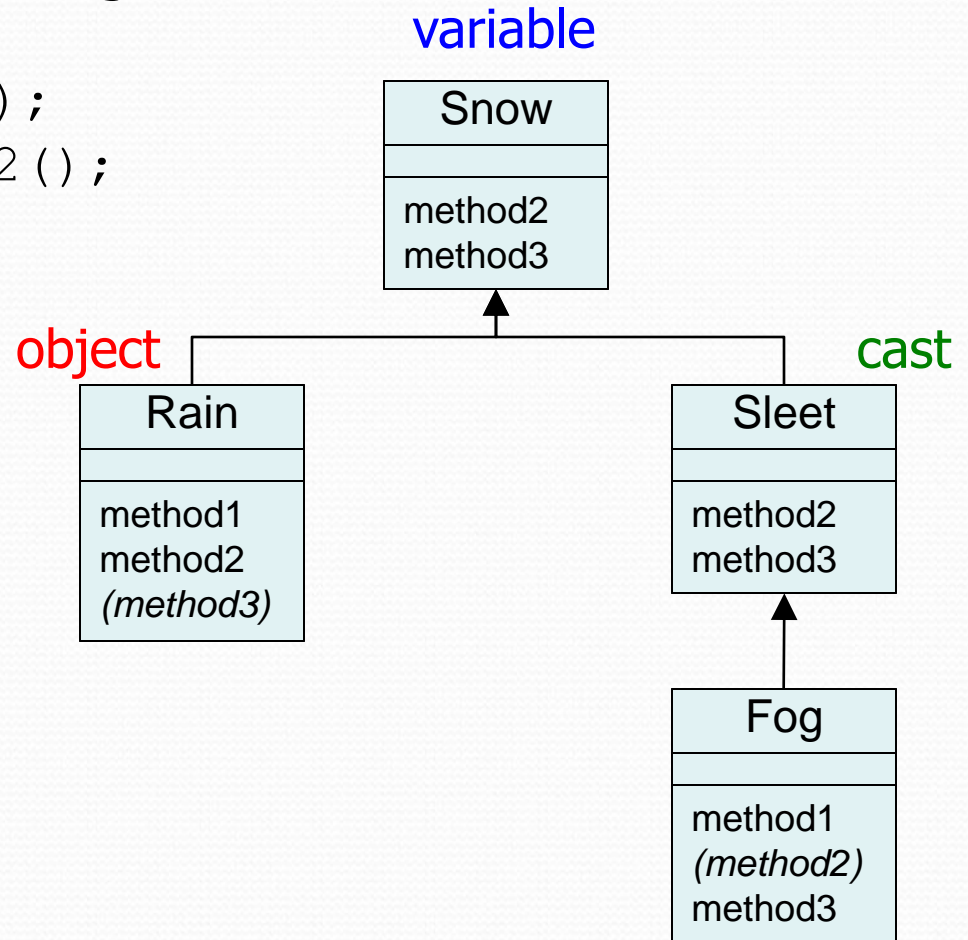
# Example 4

- What is the output of the following call?

```
Snow var2 = new Rain();  
(Sleet) var2.method2();
```

- Answer:

ERROR  
(because the object's  
type, Rain, cannot  
be cast into Sleet)



# Casting references

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();  
int hours = ed.getHours(); // ok; this is in Employee  
ed.sue(); // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue`.
- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;  
theRealEd.sue(); // ok  
  
( (Lawyer) ed ).sue(); // shorter version
```

# More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();  
((Secretary) eric).takeDictation("hi"); // ok  
((LegalSecretary) eric).fileLegalBriefs(); // exception  
  
// (Secretary object doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();  
((Secretary) linda).takeDictation("hi"); // error
```

- Casting doesn't actually change the object's behavior.  
It just gets the code to compile/run.

```
((Employee) linda).getVacationForm() // pink (Lawyer's)
```