Lecture 17: Polymorphism (Part I)

Building Java Programs: A Back to Basics Approach

by Stuart Reges and Marty Stepp

Polymorphism

Polymorphism

- **polymorphism**:

1) Ability for the same code to be used with different types of objects and behave differently with each.(Part I Lecture)
   - `System.out.println` can print any type of object.
     - Each one displays in its own way on the console.

1) Ability for a method to take on many forms. (Part II Lecture)

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
Student s1=new GradStudent();
s1.computeGrade(); //calls GradStudent’s version.
```

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 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();
    }
}
```

Note: This code will remain the same regardless of how many subclasses of Employee we add later in our code.

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
```

Exercise 1

- Suppose that the following four classes have been declared:

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

Exercise 1

```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }
    public String toString() {
        return "baz";
    }
}
public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

- What would be the output of the following client code?

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Exercise 1's Answers

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

- Output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

Exercise 2

- The order of the classes is jumbled up.
- The methods sometimes call other methods (tricky!).

```
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}

public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }

    public void b() {
        System.out.print("Ham b   ");
    }

    public String toString() {
        return "Ham";
    }
}
```

Exercise 2

```
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b   ");
    }
}

public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a   ");
        super.a();
    }

    public String toString() {
        return "Yam";
    }
}
```

- What would be the output of the following client code?

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();      // to end the line of output
    food[i].b();
    System.out.println();      // to end the line of output
    System.out.println();
```

Polymorphism at work

- `Lamb` inherits `Ham`'s `a`. `a` calls `b`. But `Lamb` overrides `b`...

```
public class Ham {
    public void a() {
        System.out.print("Ham a    ");
        b();
    }
    public void b() {
        System.out.print("Ham b    ");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b    ");
    }
}
```

- `Lamb`'s output from `a`:

```
Ham a    Lamb b
```

Exercise 2's Answers

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

- Output:

```
Ham
Ham a   Lamb b
Lamb b

Ham
Ham a   Ham b
Ham b

Yam
Yam a   Ham a   Spam b
Spam b

Yam
Yam a   Ham a   Lamb b
Lamb b
```

Casting references

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

```
Employee ed = new Lawyer();
int hours = ed.getHours();  // ok; it's 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,two sets of()
```

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();   // Class cast
                                            //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)
```

Exercise 3

- Assume that the following classes have been declared:

```
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");
    }
}
```

Exercise 3

```
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");
    }
}
```

Exercise 3

What happens when the following examples are executed?

- Example 1:

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

- Example 2:

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

- Example 3:

```
Snow var3 = new Rain();
((Sleet) var3).method3();
```

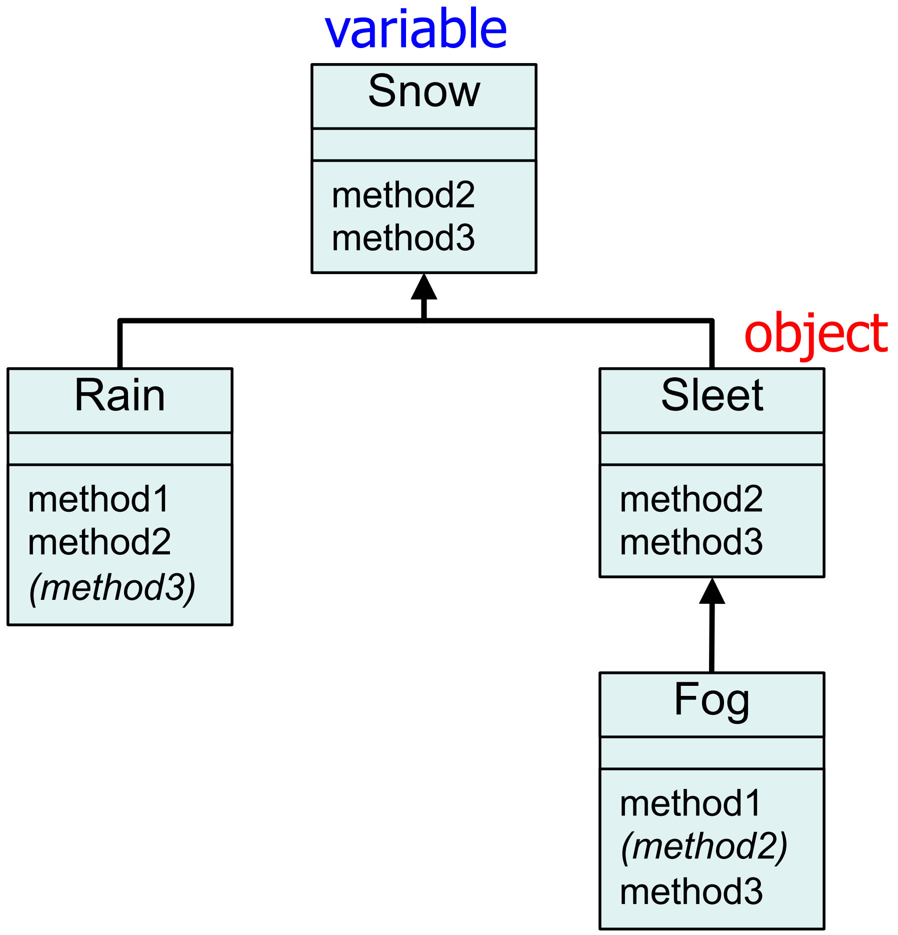
Exercise 3 Answers

- Example:

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

- Output:

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



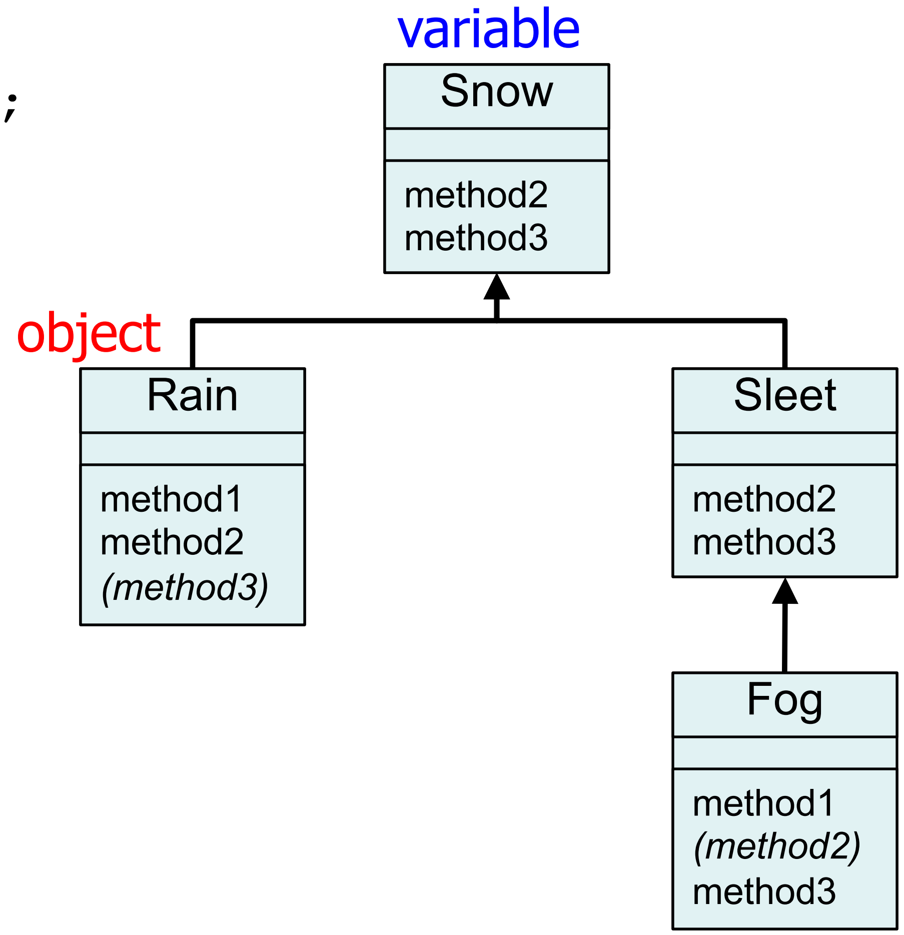
Exercise 3 Answers

- Example:

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

- Output:

None!
There is an error,
because `Snow` does not
have a `method1`.



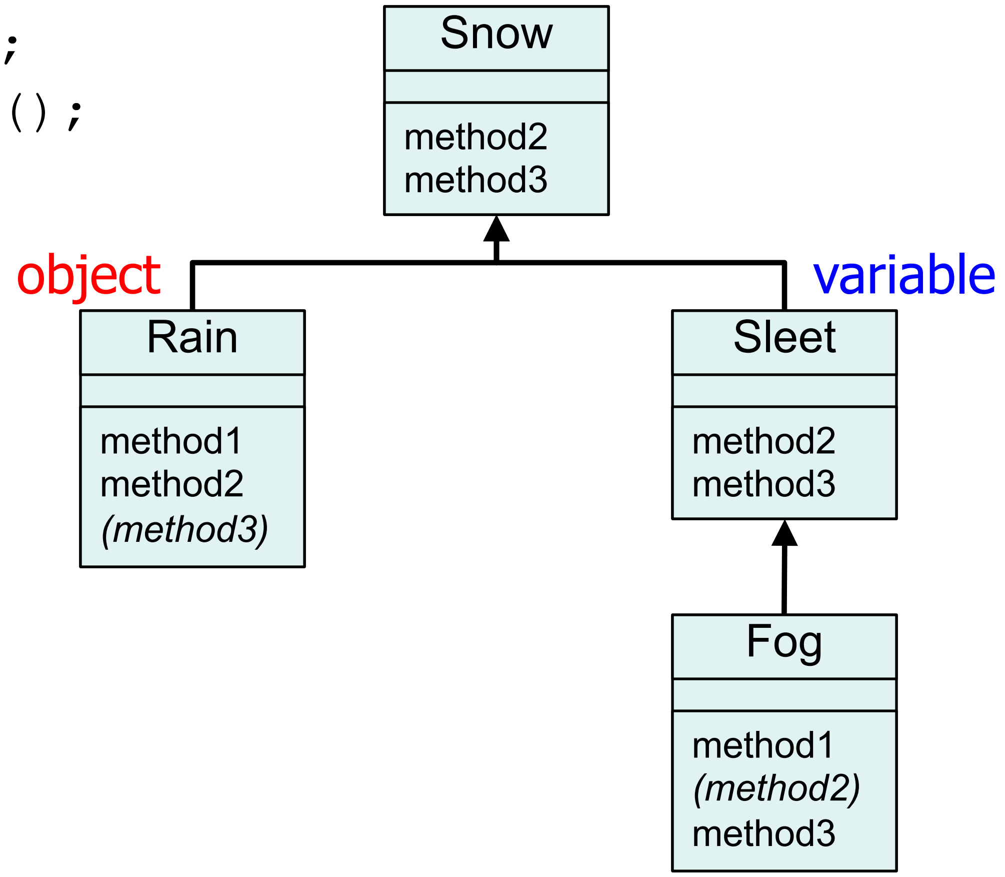
Exercise 3 Answers

- Example:

```
Snow var3 = new Rain();
((Sleet) var3).method2();
```

- Output:

  None!
  There is an error because a `Rain` is not a `Sleet`.



Review Example

```
Employee one=new Secretary();
//upcast, always ok
one.getSalary();
//calls Secretary's version
one.takeDictation();
//error, even though one holds a
//Secretary object, one is an Employee reference
//and can only call Employee's methods.

//here's how to fix the above error.
((Secretary) one).takeDictation();//cast then call
//can't cast too far down the tree
((LegalSecretary) one).fileLegalBriefs(); //error
```

Review Example 2

```
LegalSecretary two=new LegalSecretary();

//upcast doesn’t change behavior.
((Secretary) two).getSalary();
//still LegalSecretary’s
//version

((Employee) two).getSalary();
//still LegalSecretary’s version
//can’t cast sideways
((Lawyer) two).sue(); //error
```

Homework

Redo the 3 exercises in this lecture and check your answers. Then complete the Polymorphism Worksheet.