CSE 143
Lecture 22

The `Object` class; Polymorphism

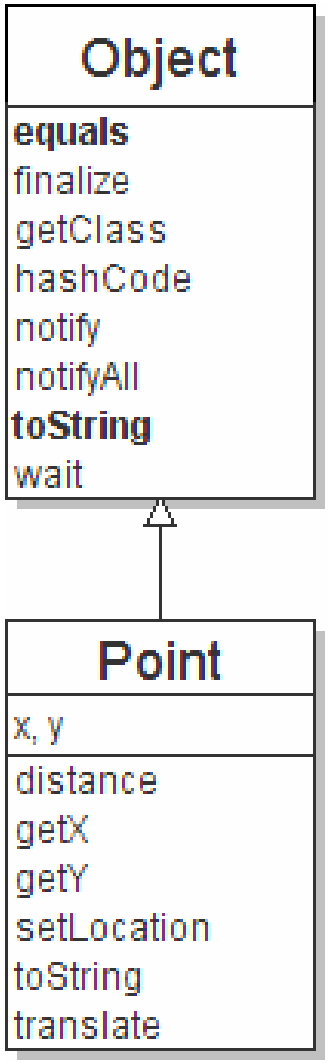
read 9.2 - 9.3

slides created by Marty Stepp and Ethan Apter
http://www.cs.washington.edu/143/

Class `Object`

- All types of objects have a superclass named `Object`.
  - Every class implicitly extends `Object`

- The `Object` class defines several methods:
  - `public String toString()`
    Returns a text representation of the object, often so that it can be printed.
  - `public boolean equals(Object other)`
    Compare the object to any other for equality. Returns `true` if the objects have equal state.



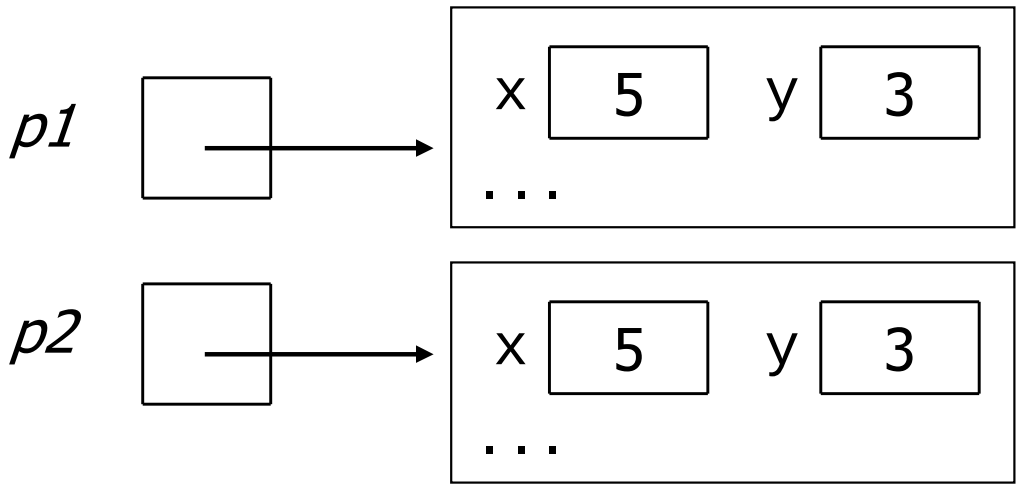
Recall: comparing objects

- The == operator does not work well with objects.

  == compares references to objects, not their state.

  It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1 == p2) {   // false
    System.out.println("equal");
}
```



The `equals` method

compares the state of objects

- The default `equals` behavior acts just like the == operator.

```
if (p1.equals(p2)) {   // false
    System.out.println("equal");
}
```

- We can change this behavior by writing an `equals` method.
  - The method should compare the state of the two objects and return `true` when the objects have the same state.

Flawed equals method

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

- It should be legal to compare a `Point` to any object (not just other `Point` objects):

```
// this should be allowed
Point p = new Point(7, 2);
if (p.equals("hello")) {   // false
    ...
```

`equals` and `Object` class

```
public boolean equals(Object name) {
    statement(s) that return a boolean value;
}
```

- The parameter to `equals` must be of type `Object` in order to override the default version of `equals`.
- `Object` is a general type that can match any object.
- Having an `Object` parameter means *any* object can be passed.

Another flawed version

```
public boolean equals(Object o) {
    return (x == o.x && y == o.y);
}
```

- Does not compile:

```
Point.java:36: cannot find symbol
symbol  : variable x
location: class java.lang.Object
return (x == o.x && y == o.y);
                 ^
```

  - Compiler: "o could be any object. Not every object has an x field."

Type-casting objects

- Solution: *Type-cast* the object parameter to a `Point`.

```
// almost correct version
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Casting objects is different than casting primitives.
  - We're casting an `Object` reference into a `Point` reference.
  - We're promising the compiler that `o` refers to a `Point` object.

Comparing different types

- When we compare `Point` objects to other types,

```
Point p = new Point(7, 2);
if (p.equals("hello")) {   // should be false
    ...
}
```

  - The code crashes:

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at Point.equals(Point.java:25)
        at PointMain.main(PointMain.java:25)
```

  - The culprit is the line with the type-cast:

```
public boolean equals(Object o) {
    Point other = (Point) o;
```

The `instanceof` keyword

asks whether a variable refers to an object of a given type

variable `instanceof` **type**

- The above is an expression with a `boolean` result.

```
String s = "hello";
Point p = new Point();

if (s instanceof Point) {
    ...
}
```

expression	result
`s instanceof String`	`true`
`s instanceof Object`	`true`
`s instanceof Point`	`false`
`p instanceof Point`	`true`
`p instanceof Object`	`true`
`p instanceof String`	`false`
`null instanceof String`	`false`

Final `equals` method

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        return false;  // not a Point; cannot be equal
    }
}
```

Polymorphism

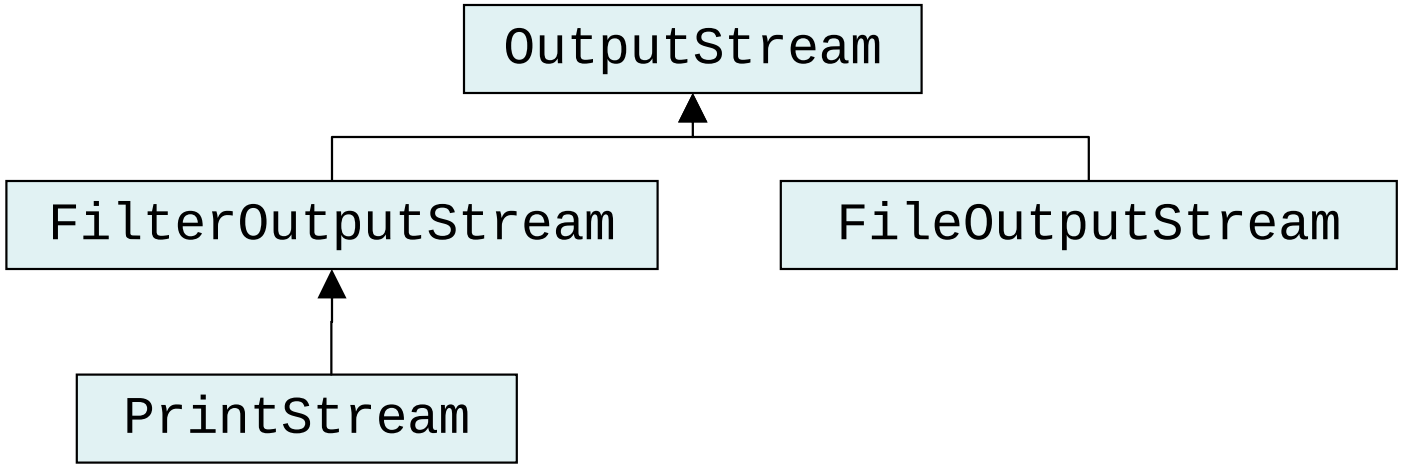
- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
  - A `Scanner` can read data from any kind of `InputStream`.
  - Every kind of `OutputStream` can write data, though they might write this to different kinds of sources.

Coding with polymorphism

- We can use polymorphism with classes like `OutputStream`.
  - Recall methods common to all `OutputStreams`:

Method	Description
`write``(int b)`	writes a byte
`close``()`	stops writing (also flushes)
`flush()`	forces any writes in buffers to be written

  - Recall part of the inheritance hierarchy for `OutputStream`:



Coding with polymorphism

- A variable of type T can refer to an object of *any subclass* of T.

```
OutputStream out  = new PrintStream(new File("foo.txt"));
OutputStream out2 = new FileOutputStream("foo.txt");
```

  - You can call any methods from `OutputStream` on `out`.
  - You can *not* call methods specific to `PrintStream` (`println`).
    - But how *would* we call those methods on `out` if we wanted to?

- When a method is called on `out`, it behaves as a `PrintStream`.

```
out.write(0);  // writes a 0 byte to foo.txt
out.close();   // closes the stream to foo.txt
```

Coding with polymorphism

- Some more polymorphism examples with `OutputStream`:

```
OutputStream out = new PrintStream(new File("foo.txt"));
out.write(0);                                  // ok
out.println("hello");                          // compiler error
((PrintStream) out).println("hello");          // ok
out.close();                                   // ok

OutputStream out2 = new FileOutputStream("foo.txt");
out2.println("hello");                         // compiler error
((PrintStream) out2).println("hello");         // runtime error
```

Inheritance mystery

- 4-5 classes with inheritance relationships are shown.

- A client program calls methods on objects of each class.
  - Some questions involve type-casting.
  - Some lines of code are illegal and produce errors.

- You must read the code and determine its output or errors.
  - For output, you must be precise
  - For errors, you need only say that an error occurred (not identify what kind of error occurred)

- **We always place such a question on our final exams!**

Inheritance mystery

- Steps to solving inheritance mystery:

1. Look at the variable type (if there is a cast, look at the casted variable type). If the variable type does not have the requested method the compiler will report an error.

2. If there was a cast, make sure the casted variable type is compatible with the object type (i.e. ensure the object type is a subclass of the variable type). If they are not compatible, a runtime error (`ClassCastException`) will occur.

3. Execute the method in question, behaving like the object type (the variable type and casted variable type no longer matter at all)

Exercise

- 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

```
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

What happens when the following examples are executed?

- Example 1:

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

- Example 2:

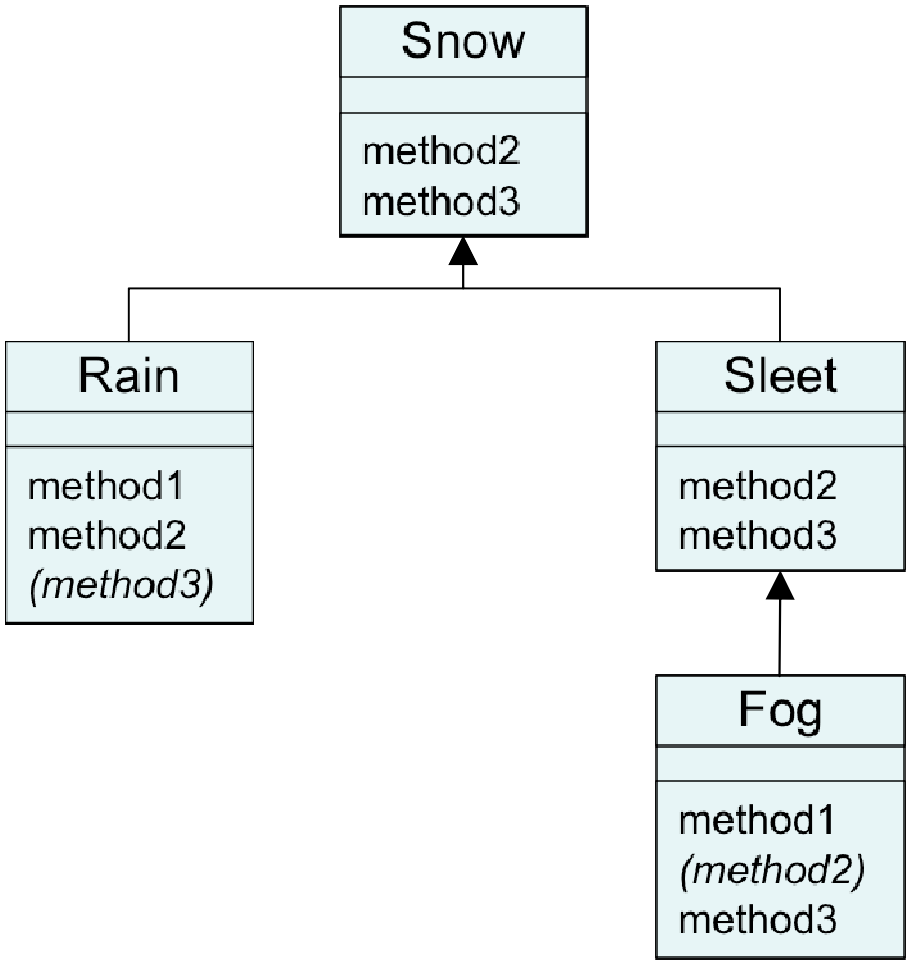
```
Snow var2 = new Rain();
var2.method1();
```

- Example 3:

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

Technique 1: diagram

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



Technique 2: table

method	Snow	Rain	Sleet	Fog
`method1`		`Rain 1`		`Fog 1`

`method2`	`Snow 2`	`Rain 2`	`Sleet 2`
			`Snow 2`
			`method3()`
				`Sleet 2`
				`Snow 2`
				`method3()`

`method3`	`Snow 3`	*`Snow 3`*	`Sleet 3`	`Fog 3`

Italics - inherited behavior
Bold - dynamic method call

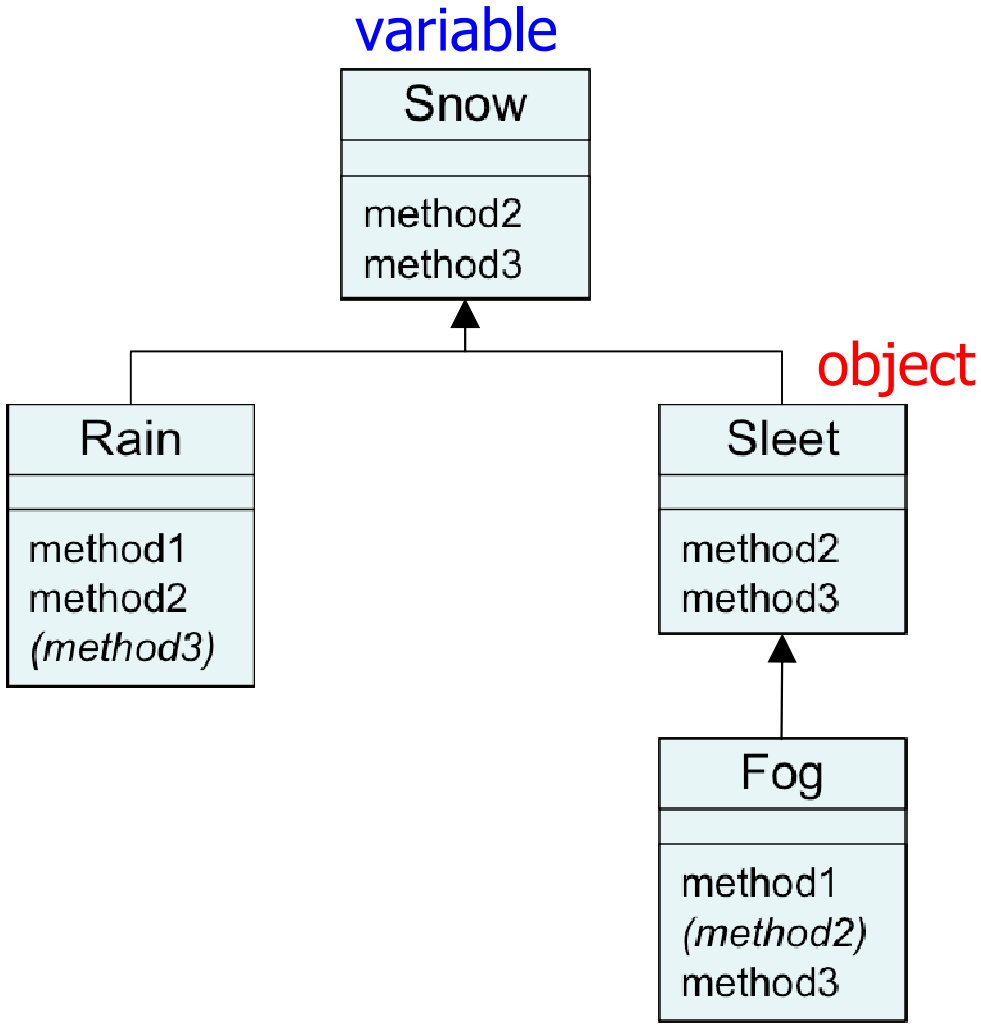
Example 1

- Example:

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

- Output:

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



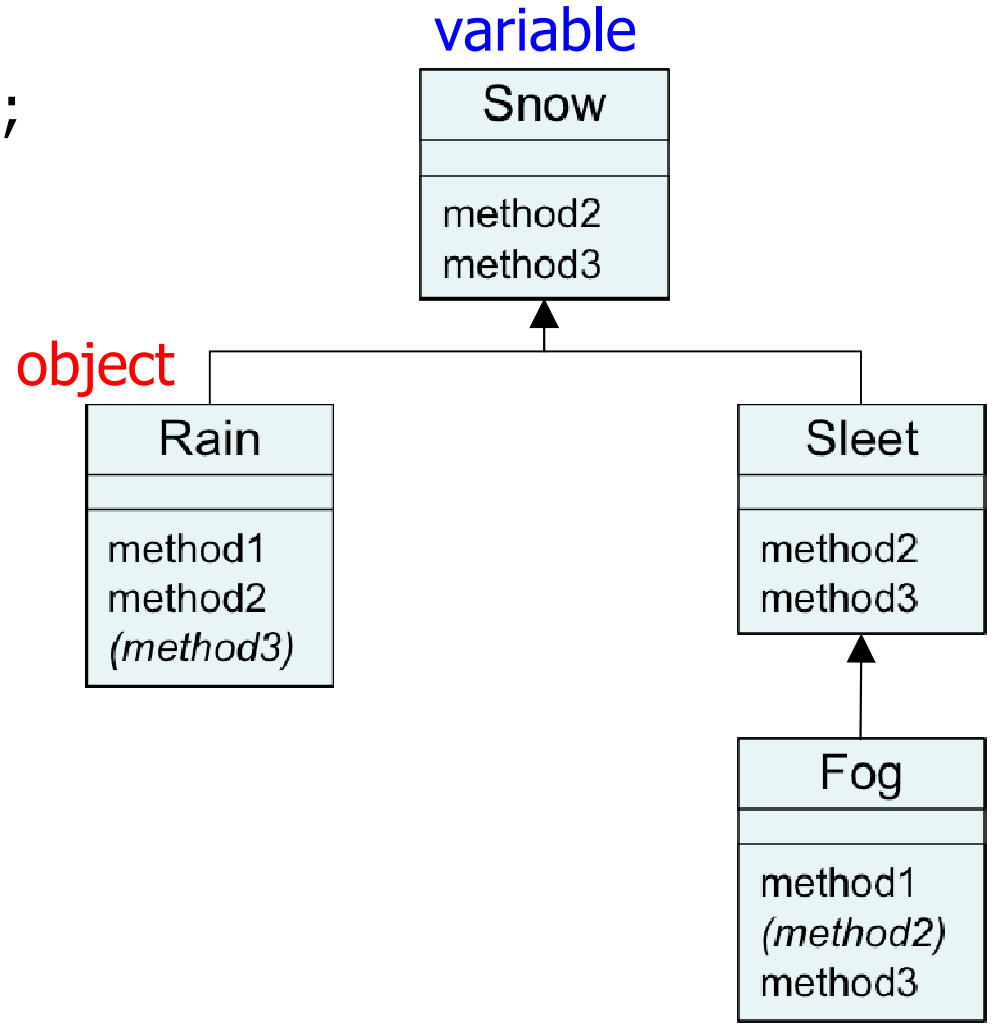
Example 2

- Example:

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

- Output:

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



Example 3

- Example:

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

- Output:

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