CSE 143
Lecture 23

Polymorphism; the `Object` class

read 9.2 - 9.3

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

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

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

```
Employee ed = new Lawyer();
Object otto = new Secretary();
```

  - You can call any methods from `Employee` on `ed`.
  - You can *not* call any methods specific to `Lawyer` (e.g. `sue`).

- 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/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("days = " + empl.getVacationDays());
        System.out.println("form = " + empl.getVacationForm());
        System.out.println();
    }
}
```

OUTPUT:

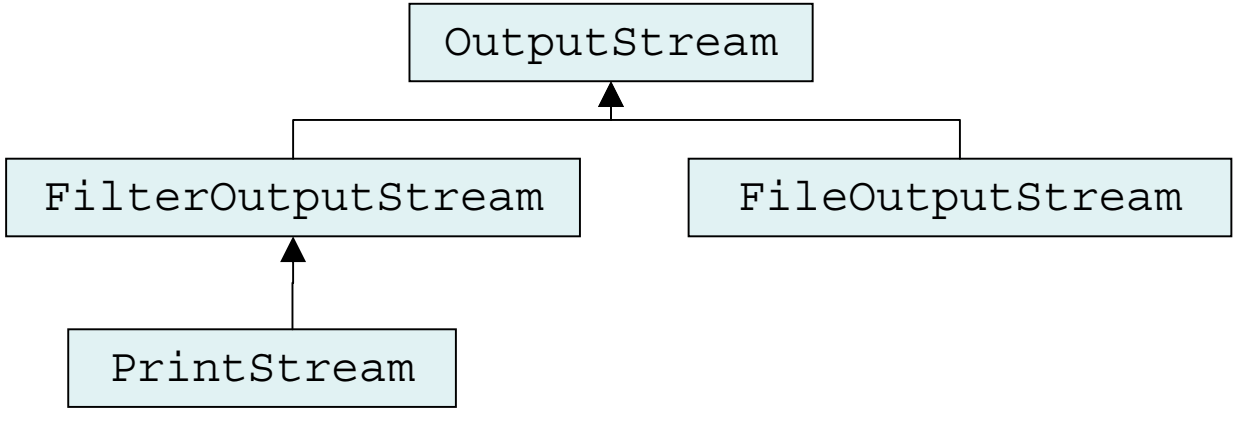
```
salary = 50000.0          salary = 50000.0
vacation days = 21        vacation days = 10
vacation form = pink      vacation form = yellow
```

Coding with polymorphism

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

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`:



Streams and 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 `out` runs a method, it behaves as a `PrintStream`.

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

Polymorphism examples

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

```
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
```

- You can't cast an object into something that it is not. Such code might compile, but it will crash at runtime.

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

Polymorphism 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!**

Polymorphism mystery

- Steps to solving polymorphism mystery problems:

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.)

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:

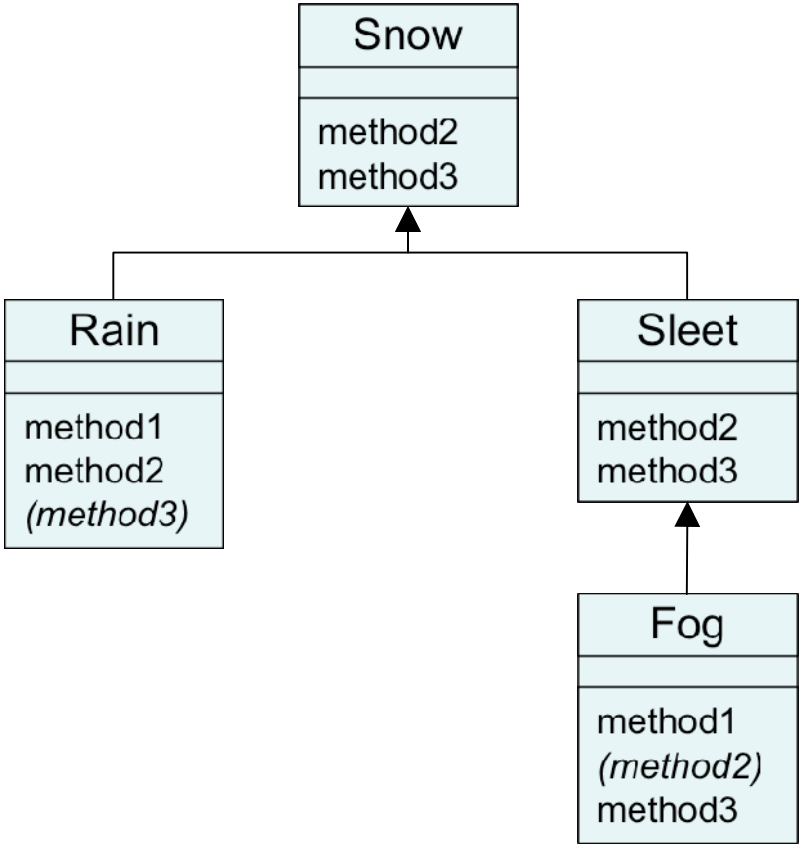
```
Snow var2 = new Rain();
var2.method1();
```

- 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`

Italic - inherited behavior
Bold - dynamic method call

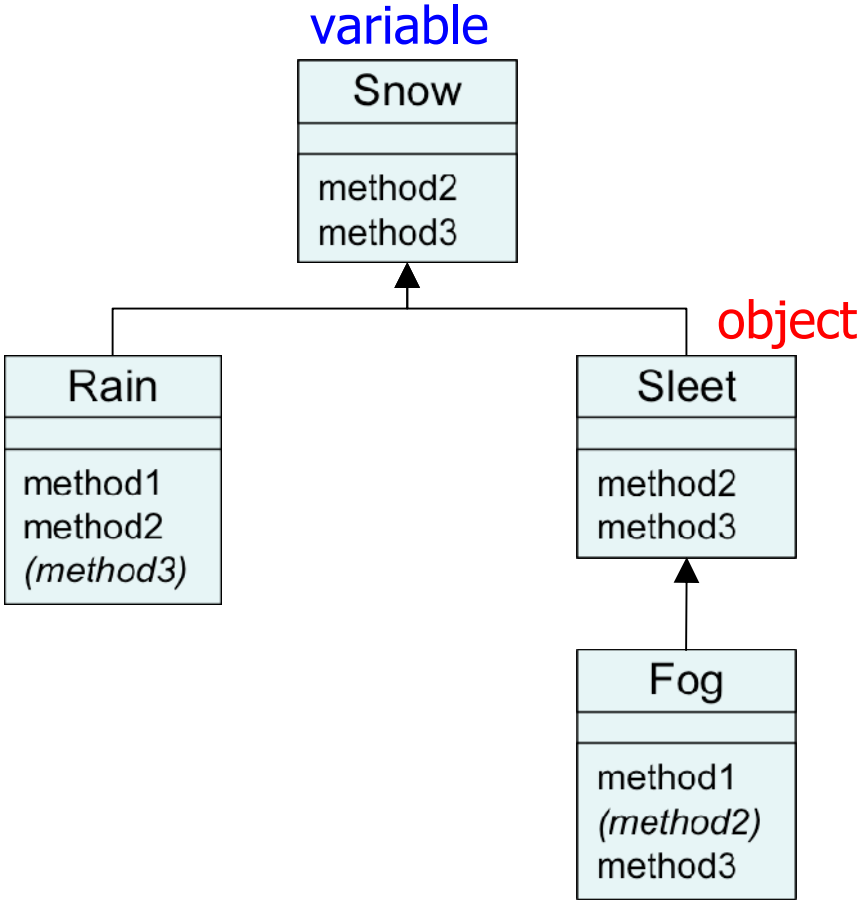
Example 1

- Example:

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

- Output:

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



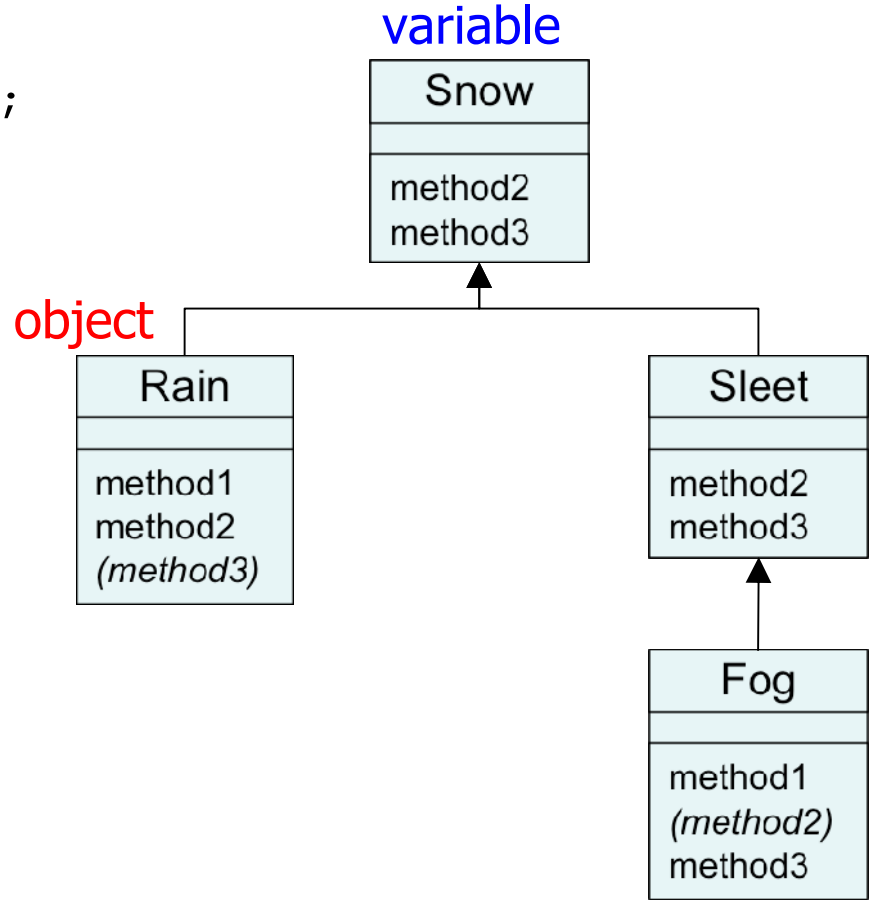
Example 2

- Example:

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

- Output:

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



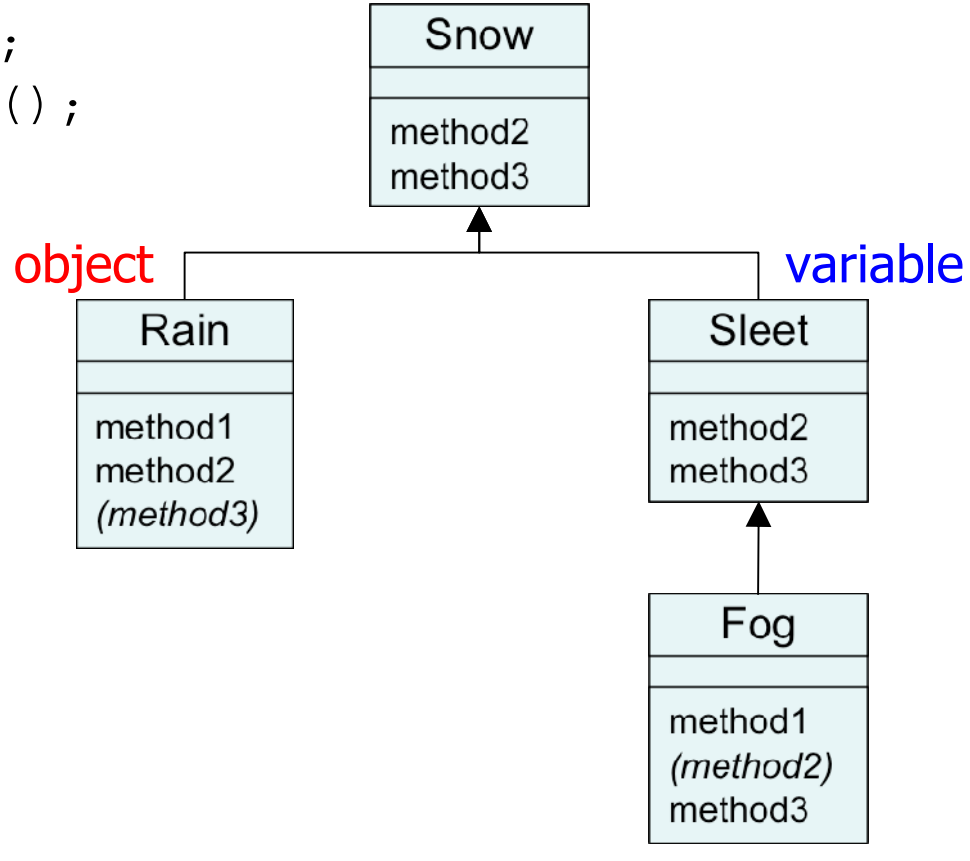
Example 3

- Example:

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

- Output:

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



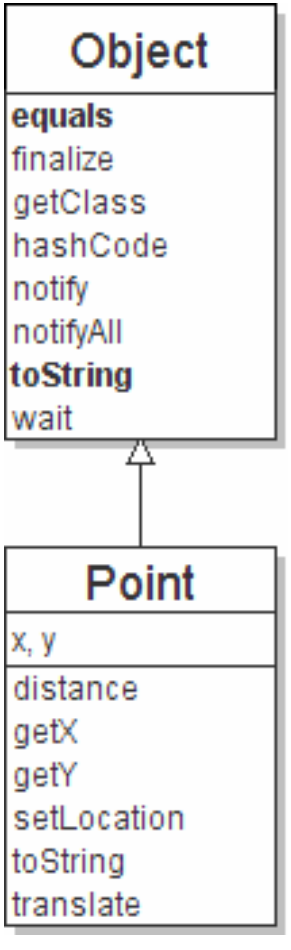
The `Object` class

read 9.3

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.



`Object` variables

- You can store any object in a variable of type `Object`.

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
Object o3 = new Scanner(System.in);
```

- An `Object` variable only knows how to do general things.

```
String s = o1.toString();       // ok
int len = o2.length();          // error
String line = o3.nextLine();    // error
```

- You can write methods that accept an `Object` parameter.

```
public void checkForNull(Object o) {
    if (o == null) {
        throw new IllegalArgumentException();
    }
}
```

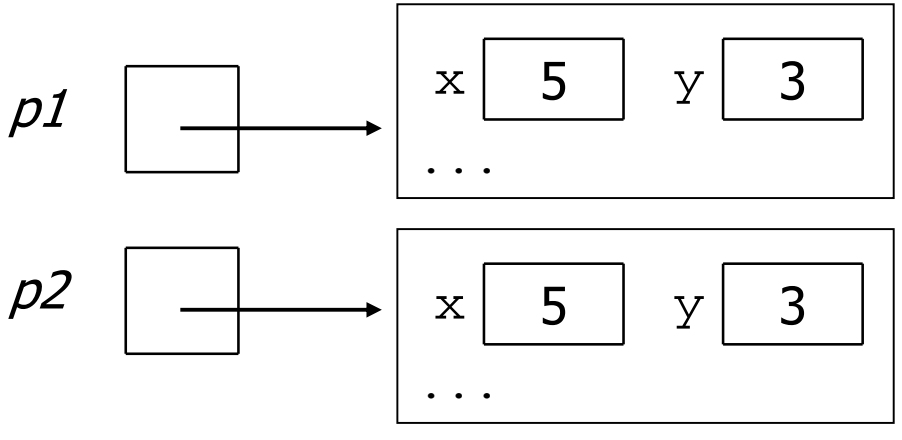
Recall: comparing objects

- 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

- The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {
    System.out.println("the strings are equal");
}
```

- But if you write a class, its `equals` method behaves like `==`

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

  - This is the behavior we inherit from class `Object`.
  - Java doesn't understand how to compare `Point`s by default.

Flawed `equals` method

- We can change this behavior by writing an `equals` method.
  - Ours will *override* the default behavior from class `Object`.
  - The method should compare the state of the two objects and return `true` if they have the same x/y position.

- A flawed implementation:

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

Flaws in our method

- The body can be shortened to the following:

```
// boolean zen
return x == other.x && y == other.y;
```

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

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

  - `equals` should always return `false` if a non-`Point` is passed.

`equals` and `Object`

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

- The parameter to `equals` must be of type `Object`.
- `Object` is a general type that can match any object.
- Having an `Object` parameter means *any* object can be passed.
  - If we don't know what type it is, how can we compare it?

Another flawed version

- Another flawed `equals` implementation:

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

- It does not compile:

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

  - The compiler is saying,
    "`o` could be any object. Not every object has an `x` field."

Type-casting objects

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

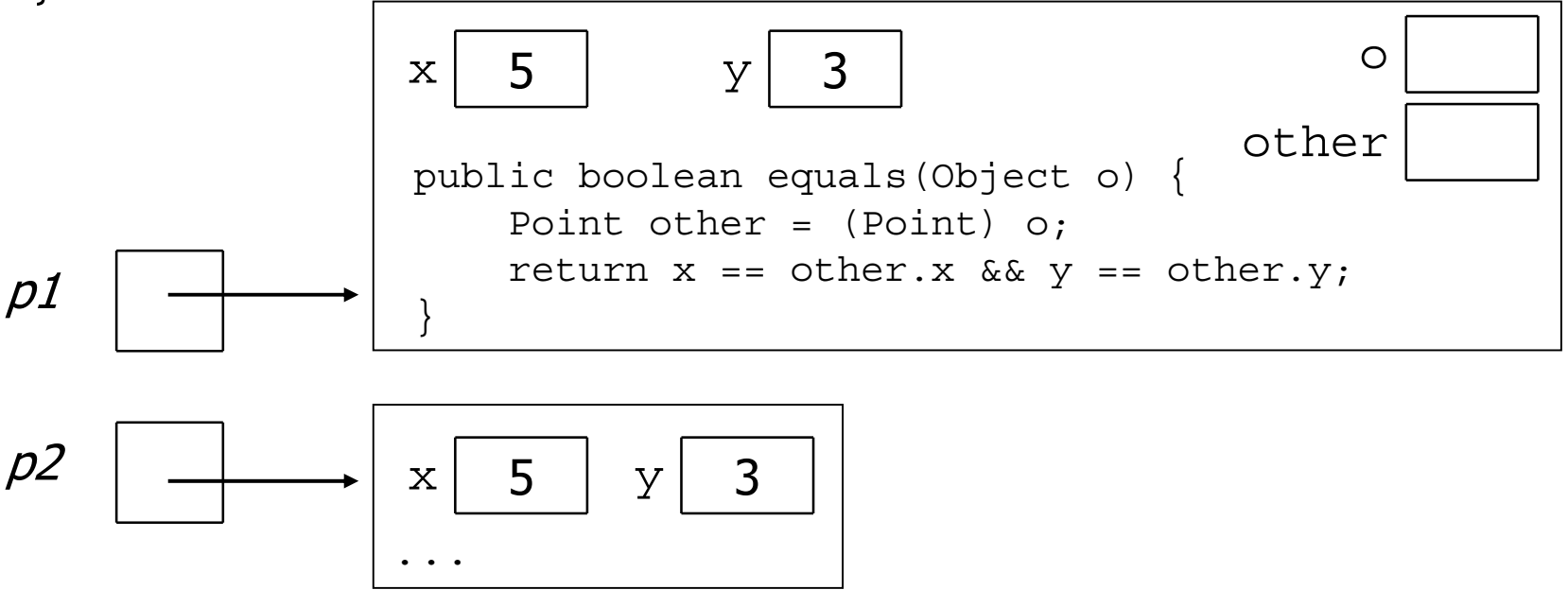
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Casting objects is different than casting primitives.
  - Really casting an `Object` reference into a `Point` reference.
  - Doesn't actually change the object that was passed.
  - Tells the compiler to *assume* that `o` refers to a `Point` object.

Casting objects diagram

- Client code:

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



Comparing different types

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

- Currently our method crashes on the above code:

```
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

```
if (variable instanceof type) {
    statement(s);
}
```

- Asks if a variable refers to an object of a given type.
  - Used as a `boolean` test.

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

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

Final `equals` method

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point.
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 {
        // o is not a Point; cannot be equal
        return false;
    }
}
```