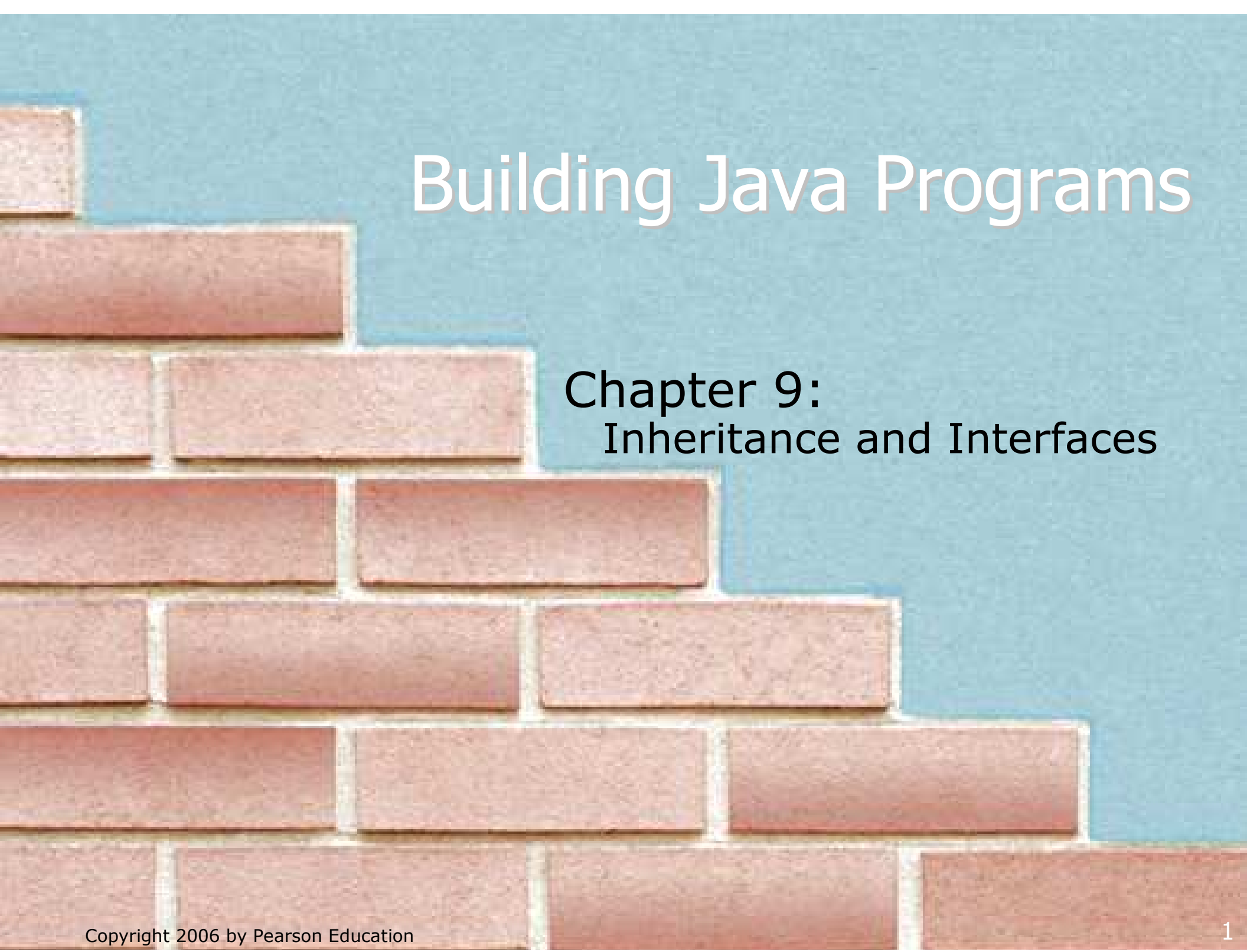
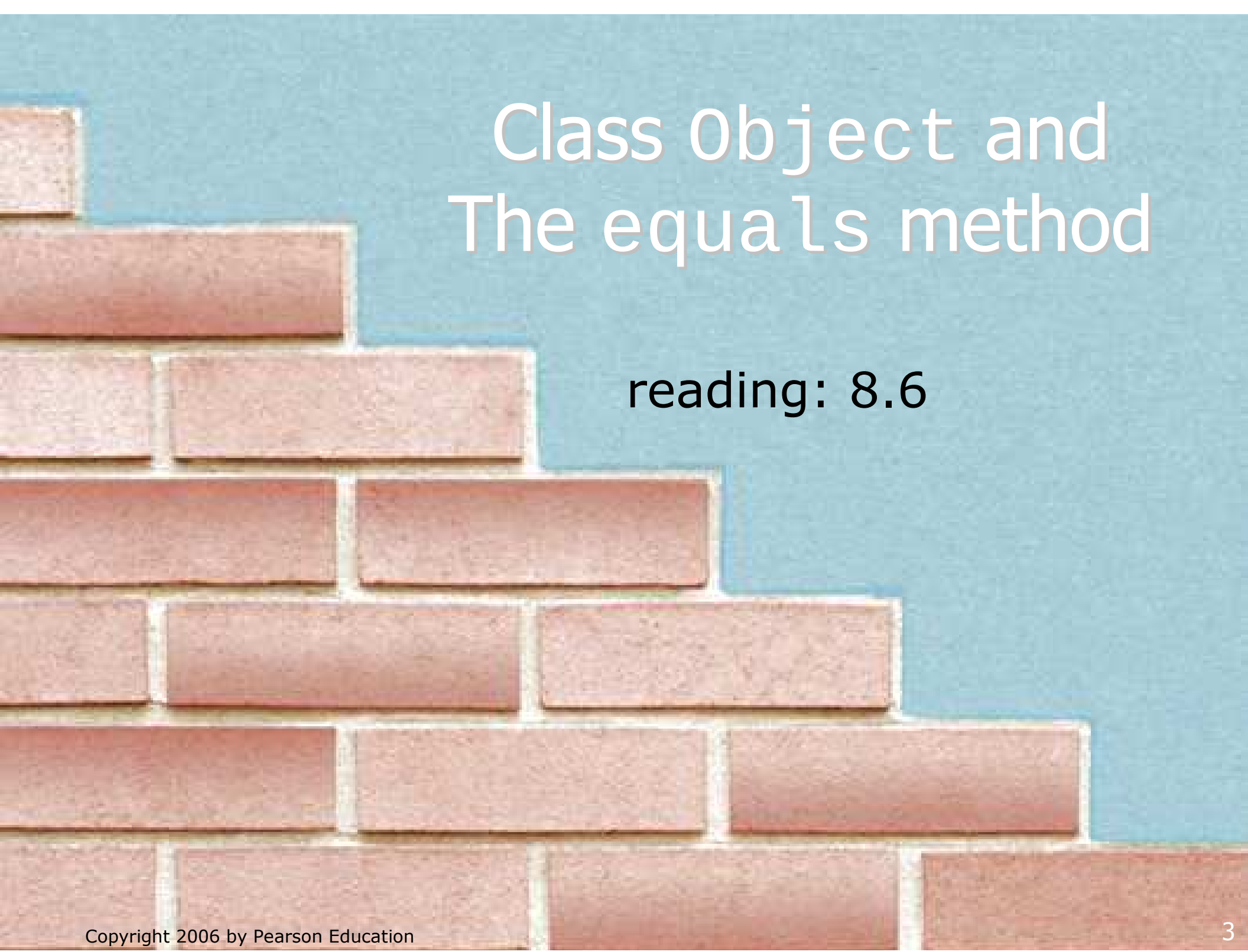


Lecture outline

- the `equals` method

- polymorphism
  - "inheritance mystery" problems



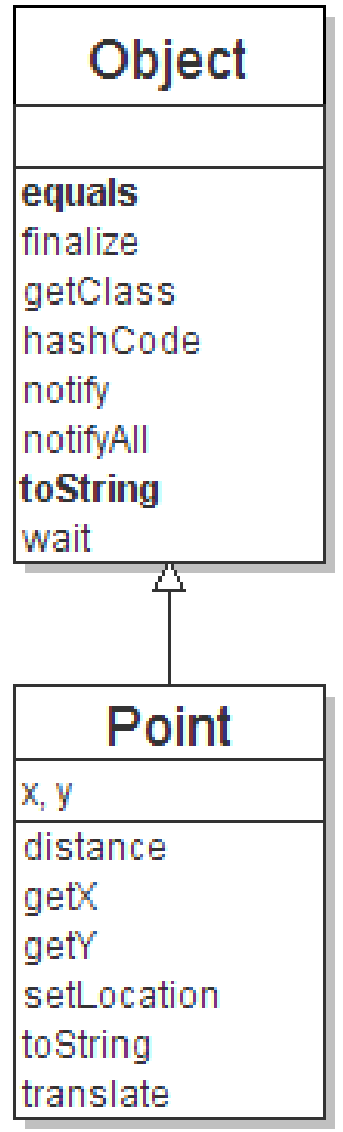
Class `Object` and The `equals` method

reading: 8.6

Class Object

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

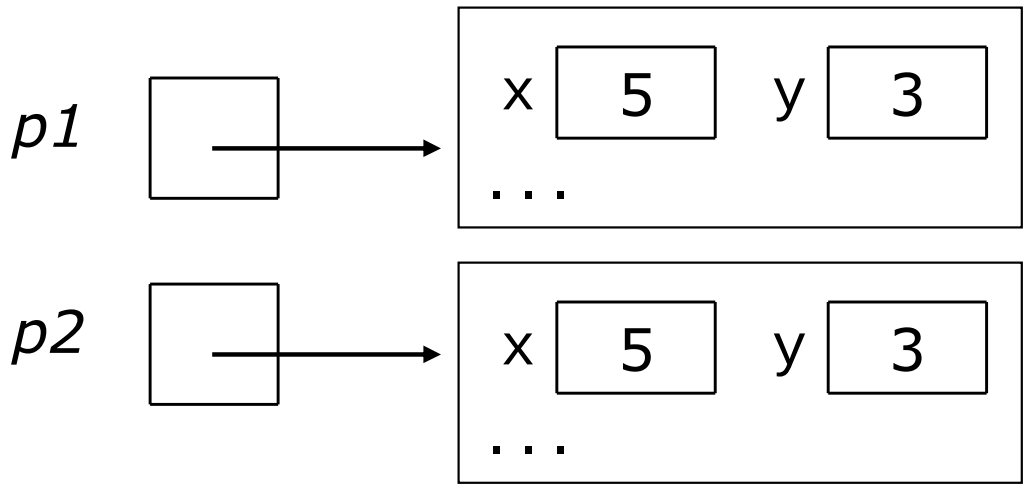
- The Object class defines several methods:
  - `public String toString()`
    Used to print the object.
  - `public boolean equals(Object other)`
    Compare the object to any other for equality.



Comparing objects

- The == operator does not work well with objects.
  - == compares references to objects, not their state.
  - Example:

```
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.
  - `equals` should be used when comparing `Strings`, `Points`, ...

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

- If you write your own class, its `equals` method will behave just like the == operator.

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

  - This is the behavior we inherit from class `Object`.

Initial 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` for cases like the above.

- A flawed implementation of the `equals` method:

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

Flaws in equals 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` objects):

```
// 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 the Object class

- `equals` method, general syntax:

```
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.

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`.

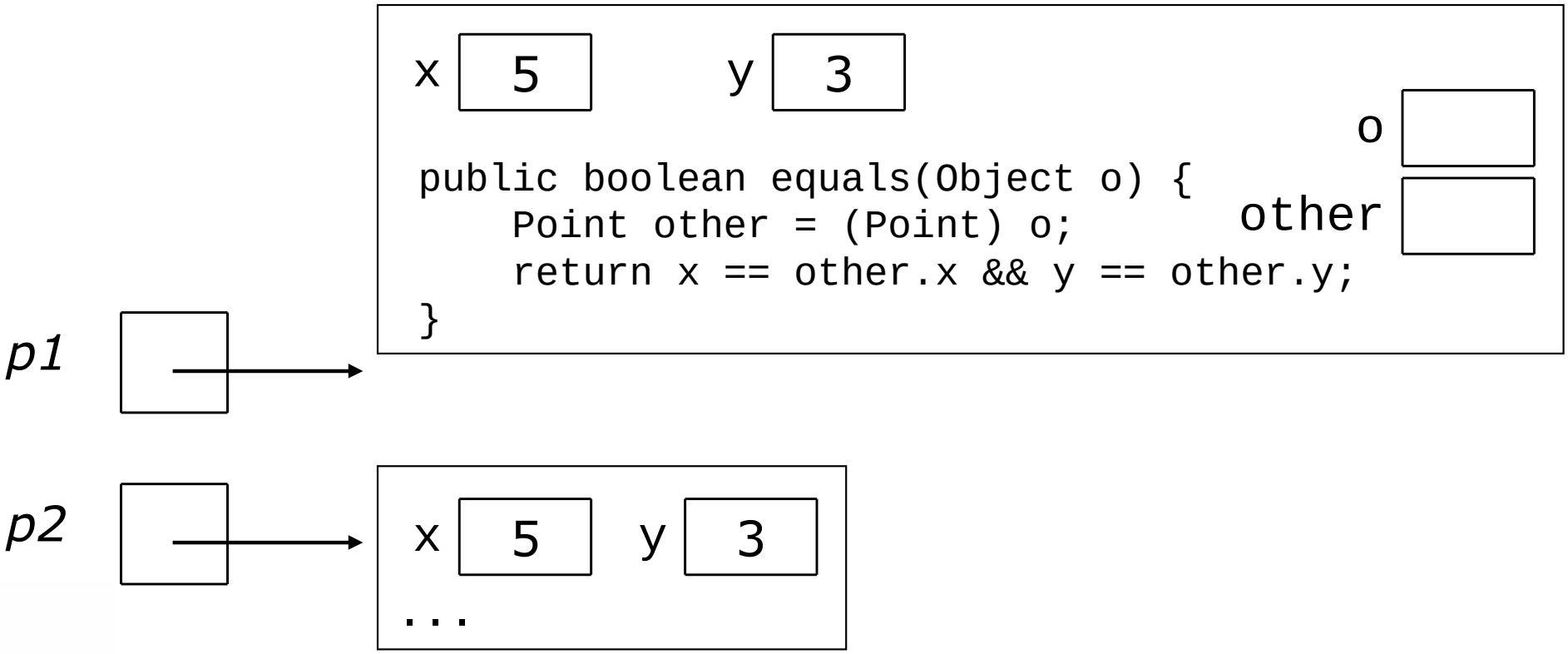
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Casting objects is different than casting primitives.
  - We're really casting an `Object` reference into a `Point` reference.
  - We're promising the compiler 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

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

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

  - Currently 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

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type.

- The `instanceof` keyword, general syntax:

  <variable> `instanceof` ***<type>***

  - The above is a `boolean` expression.

  - Examples:

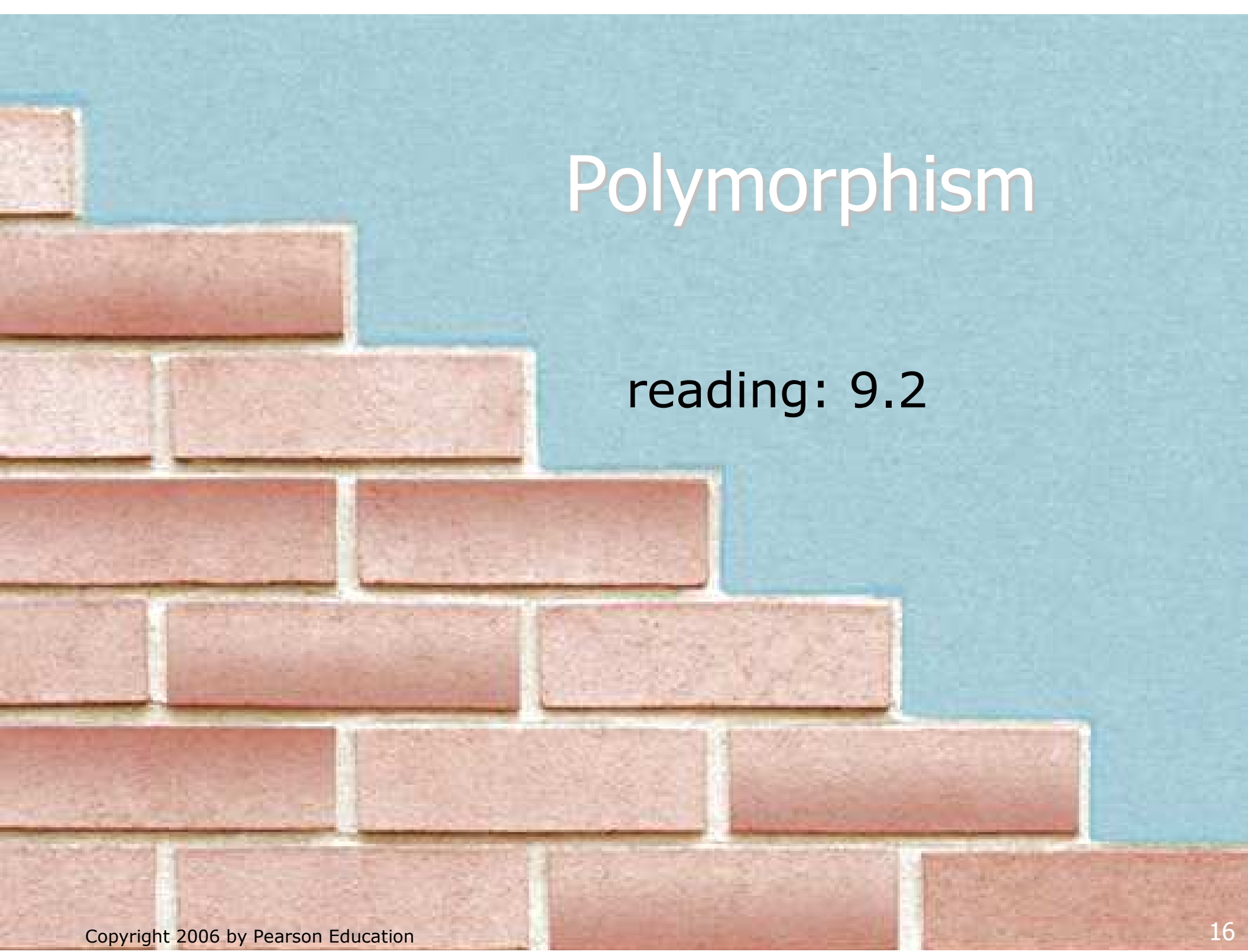
```
String s = "hello";
Point p = new Point();
```

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

Final version of 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 {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

- This version correctly compares `Points` to any type of object.



Polymorphism

- **polymorphism**: The ability for the same code to be used with several different types of objects, and behave differently depending on the type of object used.

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

```
Employee person = new Lawyer();
System.out.println(person.getSalary());          // 50000.0
System.out.println(person.getVacationForm());    // pink
```

  - You can call any methods from `Employee` on the variable `person`, but not any methods specific to `Lawyer` (such as `sue`).
  - Once a method is called on the object, it behaves in its normal way (as a `Lawyer`, not as a normal `Employee`).

Polymorphism + parameters

- You can declare methods to accept superclass types as parameters, then pass a parameter of any subtype.

```
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
vacation days = 21
vacation form = pink

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

Polymorphism + arrays

- You can declare arrays of superclass types, and store objects of any subtype as elements.

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

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

  - OUTPUT:

```
salary = 50000.0
vacation days = 15

salary = 50000.0
vacation days = 10

salary = 60000.0
vacation days = 10

salary = 55000.0
vacation days = 10
```

Polymorphism problems

- The textbook has several useful exercises to test your knowledge of polymorphism.
  - Each exercise declares a group of approximately 4 or 5 short classes with inheritance is-a relationships between them.
  - A client program calls methods on objects of each class.
  - Your task is to read the code and determine the client's output.

  (Example on next slide...)

A polymorphism problem

- Assume 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");
    }
}
```

(continued on next slide)

A polymorphism problem

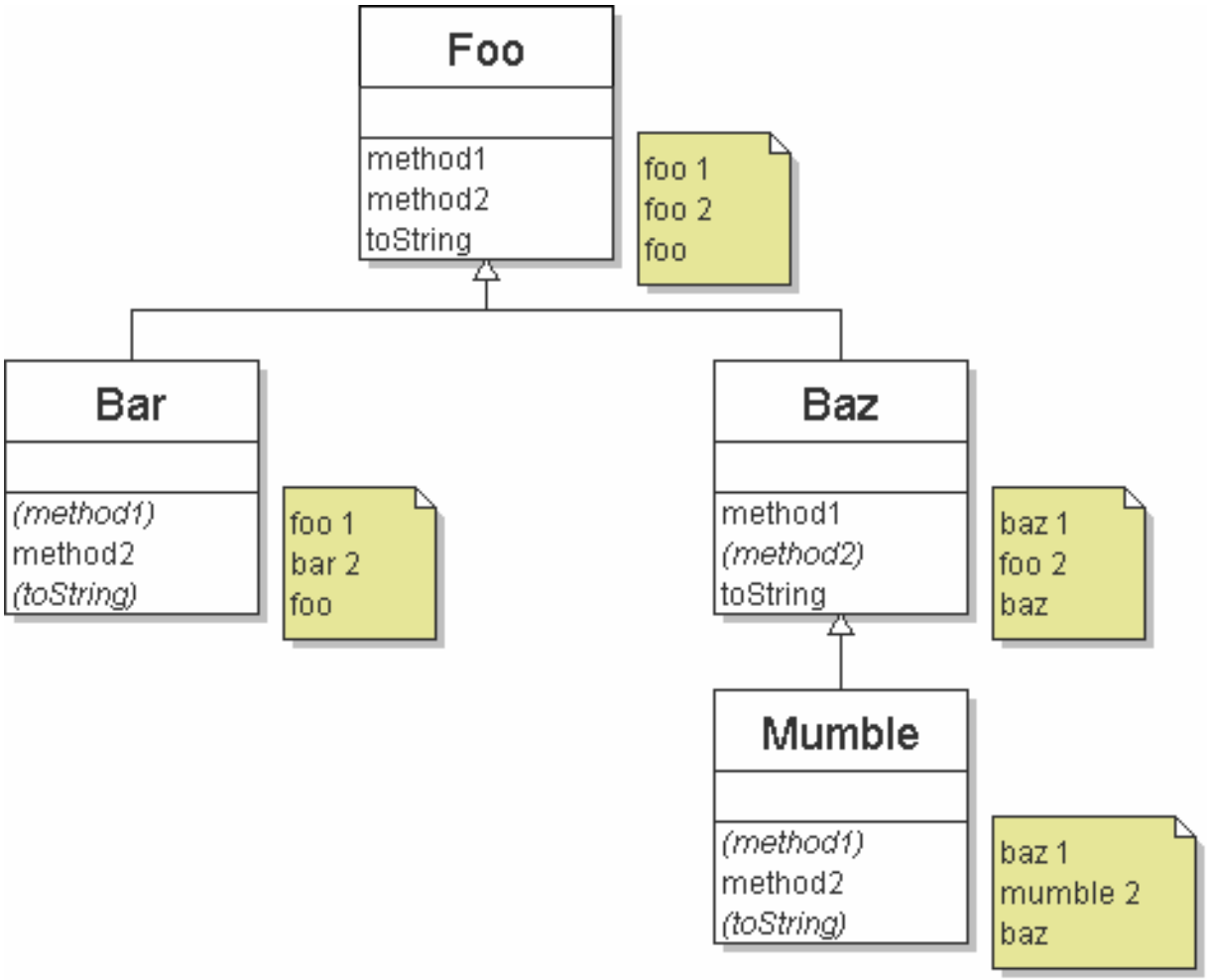
```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    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();
}
```

Finding output with diagrams

- One way to determine the output is to diagram each class and its methods, including their output:
  - Add the classes from top (superclass) to bottom (subclass).
  - Include any inherited methods and their output.



Finding output with tables

- Another possible technique for solving these problems is to make a table of the classes and methods, writing the output in each square.

method	Foo	Bar	Baz	Mumble
`method1`	`foo 1`	*`foo 1`*	`baz 1`	*`baz 1`*
`method2`	`foo 2`	`bar 2`	*`foo 2`*	`mumble 2`
`toString`	`foo`	*`foo`*	`baz`	*`baz`*

Polymorphism answer

```
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();
}
```

- The code produces the following output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

Another problem

- Assume that the following classes have been declared:
  - The order of classes is changed, as well as the client.
  - The methods now sometimes call other methods.

```
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";
    }
}
```

Another problem 2

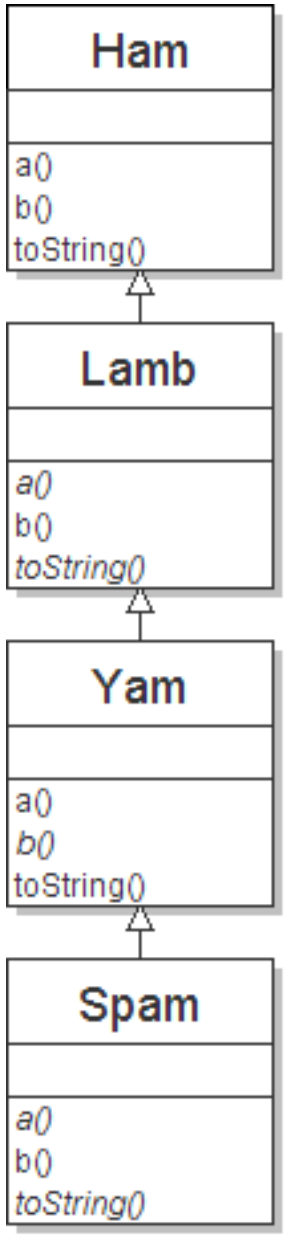
```
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b   ");
    }
}

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 Spam(), new Yam(), new Ham(), new Lamb()};
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();
}
```

The class diagram

- The following diagram depicts the class hierarchy:



Polymorphism at work

- Notice that `Ham`'s `a` method calls `b`. `Lamb` overrides `b`.
  - This means that calling `a` on a `Lamb` will also have a new result.

```
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 a output: `Ham a   Lamb b`

The table

- Fill out the following table with each class's behavior:

method	Ham	Lamb	Yam	Spam
`a`				
`b`				
`toString`				

The answer

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

- The code produces the following output:

```
Yam
Yam a   Ham a   Spam b
Spam b

Yam
Yam a   Ham a   Lamb b
Lamb b

Ham
Ham a   Ham b
Ham b

Ham
Ham a   Lamb b
Lamb b
```