Building Java Programs

Chapter 8

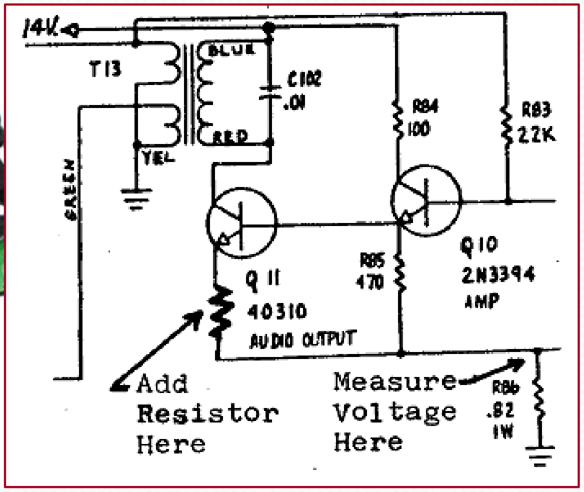
Lecture 8-3: Encapsulation; `this`; comparing objects

reading: 8.3 - 8.4; 9.2

Encapsulation

- **encapsulation**: Hiding implementation details from clients.

  - Encapsulation forces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data



Private fields

A field that cannot be accessed from outside the class

```
private type name;
```

- Examples:

```
private int id;
private String name;
```

- Client code won't compile if it accesses private fields:

```
PointMain.java:11: x has private access in Point
System.out.println(p1.x);
                      ^
```

Accessing private state

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println(p1.getX());
p1.setX(14);
```

Point class, version 4

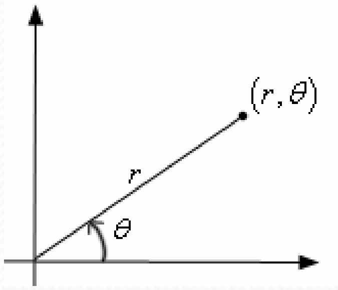
```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
}
```

Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.

- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates (*r*, *θ*) with the same methods.



- Can constrain objects' state (**invariants**)
  - Example: Only allow `Account`s with non-negative balance.
  - Example: Only allow `Date`s with a month from 1-12.

The keyword `this`

reading: 8.3

The `this` keyword

- **`this`** : Refers to the implicit parameter inside your class.

  (a variable that stores the object on which a method is called)

  - Refer to a field: `this.`**`field`**
  - Call a method: `this.`**`method`**`(`**`parameters`**`);`
  - One constructor can call another: `this(`**`parameters`**`);`

Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

```
public class Point {
    private int x;
    private int y;

    ...

    // this is legal
    public void setLocation(int x, int y) {
        ...
    }
```

  - In most of the class, `x` and `y` refer to the fields.
  - In `setLocation`, `x` and `y` refer to the method's parameters.

Fixing shadowing

```
public class Point {
    private int x;
    private int y;

    ...

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- Inside `setLocation`,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

Calling another constructor

```
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);      // calls (x, y) constructor
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

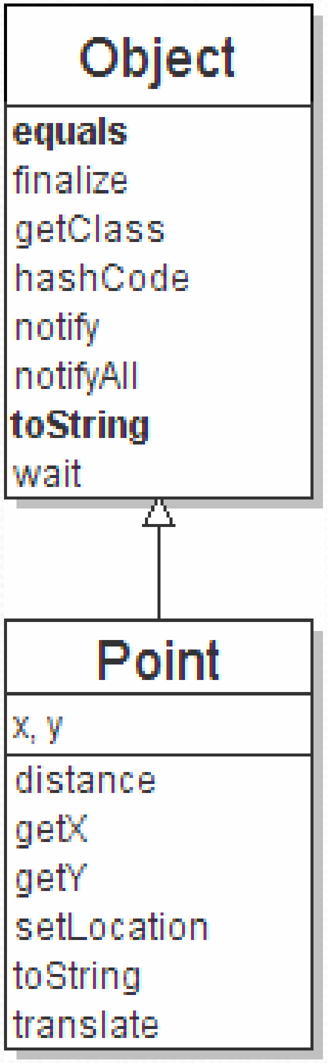
The `equals` method

reading: 9.2

Class `Object`

- Java has a class named `Object`.
  - Every class is implicitly an `Object`

- The `Object` class defines several methods that become part of every class you write:
  - `public String toString()`
    Returns a text representation of the object, usually 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.



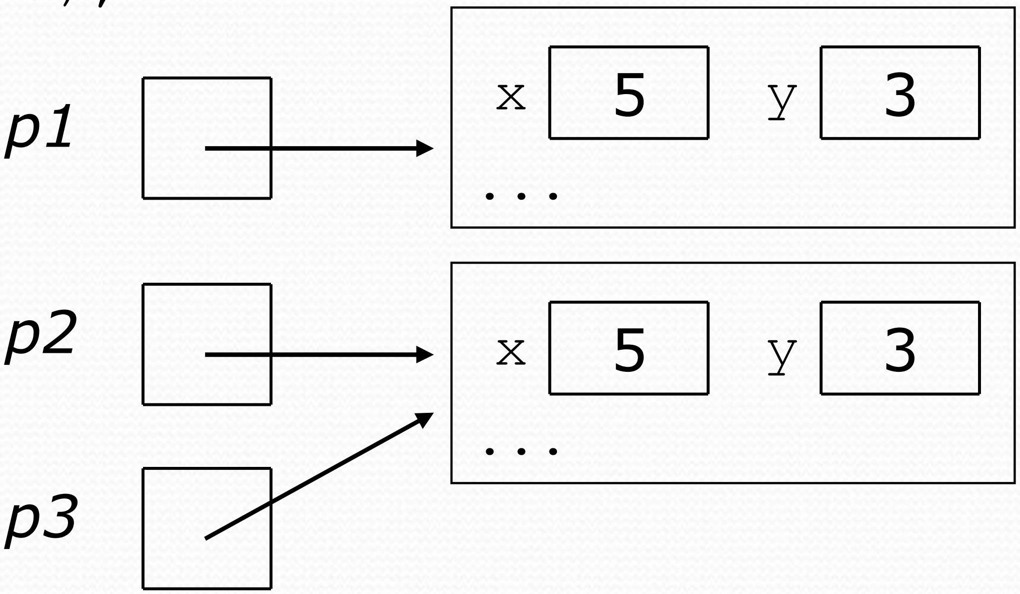
Recall: comparing objects

- 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);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true
```



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 default behavior we receive 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`.

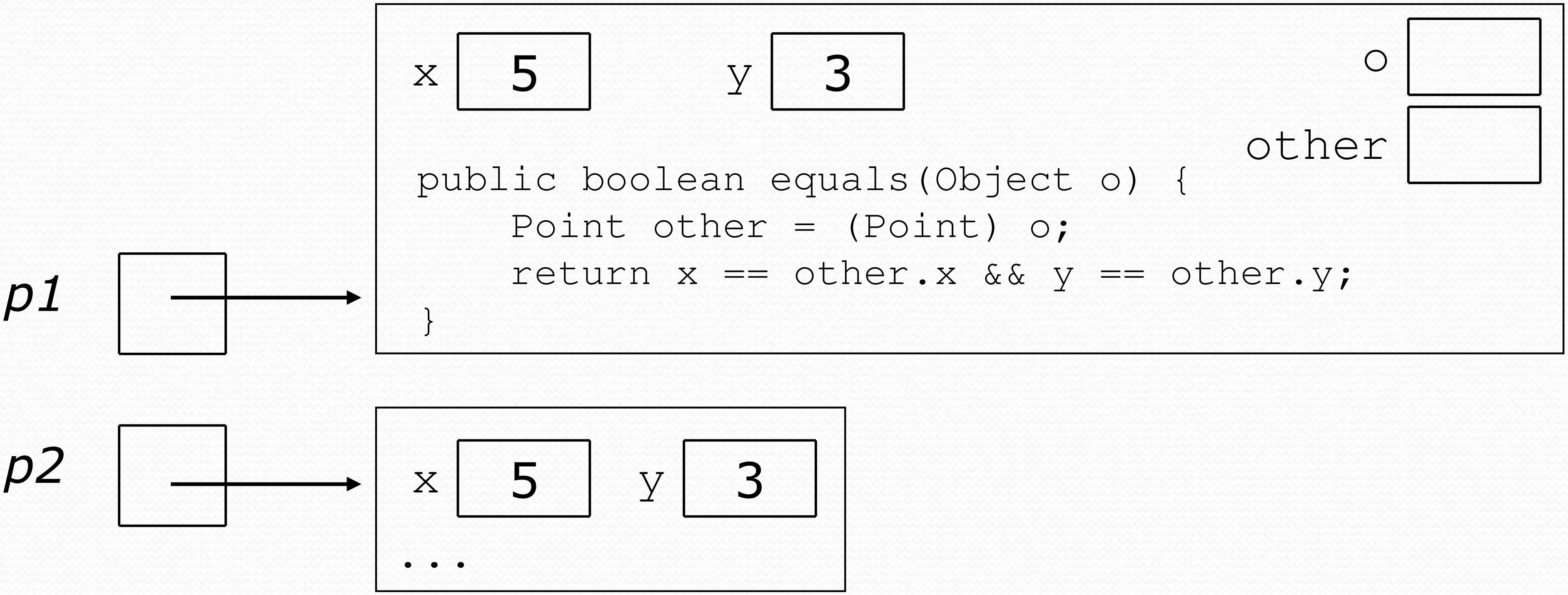
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- 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");
}
```



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

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;
    }
}
```