

# Building Java Programs

## Chapter 8

### Lecture 8-2: Object Methods and Constructors

**reading: 8.2 - 8.4**

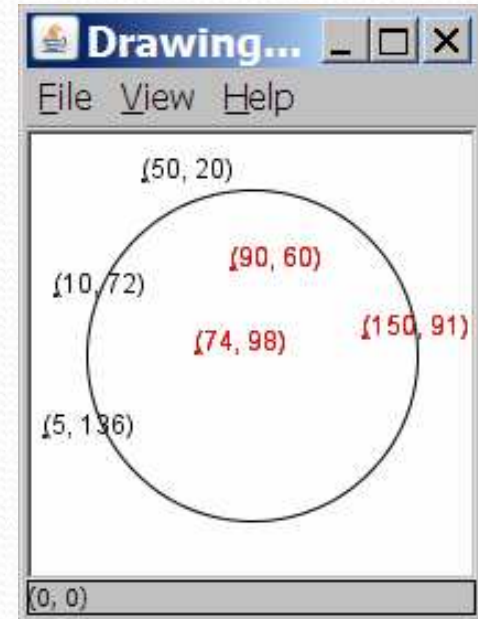
self-checks: #1-12

exercises: #1-4, 9, 11, 14, 16

# Recall: Bomb problem

- Given a file of cities' (x, y) coordinates, which begins with the number of cities:

```
6
50 20
90 60
10 72
74 98
5 136
150 91
```



- Write a program to draw the cities on a `DrawingPanel`, then drop a "bomb" that turns all cities red that are within a given radius:

```
Blast site x/y? 100 100
Blast radius? 75
```



# Object behavior: Methods

**reading: 8.3**

self-check: #7-9

exercises: #1-4

# Client code redundancy

- Our client program wants to draw `Point` objects:

```
// draw each city
```

```
g.fillOval(cities[i].x, cities[i].y, 3, 3);  
g.drawString("(" + cities[i].x + ", " + cities[i].y + ")",  
             cities[i].x, cities[i].y);
```

- To draw them in other places, the code must be repeated.
  - We can remove this redundancy using a method.



# Eliminating redundancy, v1

- We can eliminate the redundancy with a static method:

```
// Draws the given point on the DrawingPanel.  
public static void draw(Point p, Graphics g) {  
    g.fillOval(p.x, p.y, 3, 3);  
    g.drawString("(" + p.x + ", " + p.y + ")", p.x, p.y);  
}
```

- main would call the method as follows:

```
// draw each city  
draw(cities[i], g);
```

# Problems with static solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws `Points` would need a `draw` method.
- The syntax doesn't match how we're used to using objects.

```
draw(cities[i], g);    // static (bad)
```

- The point of classes is to combine state and behavior.
  - The `draw` behavior is closely related to a `Point`'s data.
  - The method belongs *inside* each `Point` object.

```
cities[i].draw(g);    // inside object (better)
```



# Instance methods

- **instance method**: One that exists inside each object of a class and defines behavior of that object.

```
public type name(parameters) {  
    statements;  
}
```

- same syntax as static methods, but without `static` keyword

Example:

```
public void shout() {  
    System.out.println("HELLO THERE!");  
}
```

# Instance method example

```
public class Point {  
    int x;  
    int y;  
  
    // Draws this Point object with the given pen.  
    public void draw(Graphics g) {  
        ...  
    }  
}
```

- The `draw` method no longer has a `Point p` parameter.
- How will the method know which point to draw?
  - How will the method access that point's x/y data?



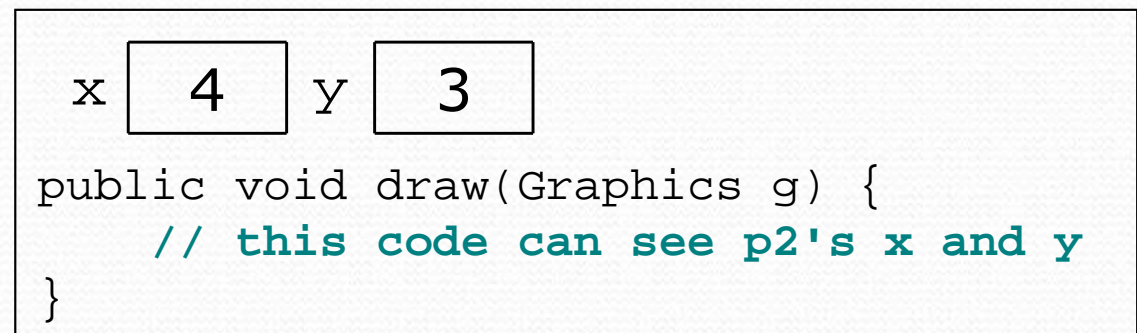
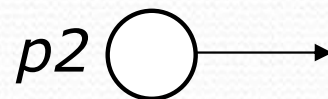
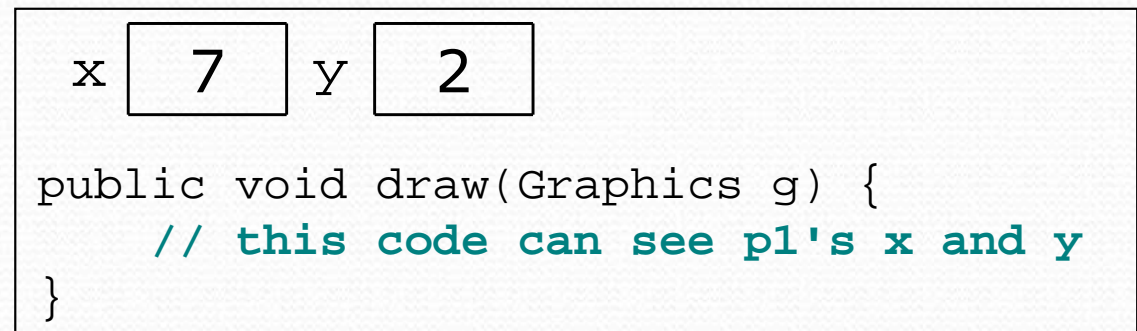
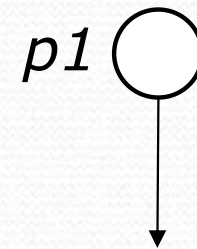
# Point objects w/ method

- Each Point object has its own copy of the draw method, which operates on that object's state:

```
Point p1 = new Point();  
p1.x = 7;  
p1.y = 2;
```

```
Point p2 = new Point();  
p2.x = 4;  
p2.y = 3;
```

```
p1.draw(g);  
p2.draw(g);
```



# The implicit parameter

- **implicit parameter:**

The object on which an instance method is called.

- During the call `p1.draw(g)` ;  
the object referred to by `p1` is the implicit parameter.
- During the call `p2.draw(g)` ;  
the object referred to by `p2` is the implicit parameter.
- The instance method can refer to that object's fields.
  - We say that it executes in the *context* of a particular object.
  - `draw` can refer to the `x` and `y` of the object it was called on.



# Point class, version 2

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

    // Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

- Now each `Point` object contains a method named `draw` that draws that point at its current `x/y` position.

# Kinds of methods

- Instance methods take advantage of an object's state.
  - Some methods allow clients to access/modify its state.
- **accessor**: A method that lets clients examine object state.
  - Example: A `distanceFromOrigin` method that tells how far a `Point` is away from `(0, 0)`.
  - Accessors often have a non-`void` return type.
- **mutator**: A method that modifies an object's state.
  - Example: A `translate` method that shifts the position of a `Point` by a given amount.



# Mutator method questions

- Write a method `setLocation` that changes a `Point`'s location to the  $(x, y)$  values passed.
  - You may want to refactor the `Point` class to use this method.
- Write a method `translate` that changes a `Point`'s location by a given  $dx, dy$  amount.
- Modify the client code to use these methods as appropriate.

# Mutator method answers

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

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

**// alternative solution**

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



# Accessor method questions

- Write a method `distance` that computes the distance between a `Point` and another `Point` parameter.

Use the formula:  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- Write a method `distanceFromOrigin` that returns the distance between a `Point` and the origin, `(0, 0)`.
- Modify the client code to use these methods.

# Accessor method answers

```
public double distance(Point other) {  
    int dx = x - other.x;  
    int dy = y - other.y;  
    return Math.sqrt(dx * dx + dy * dy);  
}
```

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

**// alternative solution**

```
public double distanceFromOrigin() {  
    return distance(new Point());  
}
```



# Object initialization: constructors

**reading: 8.4**

self-check: #10-12

exercises: #9, 11, 14, 16

# Initializing objects

- Currently it takes 3 lines to create a `Point` and initialize it:

```
Point p = new Point();  
p.x = 3;  
p.y = 8;           // tedious
```

- We'd rather pass the fields' initial values as parameters:

```
Point p = new Point(3, 8); // better!
```

- We are able to do this with most types of objects in Java.



# Constructors

- **constructor**: Initializes the state of new objects.

```
public type(parameters) {  
    statements;  
}
```

- runs when the client uses the `new` keyword
- does not specify a return type;  
it implicitly returns the new object being created
- If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

# Constructor example

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

    // Constructs a Point at the given x/y location.
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

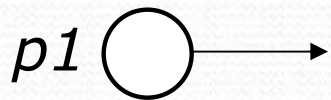
    public void translate(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```



# Tracing a constructor call

- What happens when the following call is made?

```
Point p1 = new Point(7, 2);
```



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

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

# Client code, version 3

```
public class PointMain3 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
    }
}
```

## OUTPUT:

```
p1: (5, 2)
p2: (4, 3)
p2: (6, 7)
```



# Common constructor bugs

- Accidentally writing a return type such as `void`:

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

- This is not a constructor at all, but a method!
- Storing into local variables instead of fields ("shadowing"):

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

- This declares local variables with the same name as the fields, rather than storing values into the fields. The fields remain 0.

# Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.
- Write a constructor for Point objects that accepts no parameters and initializes the point to the origin, (0, 0).

```
// Constructs a new point at (0, 0).  
public Point() {  
    x = 0;  
    y = 0;  
}
```