Building Java Programs

Chapter 8
Lecture 8-2: Object Behavior (Methods) and Constructors; Encapsulation

reading: 8.2 – 8.4

Recall: Instance methods

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```
public <type> <name>(<parameters>) {
    <statement(s)>;
}
```

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

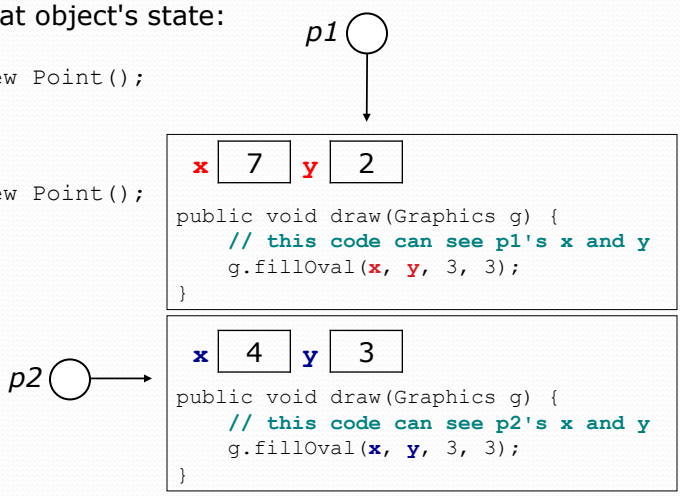
Example:

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

`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);
```



Kinds of methods

- **accessor**: A method that lets clients examine object state.
  - Examples: `distance`, `distanceFromOrigin`
  - often has a non-`void` return type

- **mutator**: A method that modifies an object's state.
  - Examples: `setLocation`, `translate`

Mutator method questions

- Write a method `setLocation` that changes a `Point`'s location to the (*x*, *y*) values passed.

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

- Modify the `translate` method from the previous lecture to use `setLocation`.

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

5

Accessor method questions

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

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

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

- Modify the `distanceFromOrigin` from the previous lecture to use `distance`.

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

6

Any redundancies?

```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.x = 5;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.x = 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)
```


Printing objects

- By default, Java doesn't know how to print objects:

```
Point p = new Point();
p.x = 10;
p.y = 7;
System.out.println("p is " + p);  // p is Point@9e8c34


// better, but cumbersome;           p is (10, 7)
System.out.println("p is (" + p.x + ", " + p.y + ")");


// desired behavior
System.out.println("p is " + p);  // p is (10, 7)
```

The `toString` method

tells Java how to convert an object into a `String`

```
Point p1 = new Point(7, 2);
System.out.println("p1: " + p1);

// the above code is really calling the following:
System.out.println("p1: " + p1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name `@` object's memory address

```
Point@9e8c34
```


`toString` syntax

```
public String toString() {
    <code that returns a String representation>;
}
```

- Method header must match exactly.
- Example:

```
// Returns a String representing this Point
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

Client code

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

        // print each point
        System.out.println("p1: " + p1);
        System.out.println("p2: " + p2);
        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: " + p2);
    }
}
```

OUTPUT:

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


Object initialization: constructors

reading: 8.3

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 specify the fields' initial values at the start:

```
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>) {
    <statement(s)>;
}
```

  - runs when the client uses the `new` keyword
  - How does this differ from previous methods?
    - no return type is specified;
      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

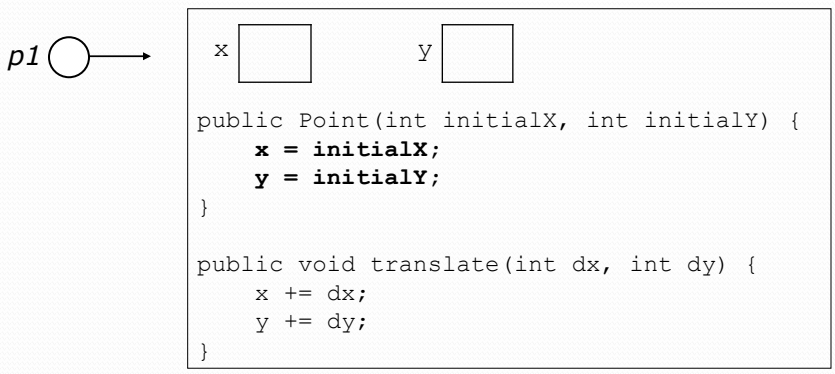
```
public class Point {
    int x;
    int y;

    // 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 = x + dx;
        y = y + dy;
    }

    ...
}
```

15

Tracing a constructor call

- What happens when the following call is made?

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



16

Common constructor bugs

1. Re-declaring fields as local variables ("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.

2. Accidentally giving the constructor a return type:

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

- This is actually not a constructor, but a method named `Point`

Client code

```
public class PointMain {
    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);
        System.out.println("p2: " + p2);

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

OUTPUT:

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

Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.

- *Exercise:* Write a `Point` constructor with no parameters that initializes the point to (0, 0).

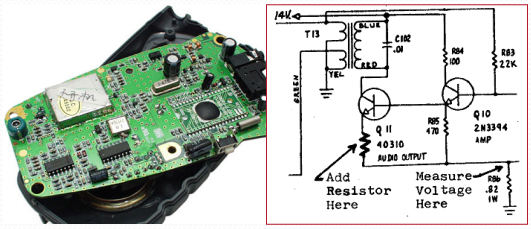
```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```


Encapsulation

reading: 8.4

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:

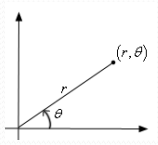
```
System.out.println(p1.getX());
p1.setX(14);
```


Point class

```
// 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.

25