Building Java Programs

Chapter 8
Lecture 8-2: Constructors and Encapsulation

reading: 8.3 - 8.4
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:

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

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

```java
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.

```java
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.
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?

```java
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;
}
```
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`:
  ```java
  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"):
  ```java
  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;
  }
Encapsulation

reading: 8.5 - 8.6
self-check: #13-17
exercises: #5
Encapsulation

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
    - Encapsulation protects the integrity of an object's data.
Private fields

• A field can be declared private.
  • No code outside the class can access or change it.

    private type name;

• Examples:

    private int id;
    private String name;

• Client code sees an error when accessing private fields:

    PointMain.java:11: x has private access in Point
    System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
    ^
Accessing private state

- We can provide methods to get and/or set a field's value:

  ```java
  // 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:

  ```java
  System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
  p1.setX(14);
  ```
// 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 double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

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

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

public class PointMain4 {
    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.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

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

OUTPUT:
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an Account's balance.
- Allows you to change the class implementation.
  - Point could be rewritten to use polar coordinates (radius $r$, angle $\theta$), but with the same methods.
- Allows you to constrain objects' state (invariants).
  - Example: Only allow Points with non-negative coordinates.