Building Java Programs

Chapter 8
Lecture 8-3: Encapsulation, toString

reading: 8.5 - 8.6
self-checks: #13-18, 20-21
exercises: #5, 9, 14
The `toString` method

reading: 8.6

self-check: #18, 20-21
exercises: #9, 14
Printing objects

- By default, Java doesn't know how to print objects:
  ```java
  Point p = new Point(10, 7);
  System.out.println("p: " + p);  // p is Point@9e8c34
  ```

- We can print a better string (but this is cumbersome):
  ```java
  System.out.println("p: (" + p.x + ", " + p.y + ")");
  ```

- We'd like to be able to print the object itself:
  ```java
  // desired behavior
  System.out.println("p: " + p);  // p is (10, 7)
  ```
The `toString` method

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

- called when an object is printed/concatenated to a `String`:
  ```java
  Point p1 = new Point(7, 2);
  System.out.println("p1 is " + p1);
  ```

- If you prefer, you can write `.toString()` explicitly.
  ```java
  System.out.println("p1 is " + p1.toString());
  ```

- Every class has a `toString`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:
    ```java
    Point@9e8c34
    ```
public String toString() {
    // code that returns a suitable String;
}

- The method name, return, parameters must match exactly.
- Example:

  // Returns a String representing this Point.
  public String toString() {
      return "(" + x + ", " + y + ");"
  }
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(7, 2);
        Point p2 = new Point(4, 3);
        // print each point
        System.out.println("p1: " + p1);
        System.out.println("p2: " + p2);
        // compute/print each point's distance from the origin
        System.out.println("p1's distance from origin: " + p1.distanceFromOrigin());
        System.out.println("p2's distance from origin: " + p2.distanceFromOrigin());
        // move p1 and p2 and print them again
        p1.translate(11, 6);
        p2.translate(1, 7);
        System.out.println("p1: " + p1);
        System.out.println("p2: " + p2);
        // compute/print distance from p1 to p2
        System.out.println("distance from p1 to p2: " + p1.distance(p2));
    }
}
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.

  ```java
  private type name;
  ```

- Examples:
  ```java
  private int id;
  private String name;
  ```

- Client code sees an error when accessing private fields:

  ```java
  PointMain.java:11: x has private access in Point
  System.out.println("p1 is (" + pl.x + ", " + pl.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: (" + pl.getX() + ", " + pl.getY() + ")");
pl.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.
The keyword this reading: 8.7
this

- **this**: A reference to the implicit parameter.
  - *implicit parameter*: object on which a method is called

- Syntax for using **this**:
  - To refer to a field:
    ```
    this.field
    ```
  - To call a method:
    ```
    this.method(parameters);
    ```
  - To call a constructor from another constructor:
    ```
    this(parameters);
    ```
Variable names and scope

- Usually it is illegal to have two variables in the same scope with the same name.

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

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

- The parameters to setLocation are named newX and newY to be distinct from the object's fields x and y.
Variable shadowing

- An instance method parameter can have the same name as one of the object's fields:

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

- Fields `x` and `y` are *shadowed* by parameters with same names.
- Any `setLocation` code that refers to `x` or `y` will use the parameter, not the field.
Avoiding shadowing w/ this

public class Point {
    private int x;
    private int y;
    ...
    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

- **Inside the setLocation method,**
  - When this.x is seen, the field x is used.
  - When x is seen, the parameter x is used.
Multiple constructors

- It is legal to have more than one constructor in a class.
- The constructors must accept different parameters.

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

    public Point() {
        x = 0;
        y = 0;
    }

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

    ...
}
```
Constructors and this

• One constructor can call another using this:

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

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

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