Building Java Programs

Chapter 8
Lecture 8-3: Encapsulation, this

reading: 8.5 - 8.6
self-checks: #13-17
exercises: #5
Abstraction

Don't need to know this

Can focus on this!!

AN x64 processor is screaming along at billions of cycles per second to run the XNU kernel, which is frantically working through all the POSIX-specified abstraction to create the Darwin system underlying OS X, which in turn is straining itself to run Firefox and its Gecko renderer, which creates a Flash object which renders dozens of video frames every second

Because I wanted to see a cat jump into a box and fall over.

I am a god.
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.
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

```java
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;
    }

    ...
}
```