hi

Building Java Programs

Chapter 8
Lecture 8-3: Encapsulation

reading: 8.4 - 8.6

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


The `toString` method

reading: 8.2

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 (base 16)

```
Point@9e8c34
```

hi

toString syntax

```
public String toString() {
    code that returns a String representing this object;
}
```

- Method name, return, and parameters must match exactly.
- Example:

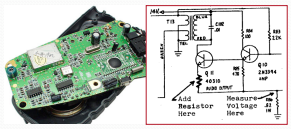
```
// Returns a String representing this Point.
public String toString() {
    return "(" + x + ", " + y + ")";
}
```


Encapsulation

reading: 8.4 – 8.5

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, version 4

```
// 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);
    }
}
```


bye

hi

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.

The keyword `this`

reading: 8.7

The `this` keyword

- **`this`** : Refers to the implicit parameter inside your class.
  (a variable that stores the object on which a method is called)
  - Refer to a field: `this.`**`field`**
  - Call a method: `this.`**`method`**`(`**`parameters`**`);`
  - One constructor can call another: `this(`**`parameters`**`);`

Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

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

    ...

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

  - In most of the class, `x` and `y` refer to the fields.
  - In `setLocation`, `x` and `y` refer to the method's parameters.

Fixing shadowing

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

    ...

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

- Inside `setLocation`,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

Calling another constructor

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

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

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

    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

bye