Building Java Programs

Chapter 8

Encapsulation, `this`, Subclasses

Today

- Finish our earthquake example
  - Use a `Circle` class to draw the circle and decide red-ness
- Encapsulation
  - A really big deal when writing larger programs
  - Need to use `private` fields on homework 8 (not difficult)
- The keyword `this`: Kind of a Chapter 8 loose end
- Subclasses and polymorphism
  - Will continue next Wednesday

Using the Circle class

- Has lots of features we don't need
  - That's normal
- Implementation uses some features we'll learn later today
  - But clients don't care
- Uses a `Point` object
  - It's normal for many classes to interact in many ways
- Simplifies the red-ness calculation
  - Just to clients, the `contains` method has the same computation

Encapsulation

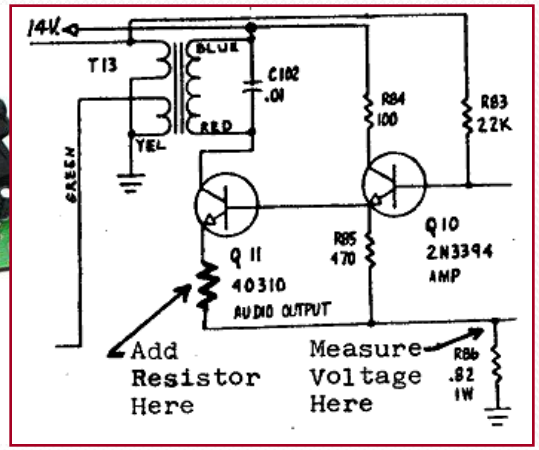
reading: 8.5 - 8.6

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:

```
// 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: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Point class, revised

```
// 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;
    }
}
```

Client code

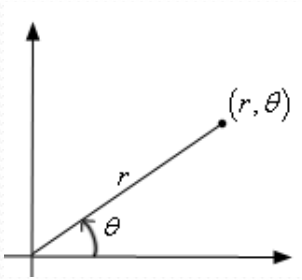
```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.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 *θ*), but with the same methods.
  - Like Apple building a cheaper iPod w/o you knowing



- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Point`s with non-negative coordinates.

Example: Polar points

```
// A Point object represents an (x, y) location.
// This version has a simpler distanceFromOrigin but more complicated
// everything else, but clients can't tell
public class Point {
    private double r;
    private double theta;

    public Point(int initialX, int initialY) {
      setLocation(initialX, initialY);
    }

    public double distanceFromOrigin() {
        return r;
    }

    public int getX() {
        return (int) (r * Math.cos(theta));
    }

    public int getY() {
        return (int) (r * Math.sin(theta));
    }

    public void setLocation(int newX, int newY) {
        r = Math.sqrt(newX * newX + newY * newY);
        theta = Math.atan2(newX, newY); // library method of just what we need
    }

    public void translate(int dx, int dy) {
       setLocation(dx + getX(), dy + getY());
    }
}
```

The keyword `this`

reading: 8.7

this

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

- Common uses for `this`:
  - To refer to a field (`this` is usually optional):
    `this.`**field**
  - To call a method (`this` is optional):
    `this.`**method**`(`**parameters**`);`
  - To use "yourself" as an argument:
    `this`
  - 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.

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

```
// 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.

Using `this` with shadowing

```
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.
- Can always use `this.x` for field access if you want

`this` for method calls

- We know one instance method can call another:

```
public String toString() {
   return "(" + x + ", " + y + ")";
}
public void draw(Graphics g) {
   g.fillOval(x, y, 2, 2);
   g.drawString(toString(), x, y);
}
```

- The implicit parameter is "passed along to the callee"
  - Can make this explicit if you want, but not necessary

```
public String toString() {
   return "(" + x + ", " + y + ")";
}
public void draw(Graphics g) {
   g.fillOval(x, y, 2, 2);
   g.drawString(this.toString(), x, y);
}
```

Passing yourself

- Occasionally want to pass "the whole current object" to another method

- Example that works as a more complicated replacement:
  - Instead of:

```
public double distFromOrigin() {
    Point p = new Point(0,0);
    return distance(p);
  }
```

  - Could do:

```
public double distFromOrigin() {
    Point p = new Point(0,0);
    return p.distance(this);
}
```

Multiple constructors

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

```
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`
  - This use of `this` is different from the others (weird but useful)

```
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;
    }

    ...
}
```

Inheritance

Chapter 9
Lecture 9-1: Inheritance

reading: 9.1 - 9.2

An `Employee` class

```
// A class to represent employees in general
public class Employee {
    public int getHours() {
        return 40;              // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;         // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;              // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";        // use the yellow form
    }
}
```

- Exercise: Implement class `TechWriter`, based on the previous employee regulations. (Tech writers can write manuals.)

Redundant `TechWriter` class

```
// A redundant class to represent tech writers.
public class TechWriter {
    public int getHours() {
        return 40;              // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;         // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;              // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";        // use the yellow form
    }

    public void writeManual(String app) {
        System.out.println("Writing a manual about: " + app);
    }
}
```

Desire for code-sharing

- `writeManual` is the only unique behavior in `TechWriter`.

- We'd like to be able to say:

```
// A class to represent tech writers.
public class TechWriter {
    copy all the contents from the Employee class;

    public void writeManual(String app) {
        System.out.println("Writing a manual about: " + app);
    }
}
```

Inheritance

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

- One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass

Inheritance syntax

```
public class name extends superclass {
```

- Example:

```
public class TechWriter extends Employee {
    ...
}
```

- By extending `Employee`, each `TechWriter` object now:
  - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
  - can be treated as an `Employee` by client code (seen later)

Improved `TechWriter` code

```
// A class to represent tech writers.
public class TechWriter extends Employee {
    public void writeManual(String app) {
        System.out.println("Writing a manual about: " + app);
    }
}
```

- Now we only write the parts unique to each type.
  - `TechWriter` inherits `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` methods from `Employee`.
  - `TechWriter` adds the `writeManual` method.

Mini-Exercise

- Define a Programmer class that includes a "designGame" method (these programmers work for a gaming company). This method should just print out an informative note.

Cheat sheet:

```
  // A class to represent tech writers.
public class TechWriter extends Employee {
   public void writeManual(String app) {
      System.out.println("Writing a manual about: " + app);
   }
}
```

Mini-Exercise - solution

- Define a Programmer class that includes a "designGame" method (these programmers work for a gaming company).

```
  // A class to represent programmers at a game company.
public class Programmer extends Employee {
   public void designGame(String name) {
      System.out.println(Designing the " + name + " game");
   }
}

// sample uses:
// Programmer chris = new Programmer();
// chris.designGame("Dragon5000");
// double dollars = chris.getSalary();
```

Implementing `Lawyer`

- Consider the following lawyer regulations:
  - Lawyers who get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
  - Lawyers have some unique behavior: they know how to sue.

- Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.

Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {
    // overrides getVacationForm method in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

  - Exercise: Complete the `Lawyer` class.
    - (3 weeks vacation, pink vacation form, can sue)

Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;              // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- Exercise: Complete the `Marketer` class. Marketers make $10,000 extra ($50,000 total) and know how to advertise.

Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A bilingual tech writer is the same as a regular tech writer but makes more money ($45,000) and can also write manuals in (say) German.

```
public class BilingualTechWriter extends TechWriter {
    ...
}
```

- Next time: Using the fact that any BilingualTechWriter or TechWriter is also an Employee
  - And the Java compiler knows it