Building Java Programs

Chapter 8

Lecture 8-1: Classes and Objects

reading: 8.1-8.3

self-checks: Ch. 8 #1-9

exercises: Ch. 8 #1-4

Problem

- Declaring same group of related variables several times in a program

```
int x1 = 3;
int y1 = 5;
int x2 = 12;
int y2 = 4;
```

  - Annoying and redundant
  - Unclear and hard to keep track of variables

Solution: Objects

- Group together related variables into an **object**
  - Like creating your own data structure out of Java building blocks

```
public class <object name> {
    <field(s)>;
}
```

- Syntax to use this data structure:

```
<object> <variable> = new <object>();
```

Solution: Objects

- Group together related variables into an **object**
  - Like creating your own data structure out of Java building blocks

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

- Syntax to use this data structure:

```
Point p1 = new Point();
```

Two Uses for Java Classes

- **class**: A program entity that represents either:
  1. A program / module, or
  2. **A template for a new type of objects.**

  - The `DrawingPanel` class is a template for creating `DrawingPanel` objects.

- **object**: An entity that combines state and behavior

Java class: Program

- An **executable program** with a **main method**
  - Can be run; statements execute procedurally
  - What we've been writing all quarter

```
public class BMI2 {
    public static void main(String[] args) {
        giveIntro();
        Scanner console = new Scanner(System.in);
        double bmi1 = getBMI(console);
        double bmi2 = getBMI(console);
        reportResults(bmi1, bmi2);
    }
    ...
}
```

Java class: Object Definition

- A **blueprint** for a new data type
  - Not executable, not a complete program
- Created objects are an **instance** of the class

- Blueprint:

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

- Instance:

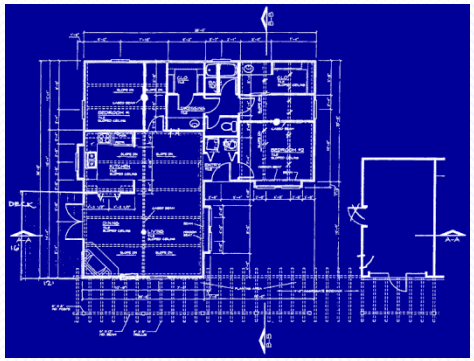
```
Point p1 = new Point();
```

Blueprint analogy

iPod blueprint

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song



creates

iPod #1

state:
song = "Octopus's Garden"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod #2

state:
song = "Lovely Rita"
volume = 9
battery life = 3.41 hrs

behavior:
power on/off
change station/song
change volume
choose random song

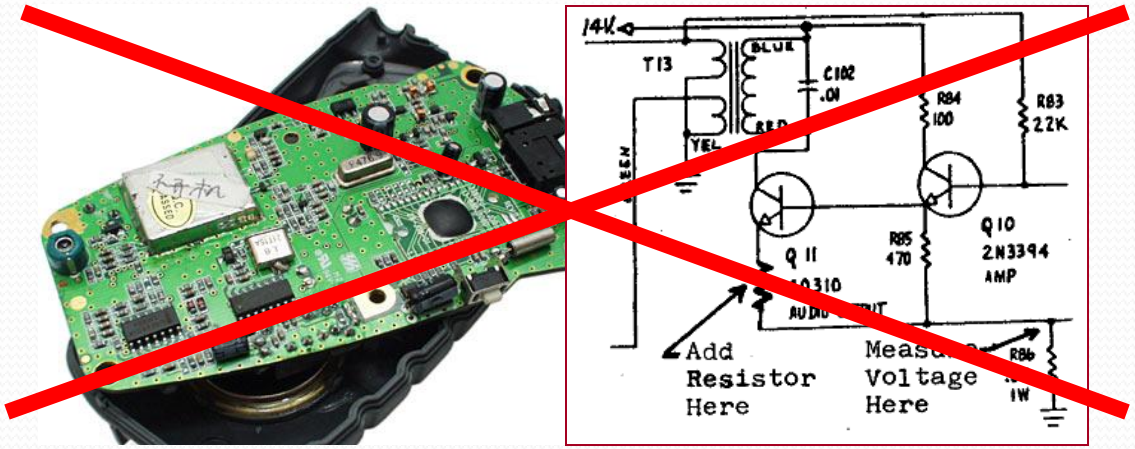
iPod #3

state:
song = "For No One"
volume = 24
battery life = 1.8 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Abstraction

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.
- abstraction in an iPod:
  - You understand its external behavior (buttons, screen).
  - You don't understand its inner details, and you don't need to.



Client and Object Classes

- **client program**: A program that uses objects.
  - Example: HW6 `Names` is a client of `DrawingPanel` and `Graphics`.

- **object**: An entity that combines state and behavior
  - *state*: data fields
  - *behavior*: methods

The Object Concept

- **procedural programming:** Programs that perform their behavior as a series of steps to be carried out

- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects
  - Takes practice to understand the object concept

Fields

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.

- Clients can access/modify an object's fields
  - access: **<variable>.<field>**
  - modify: **<variable>.<field>** = **<value>**;

- Example:

```
Point p1 = new Point();
Point p2 = new Point();
System.out.println("the x-coord is " + p1.x);    // access
p2.y = 13;                                       // modify
```

Behavior

- Objects can tie related data and *behavior* together
- **instance method:** A method inside an object that operates on that object

```
public <type> <name> (<parameter(s)>) {
    <statement(s)>;
}
```

- Syntax to use method:
  <variable>.<method>(<parameter(s)>);
- Example:

```
p1.translate(11, 6);
```

Implicit Parameter

- Each instance method call happens on a particular object.
  - Example: `p1.translate(11, 6);`
- The code for an instance method has an implied knowledge of what object it is operating on.
- **implicit parameter**: The object on which an instance method is called.
  - Can be referred to inside the object using `this` keyword

Accessors

- **accessor**: An instance method that provides information about the state of an object.

- Example:

```
public double distanceFromOrigin() {
    return Math.sqrt(x * x + y * y);
}
```

- This gives clients "read-only" access to the object's fields.

Mutators

- **mutator**: An instance method that modifies the object's internal state.

- Example:

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

- This gives clients both read and write access to code.

Building Java Programs

Chapter 8

Lecture 8-2: Constructors and Encapsulation

reading: 8.4 - 8.5

self-checks: #10-17

exercises: #9, 11, 14, 16

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:

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

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

```
Point p = new Point(3, 8);    // better!
```

  - We are able to this with most types of objects in Java.

Constructors

- **constructor**: Initializes the state of new objects.

```
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.

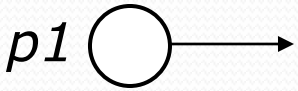
Constructor example

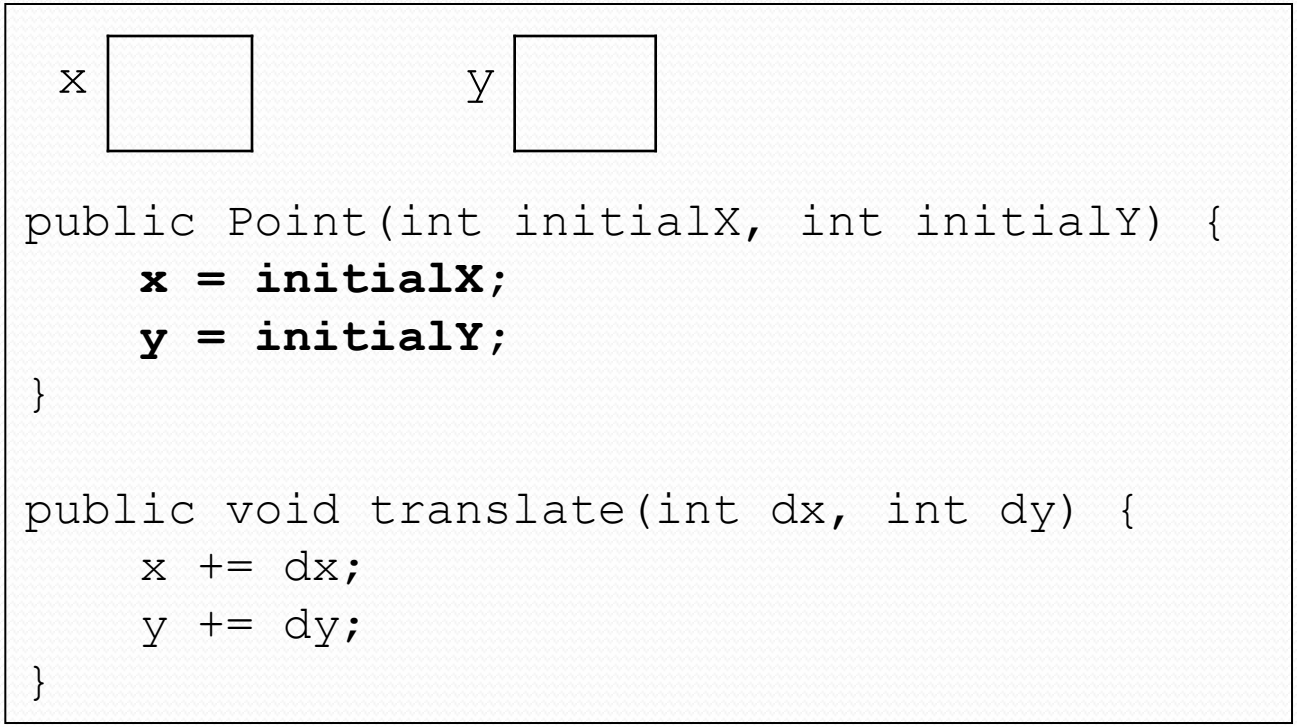
```
public class Point {
    int x;
    int y;

    // 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?

```
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;
}
```

Client code, version 3

```
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`:

```
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"):

```
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

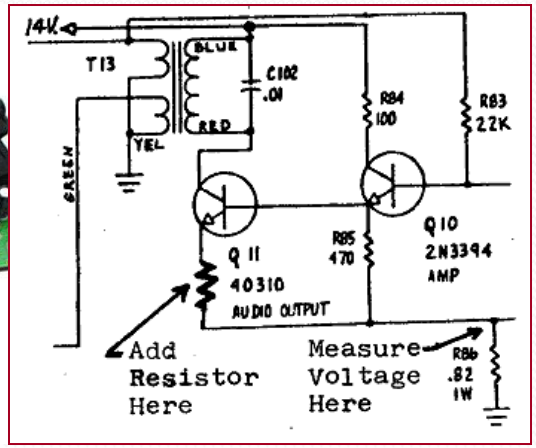
reading: 8.5 - 8.6

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:

```
// 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, 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 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, version 4

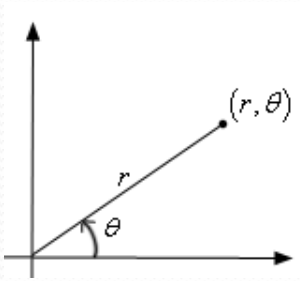
```
public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // 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 *θ*), but with the same methods.



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

Building Java Programs

Chapter 8

Lecture 8-3: `toString`, `this`

reading: 8.6 - 8.7

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:

```
Point p = new Point(10, 7);
System.out.println("p: " + p);  // p: Point@9e8c34
```

- We can print a better string (but this is cumbersome):

```
System.out.println("p: (" + p.x + ", " + p.y + ")");
```

- We'd like to be able to print the object itself:

```
// desired behavior
System.out.println("p: " + p);  // p: (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`:

```
Point p1 = new Point(7, 2);
System.out.println("p1: " + p1);
```

  - If you prefer, you can write `.toString()` explicitly.

```
System.out.println("p1: " + 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:

```
Point@9e8c34
```

`toString` syntax

```
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 + ")";
}
```

Client code

```
// This client program uses the Point class.
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: " + p1.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));
    }
}
```

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.

```
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.

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.

```
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`:

```
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;
    }

    ...
}
```