

# Building Java Programs

## Chapter 8

### Lecture 8-3: Constructors; Encapsulation

**reading: 8.3 - 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:

```
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
```

# 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 + ")";  
}
```

# 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 specify the fields' initial values at the start:

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

- We are able to do 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
- no return type is specified;  
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.



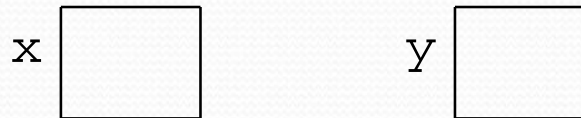
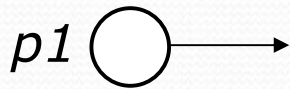
# Constructor example

```
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 = x + dx;  
        y = 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)
```

# Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.
- *Exercise:* Write a `Point` constructor with no parameters that initializes the point to (0, 0).

```
// Constructs a new point at (0, 0).  
public Point() {  
    x = 0;  
    y = 0;  
}
```

# Common constructor bugs

## 1. Re-declaring fields as local variables ("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.

## 2. Accidentally giving the constructor a return type:

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

- This is actually not a constructor, but a method named `Point`

# Encapsulation

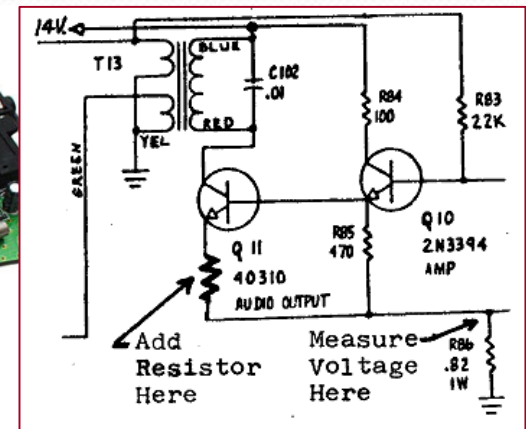
**reading: 8.5 - 8.6**

self-check: #13-17

exercises: #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;
    }

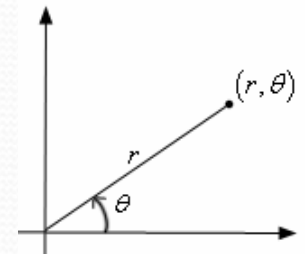
    public int getY() {
        return y;
    }

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

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

# 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  $(r, \theta)$  with the same methods.
- Can constrain objects' state (**invariants**)
  - Example: Only allow `Accounts` with non-negative balance.
  - Example: Only allow `Dates` 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