

# Building Java Programs

Chapter 8

Lecture 21: toString, constructors, encapsulation

**reading: 8.1 - 8.2**

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)



# Object initialization: constructors

**reading: 8.3**

# 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); // desired; doesn't work (yet)
```

- 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;
    }

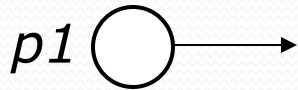
    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }

    ...
}
```

# 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;  
}
```

# 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`



# 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;  
}
```

# 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 + " )";  
}
```

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`

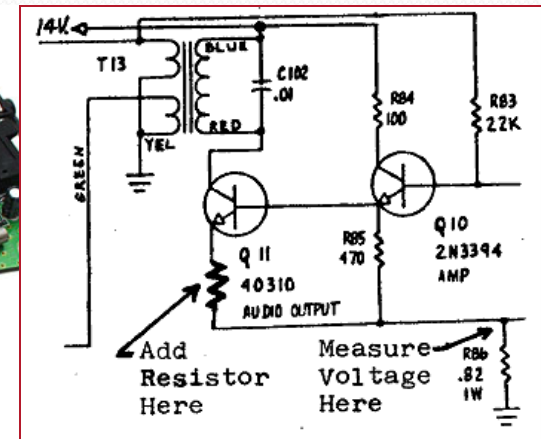
- One constructor can call another using `this`:

```
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;  
    }  
  
    ...  
}
```



# 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  $\theta$ ), but with the same methods.
- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Points` with non-negative coordinates.

