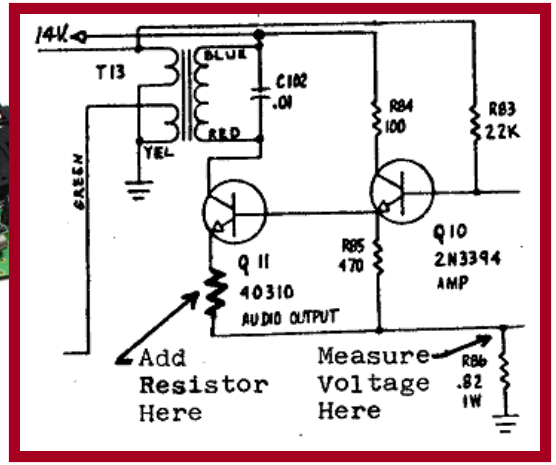
Lecture 12: Classes II

Building Java Programs: A Back to Basics Approach
by Stuart Reges and Marty Stepp

Encapsulation

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;
}

// access to modify the x field (”mutator")
// often the use of if conditional
// limits this ability(for example only allow
// positive x values.)
public void setX(int newX) {
    x = newX;
}
```

– Client code will look more like this:

```
System.out.println(p1.getX());
p1.setX(10);
```

Point class

```
// 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;
    }
    // accessor methods
    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);
    }
}
```

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

OUTPUT:

```
p1:(5,2)
p2:(4,3)
```

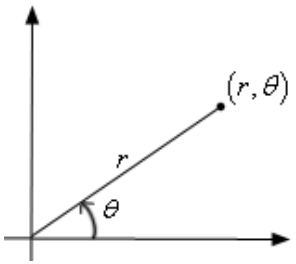
Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.

- 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.

- Can change the class implementation later.
  - Example: `Point` could be rewritten in polar coordinates (r, θ) with the same methods.



Point class

Can change the class implementation later. `Point` can be rewritten in polar coordinates (r, θ) with the same methods without affecting the client's use of the class.

```
public class Point {
    private double r;
    private double angle;

     // accessor methods
    public int getX() {
        return r*Math.cos(angle);
    }

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

..

}
```

Client code

The client code is unchanged even if the Point class is implemented using polar coordinates, a benefit of encapsulation.

```
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()+")");

}
}
```

OUTPUT:

```
p1:(5,2)
p2:(4,3)
```

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

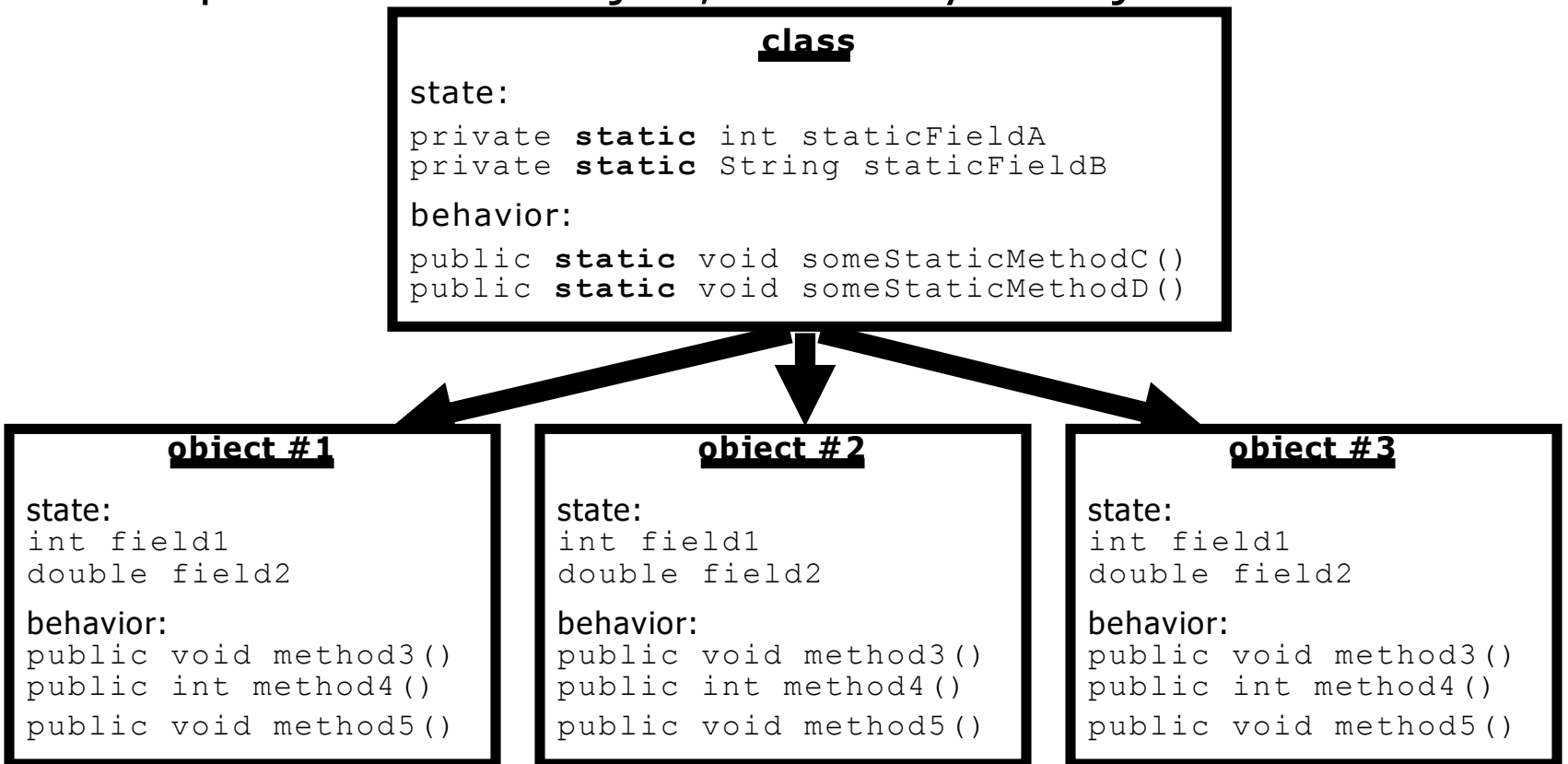
```
public class Point {
    private int x;
    private int y;

    ...

    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`

Static methods/fields

Static members

- **instance**: Part of an object, rather than shared by the class.
- **static**: Part of a class, rather than part of an object.
  - Object classes can have static methods *and fields*.
  - Not copied into each object; shared by all objects of that class.



Static fields

```
private static type name;
```

or,

```
private static type name = value;
```

- Example:

```
private static int theAnswer = 42;
```

- **static field**: Stored in the class instead of each object.
  - A "shared" global field that all objects can access and modify.
  - Like a class constant, except that its value can be changed.

Final Static fields

```
public static final type name;
```

or,

```
public static final type name = value;
```

- Example:

```
public static final int NUMOFMONTHS = 12;
```

- **Final static field**:
  - A class constant whose value cannot be changed. Usually public.
  - ALL CAPS by convention.

Modules in Java libraries

```
// Java's built in Math class is a module, only static
// variables and methods.

public class Math {

    public static final double PI = 3.14159265358979323846;

    ...

    public static int abs(int a) {
        if (a >= 0) {
            return a;
        } else {
            return -a;
        }
    }

    public static double toDegrees(double radians) {
        return radians * 180 / PI;
    }
}
```

Accessing static fields

- From inside the class where the field was declared:

```
fieldName                          // get the value
fieldName = value;                 // set the value
```

- From another class (if the field is `public`):

```
ClassName.fieldName                // get the value
ClassName.fieldName = value;       // set the value
```

  - generally static fields are not `public` unless they are `final`

Examples

```
public class PointMain {
    public static void main(String[] args) {
      int a = Math.abs(-4);
      //abs() is static

      String b = “hello”;
      int c = b.length();
      //length() is instance

    }
}
```

BankAccount

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;      // advance the id, and
        id = objectCount;   // give number to account
    }

    ...

    public int getID() {    // return this account's id
        return id;
    }
}
```

Static methods

```
// the same syntax you've already used for
//methods
public static type name(parameters) {
    statements;
}
```

- **static method**: Stored in a class, not in an object.
  - Shared by all objects of the class, not replicated.
  - Does not have any *implicit parameter*, `this`; therefore, cannot access any particular object's fields.

- **Exercise: Make it so that clients can find out how many total `BankAccount` objects have ever been created.**

BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;      // advance the id, and
        id = objectCount;   // give number to account
    }
    // clients can call this to find out # accounts created
    public static int getNumAccounts() {
        return objectCount;
    }

    ...
    public int getID() {    // return this account's id
        return id;
    }
}
```

Printing objects

- By default, Java doesn't know how to print objects:

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


// better, but cumbersome;              p is (10, 7)
System.out.println("p is (" + p.getX() + ", " +
  p.getY() + ")");


// 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

`toString` can be overwritten to return a desired String representation of the object.

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

Point class

```
// 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;
    }
    // accessor methods
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

   public String toString() {
      return "(" + x + ", " + y + ")";
    }

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

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

Client code

```
public class PointMain {
    public static void main(String[] args) {

        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        System.out.println("p1: ” + p1);
         //same as above
        System.out.println(”p2: ” + p2.toString());



}
}
```

OUTPUT:

```
p1:(5,2)
p2:(4,3)
```

Summary of Java classes

- A class is used for any of the following in a large program:
  - a *program* : Has a main and perhaps other static methods.
    - example: `GuessingGame`, `Birthday`, `MadLibs`,
    - does not usually declare any static fields (except `final`)
  - an *object class* : Defines a new type of objects.
    - example: `Point`, `BankAccount`, `Date`, `Car`, `TetrisPiece`
    - declares object fields, constructor(s), and methods
    - might declare static fields or methods, but these are less of a focus
    - should be encapsulated (all fields and static fields `private`)
  - a *module* : Utility code implemented as static methods.
    - example: `Math`

Lab 1

Write Student class is below.

```
public class Student{
  private String name;
  private double average;
  public Student(String a, double ave){…}
  public String getName(){…}
  public double getAverage(){…}
}
```

In the driver class:

1) Create an array of 3 Student objects in the driver class. Populate the array with some Student objects.

2) Print the name of the first student on the list.

Lab 1

Write the static method in the Student class called `highestAverage` which accepts an array of Students returns the highest average of all of the students.

```
public static double highestAverage(Student[]
  array)
  …
}
```

This is a popular variant of many free response AP questions!

Lab 1

Here's another popular variant to the last question.

Write the static method `bestStudent` in the Student class which accepts an array of Students returns the name of the student with the highest average. If there are more than one student with the highest average, returns the first in the list.

```
public static String bestStudent(Student[] array){
  …
}
```

Lab 2

Write a class called `MyComplex` which models the complex numbers a+bi. It contains:

1)Two private double variables `real` and `img`.

2)A default constructor to create a complex number at 0+0i.

3)A constructor which takes two double a and b and initializes this complex number to a+bi.

4)Setters and Getters(accessors and mutators) for private variables real and img.

5)`toString()` that returns "(a+bi") form of the complex number.

6)`isReal()` and `isImaginary()` that returns whether this complex number is real or imaginary, respectively. For example, 4 is real while 5i is imaginary but 4+5i is neither and 0 is both.

Lab 2

6) `void add(MyComplex c)`: Add complex number c to this complex number. Hint: (a+bi)+(c+di)=(a+c)+(b+d)i

7) `void multiply(MyComplex c)`: Multiply c to this complex number. Hint: (a+bi)*(c+di)=(ac-bd)+(ad+bc)i

8) `void conjugate()` changes this complex number into its conjugate. Hint: The conjugate of a+bi is a-bi.

Lab 2

9) `double argument()` returns the argument(angle) in radians of this complex number. This angle is the same as theta of polar coordinates. Hint: Use Math.atan2(y,x).

10) `double magnitude()` returns the magnitude or length of this complex number. Hint: The magnitude of a+bi is Math.sqrt(a*a+b*b). For example, the magnitude of 3+4i is 5.

Lab 2

MyComplex should also contains the following static methods:

1)`MyComplex addNew(MyComplex a, MyComplex b)` returns a complex number that is the sum of a and b.

2)`MyComplex multiplyNew(MyComplex a, MyComplex b)` returns the complex number that is the product a*b.

Lab

Write a driver class ComplexTester to test all of the methods/constructors of MyComplex.