Chapter 9

Class Members

An individual object has attributes that define the object's state. Some properties of objects are the same for all instances of its class. These common properties should not be stored in individual objects, but should be kept in a central location and shared by all instances of the class. This notion of class attributes generalizes to class methods, methods that need no instance-specific information to do their work.

9.1 Class Variables

`RYGTrafficLight` objects are automatically made red when created. It might be nice to have a constructor that accepts a color so the client can create a traffic light object of the desired color right away without having to call `change()`. Ideally

- The client should use symbolic constants like `RED` and `GREEN` instead of numbers like 1 and 2.
- The traffic light class author should control these constants so she has the freedom to change their values if necessary. For example, if the traffic light class author changes RED to be 2 instead of 1, clients should be oblivious, and the clients' code should continue to work without modification.
- The traffic light class author would make these constants available to clients.

If first glance, the solution seems simple enough—make the constants `RED`, `YELLOW`, and `GREEN`. in the `RYGTrafficLight` class public instead of private:

```
public final int RED    = 1;
public final int GREEN  = 2;
public final int YELLOW = 3;
```

Based on these definitions:

- The elements are constants (declared `final`), so clients cannot modify them.
- The elements are declared `public`, so clients can freely see their values.

Now we can overload the constructor providing a new one that accepts an integer argument:


```
public RYGTrafficLight(int initialColor) {
    if (initialColor >= RED && initialColor <= GREEN) {
        color = initialColor;
    } else {
        color = RED;  //  defaults to red
    }
}
```

and a client can use this constructor along with the color constants. Notice that the client can supply any integer value, but the constructor ensures that only valid color values are accepted. Arbitrary values outside the range of valid colors make a red traffic light.

So what is stopping us from using these constants this way? What happens when the client wants to create a traffic light that is initially green? If we write the code fragment that creates a green traffic light

```
RYGTrafficLight light = new RYGTrafficLight(???.GREEN);
```

what goes in the place of the `???` symbols? We need to replace the `???` with a `RYGTrafficLight` instance. As it now stands, we must have an object to access the `RED`, `YELLOW`, and `GREEN` constants. The problem is, the constants belong to `RYGTrafficLight` objects, and we cannot access them until we have at least one `RYGTrafficLight` object! This ultimately means we cannot create a traffic light object until we have created at least one traffic light object!

There is another problem with the color constants being members of each traffic light object. Every traffic light object needs to keep track of its own color, so `color` is an instance variable. There is no reason, however, why each traffic light needs to store its own copy of these three constant values. Suppose, for example, we were developing a program to model traffic flow though a section of a big city, and our system had to simultaneously manage 1,000 traffic light objects. The memory required to store these color constants in 1,000 objects would be

$$1{,}000 \text{ objects} \times 3 \frac{\texttt{int}\text{s}}{\text{object}} \times 4 \frac{\text{bytes}}{\texttt{int}} = 12{,}000 \text{ bytes}$$

This is unnecessary since these constants have the same value for every traffic light object. It would be convenient if these constants could be stored in one place and *shared* by all traffic light objects.

In Java, a class is more than a template or pattern for building objects. A class is itself an object-like entity. A class can hold data. This class data is shared among all objects of that class. The reserved word `static` is used to signify that a field is a *class variable* instead of an instance variable. Adding the `final` qualifier further makes it a *class constant*.

The option of class fields solves both of our problems with the color constants of the traffic light class. Declaring the constants this way:

```
public static final int RED    = 1;
public static final int GREEN  = 2;
public static final int YELLOW = 3;
```

(note the addition of the reserved word `static`) means that they can be accessed via the class itself, as in

```
RYGTrafficLight light = new RYGTrafficLight(RYGTrafficLight.GREEN);
```

The class name `RYGTrafficLight` is used to access the `GREEN` field instead of an instance name. If `GREEN` were not declared `static`, the expression `RYGTrafficLight.GREEN` would be illegal.

If we create 1,000 traffic light objects, they all share the three constants, so

$$3 \frac{\texttt{int}\text{s}}{\text{class}} \times 4 \frac{\text{bytes}}{\texttt{int}} = 12 \text{ bytes}$$

only 12 bytes of storage is used to store these constants no matter how many traffic light objects we create.

Given the declaration

```
public static final int RED    = 1;
```

we see, for example, `RED` is

- readable by any code anywhere (`public`),
- shared by all `RYGTrafficLight` objects (`static`),
- a constant (`final`),
- an integer (`int`), and
- has the value one (`= 1`).

We say that `RED` is a class constant. The following interactive session shows how the class fields can be used:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> RYGTrafficLight.RED
1
> RYGTrafficLight.YELLOW
2
> RYGTrafficLight.GREEN
3
```

Observe how we created no objects during the interactive session above (the `new` operator was not used). It is not necessary to create an object to access class fields.

Clients should be conscientious and use publicly defined class constants instead of literal values. Class designers provide public class constants as part of the contract they make with clients. Assuming our `RYGTrafficLight` class has been modified with the class constants described above, the following client code:

```
RYGTrafficLight light = new RYGTrafficLight(2);
```

makes a yellow traffic light object. The symbol `YELLOW` is guaranteed to stand for the yellow color, but `YELLOW`'s actual value may change. If the maintainer of `RYGTrafficLight` updated the class by adding some colors (like black, for no lamps lit), `YELLOW` might be 3 instead of 2. The above code might then make a red light instead of the intended yellow light! The client's code is now broken, but it is the client's fault for using a literal constant (2) instead of the symbolic constant (`RYGTrafficLight.YELLOW`). If the statement above were part of a large complex program with many more similar statements throughout, the task to repair the code could be enormous. Just because you can see the value of a constant:

```
System.out.println(RYGTrafficLight.YELLOW);
```

does not mean it is wise to use its literal value.

9.2 Class Methods

Like fields, methods can be declared `static`. As with fields, this means the methods are *class methods*. All the methods we have seen so far have been *instance methods*. Instance methods must be called on behalf of an object, but a class method may be called on behalf of the class itself, and no instance is needed. In the `RYGTrafficLight` class, the `setColor()` method is an instance method, so

```
RYGTrafficLight t = new RYGTrafficLight(RYGTrafficLight.GREEN);
t.setColor(RYGTrafficLight.RED);  //  Legal
```

is valid Java, but

```
//  Illegal!  Compile-time error
RYGTrafficLight.setColor(RYGTrafficLight.RED);
```

is not. Since `setColor()` is an instance method, it cannot be called on behalf of the class; an object is required. Why is an object required? The code within `setColor()` modifies the `color` instance variable. Instance variables are not stored in the class; they are stored in objects (instances). You must call `setColor()` on behalf of an object, because the method needs to know which `color` variable (that is, the `color` variable in which object) to change.

While most of the methods we encounter are instance methods, class methods crop up from time to time. The `gcd()` method in `RationalNumber` (7.5) would be implemented better as a class method. `gcd()` gets all the information it needs from its parameters; it works the same regardless of the `RationalNumber` object with which it is called. No instance variables are used in `gcd()`. Any method that works independently of all instance information should be a class (`static`) method. The `gcd()` would better be written:

```
private static int gcd(int m, int n) {
    . . .
```

The JVM executes class methods more quickly than equivalent instance methods. Also, a class method cannot directly alter the state of the object upon which it is called since it has no access to the instance variables of that object. This restriction can actually simplify the development process since if an object's state becomes messed up (for example, a traffic light switching from red to yellow instead of red to green), a class method cannot be directly responsible for the ill-defined state. A developer would limit his search for the problem to instance methods.

A method that does not use any instance variables does not need to be an instance method. In fact, since a class method can be called on behalf of the class, it is illegal for a class method to attempt to access an instance variable or instance constant. It is also illegal for a class method to make an unqualified call to an instance method within the same class (because it could be accessing instance information indirectly). However, instance methods can freely access class variables and call class methods within the class.

9.3 Updated Rational Number Class

Armed with our knowledge of class fields and class methods, we can now enhance `RationalNumber` (7.5) to take advantage of these features. `Rational` (9.1) is our final version of a type representing mathematical rational numbers.

```
public class Rational {
```



```
    private int numerator;
    private int denominator;

    //  Some convenient constants
    public static final Rational ZERO = new Rational(0, 1);
    public static final Rational ONE = new Rational(1, 1);

    //  num is the numerator of the new fraction
    //  den is the denominator of the new fraction
    public Rational(int num, int den) {
        if (den != 0) {  //  Legal fraction
            numerator = num;
            denominator = den;
        } else {  //  Undefined fraction changed to zero
            System.out.println("*** Notice: Attempt to create an "
                               + "undefined fraction ***");
            numerator = 0;
            denominator = 1;
        }
    }

    //  Returns the value of the numerator
    public int getNumerator() {
        return numerator;
    }

    //  Returns the value of the denominator
    public int getDenominator() {
        return denominator;
    }

    //  Provide a human-readable string for the fraction, like "1/2"
    public String show() {
        return numerator + "/" + denominator;
    }

    //  Compute the greatest common divisor for two integers m and n
    //  Uses Euclid's algorithm, circa 300 B.C.
    private static int gcd(int m, int n) {
        if (n == 0) {
            return m;
        } else {
            return gcd(n, m % n);
        }
    }

    //  Returns a new Rational consisting of the current
    //  fraction reduced to lowest terms
    public Rational reduce() {
        int factor = gcd(numerator, denominator);
```



```
        return new Rational(numerator/factor,
                            denominator/factor);
    }

    //  Returns a new Rational consisting of the sum of this
    //  fraction and another fraction (other).  The result is reduced
    //  to lowest terms.
    public Rational add(Rational other) {
        int num1 = numerator * other.denominator,
            num2 = other.numerator * denominator;
        return new Rational(num1 + num2,
                            denominator * other.denominator).reduce();
    }

    //  Returns a new Rational consisting of the product of this
    //  fraction and another fraction (other).  The result is reduced
    //  to lowest terms.
    public Rational multiply(Rational other) {
        int num = numerator * other.numerator,
            den = other.denominator * denominator;
        return new Rational(num, den).reduce();
    }
}
```

Listing 9.1: `Rational`—updated version of the `RationalNumber` class

The list of enhancements consist of:

- The class

  ```
  public class Rational {
  ```

  is declared `public` which means it is meant for widespread use in many applications.

- The instance variables

  ```
  private int numerator;
  private int denominator;
  ```

  are hidden from clients. Since none of the methods modify these instance variables, a `Rational` object is effectively *immutable*. (A Java `String` is another example of an immutable object.) The only way a client can influence the values of `numerator` and `denominator` in a particular fraction object is when it is created. After creation its state is fixed.

  Note that even though the object may be immutable, an object reference is a variable and can be changed:

  Interactions

  ```
  Welcome to DrJava.  Working directory is /Users/rick/java
  > Rational r = new Rational(1, 2);
  > r.show()
  ```



```
"1/2"
> r = new Rational(2, 3);
> r.show()
"2/3"
```

Did the object `r` which was originally $\frac{1}{2}$ change to $\frac{2}{3}$? No, `Rational` objects are immutable and, therefore, cannot change. The object *reference* `r` was reassigned from pointing to object $\frac{1}{2}$ to the new object $\frac{2}{3}$. The original object $\frac{1}{2}$ is unchanged.

- The constructor now prints to the console a warning when a client attempts to create a `Rational` object with a `denominator` of zero.
- It is reasonable to allow clients to see the instance variables:

  ```
  public int getNumerator() { . . .
  public int getDenominator() { . . .
  ```

  These methods return copies of the values of the instance variables; the instance variables themselves in a `Rational` object are safe from client tampering.
- Since zero (identity element for addition) and one (identity element for multiplication) are commonly used numbers, they have been provided as class constants:

  ```
  public static final Rational ZERO = new Rational(0, 1);
  public static final Rational ONE = new Rational(1, 1);
  ```

  Based on their definition:
  - `public`—they are freely available to clients
  - `static`—they are class members and, therefore,
    - they can be used even without creating a `Rational` object, and
    - only one copy of each exists even if many `Rational` objects are created.
  - `final`—they are constants and cannot be changed; for example, the `ONE` reference cannot be changed to refer to another `Rational` object (like $\frac{1}{2}$).
  - `Rational`—they are references to `Rational` objects and so have all the functionality of, and can be treated just like, any other `Rational` object. For example,

    Interactions
    ```
    Welcome to DrJava.  Working directory is /Users/rick/java
    > Rational r = new Rational(1, 2);
    > r = r.add(Rational.ZERO);
    > System.out.println(r.show());
    1/2
    > r = r.add(Rational.ONE);
    > System.out.println(r.show());
    3/2
    > r = r.add(Rational.ONE);
    > System.out.println(r.show());
    5/2
    ```

    can be used without creating a `Rational` object

- The `gcd()` method

  ```
  private static int gcd(int m, int n) {
  ```

  is meant to be used internally by other methods within the class. The `reduce()` method uses `gcd()` to simplify a fraction. Our belief is that clients are interesting in reducing a fraction to lowest terms but are not concerned with the lower-level details of how this reduction is accomplished. We thus make `gcd()` `private` so clients cannot see or use it.

  We further observe that `gcd()` does not access any instance variables. It performs its job strictly with information passed in via parameters. Thus we declare it `static`, so it is a class method. A class method is executed by the JVM more quickly than an equivalent instance method.

- The `show()`, `reduce()`, and `add()` methods are all made public since they provide a service to clients.

- This version provides a `multiply()` method.

9.4 An Example of Class Variables

We know that each object has its *own copy* of the instance variables defined in its class, but all objects of the same class *share* the class variables defined in that class. `Widget` (9.2) shows how class variables (not class constants) and instance variables might be used within the same class:

```
public class Widget {
    //  The serial number currently available to be assigned to a new
    //  widget object.  This value is shared by all widget instances.
    private static int currentSerialNumber = 1;

    //  The serial number of this particular widget.
    private int serialNumber;

    public Widget() {
        //  Assign the currently available serial number to this
        //  newly created widget, then update the currently available
        //  serial number so it is ready for the next widget creation.
        serialNumber = currentSerialNumber++;
    }

    public void getSerialNumber() {
        System.out.println(serialNumber);
    }
}
```

Listing 9.2: `Widget`—Create widgets with unique, sequential serial numbers

The following interactive session makes some widgets and looks at their serial numbers:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> Widget w1 = new Widget(),
         w2 = new Widget(),
         w3 = new Widget(),
         w4 = new Widget(),
         w5 = new Widget();
> w1.getSerialNumber()
1
> w2.getSerialNumber()
2
> w3.getSerialNumber()
3
> w4.getSerialNumber()
4
> w5.getSerialNumber()
5
```

Exactly one copy of `currentSerialNumber` exists, but each `Widget` object has its own copy of `serialNumber`. The constructor uses this shared class variable to properly initialize the instance variable in each newly created object. Observe also that `serialNumber` is read-only; that is, the `getSerialNumber()` method can be used to read its value, but only the constructor can assign its value. Client code cannot change the `serialNumber` of an object to an arbitrary value. The `currentSerialNumber` class variable is totally inaccessible to the outside world, but it ensures that all widgets that a client creates will be numbered sequentially.

9.5 The `main()` Method

Any Java class may contain a method named `main()`, declared as

```
public static void main(String[] args) {
    /*  Appropriate statements go here . . . */
}
```

We are familiar with all the parts of this method's declaration, except the square brackets (`[]`). The square bracket notation is used with arrays which we will not investigate until Chapter 20. Fortunately, since this method is special, we safely can ignore the `args` parameter for now. In fact, the overwhelming majority of Java programs containing such a `main()` method ignore the `args` parameter.

Classes with a `main()` method can be executed directly by the Java runtime environment (JRE), sometimes called the Java interpreter (not to be confused with the DrJava interpreter). We see that `main()` in a class method (`static`), so it can be executed without creating an object first. `VerySimpleJavaProgram` (9.3) shows how `main()` works:

```
public class VerySimpleJavaProgram {
    public static void main(String[] args) {
        System.out.println("This is very simple Java program!");
    }
}
```


Listing 9.3: `VerySimpleJavaProgram`—a very simple Java program

Once compiled, this code can be executed in the DrJava interpreter as:

Interactions

```
java VerySimpleJavaProgram
```

or similarly from a command shell in the operating system as

```
java VerySimpleJavaProgram
```

or by pressing the Run button in the DrJava button bar.

All of the parts of the above `main()` declaration must be present for things to work:

`public`	It must be accessible to the Java interpreter to be executed. Since the Java interpreter code is not part of our class, `main()` in our class must be public.
`static`	We as users must be able to run the program without having an object of that class available; hence, `main()` is a class (`static`) method.
`void`	Since the interpreter does not use any value that our `main()` might return, `main()` is required to return nothing (`void`).
`String[] args`	This allows extra information to be passed to the program when the user runs it. We have no need for this feature here for now, but we consider it later (§ 21.4).

9.6 Summary

- Class members (variables and methods) are specified with the `static` qualifier.
- A class variable is shared by all instances of that class.
- Class variables are stored in classes; instance variables are stored in instances (objects)
- Class methods may not access instance variables.
- Class methods may access class variables.
- Class methods may not make unqualified calls to instance methods.
- An instance method may be called only on behalf of an instance.
- A class method can be called on behalf of a class or on behalf of an instance.

9.7 Exercises

1. What reserved word denotes a class field or class method?

2. In the original version of `RYGTrafficLight` (7.4) (where the constants were instance fields, not class fields) if a client created 1,000 `RYGTrafficLight` objects, how much memory in bytes would the data (`color`, `RED`, `YELLOW`, `GREEN`) for all those objects require?

3. In the new version of `RYGTrafficLight` (7.4) (described in this chapter where the constants were class fields, not instance fields) if a client created 1,000 new style `RYGTrafficLight` objects, how much memory in bytes would the data (`color`, `RED`, `YELLOW`, `GREEN`) for all those objects require?

4. Given the class

```
public class Widget {
    public final int VALUE = 100;
    public Widget(int v) {
        System.out.println(v);
    }
}
```

   does the following client code work?

```
Widget w = new Widget(Widget.VALUE);
```

   If it works, what does it do? If it does not work, why does it not work?

5. Why is it not possible to make `color` a class variable in `RYGTrafficLight` (7.4)?

6. What determines whether a method should be a class method or an instance method?

7. If clients can see the values of public class constants, why should clients avoid using the constants' literal values?

8. Enhance `Rational` (9.1) by adding operations for:

   - subtraction
   - division
   - computing the reciprocal

9. Enahance `Rational` (9.1) show that the `show()` method properly displays mixed numbers as illustrated by the following interactive sequence:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> Rational r = new Rational(5, 2);
> System.out.println(r.show());
2 1/2
> r = new Rational(4, 1);
> System.out.println(r.show());
4
> r = new Rational(11, 4);
> System.out.println(r.show());
2 3/4
> r = new Rational(1, 4);
> System.out.println(r.show());
1/4
```


10. Next item . . .

11. Consider the following class definition:

```
public class ClassDef {
    static public double value;
    static public int number;
    public double quantity;
    public int amount;
```

    Suppose 10,000 instances of `ClassDef` were created. How much memory would be consumed by the variables held by all the objects?

12. What is the exact structure of the `main()` method that allows a class to be executed as a program.

13. Why must `main()` be declared `static`?

14. Create a class named `Point` which can be used to create mathematical point objects. Such objects:

    (a) have (x,y) coordinates which are `double`s

    (b) have a `distance()` method to compute the distance to another point.

    (c)

15. Augment the `RationalNumber` class so that `RationalNumber` objects can:

    (a) multiply

    (b) subtract

    (c) etc.

16. We have seen how `reduce()` (an instance method) in `Rational` calls `gcd()` (a class method) to assist its work. What happens if `gcd()` calls an instance method of `Rational`? Explain what happens.

