Classes to Organize Code

Programs can become very large and complex. Much of the software you are used to using includes tens of millions of lines of code – Windows, MacOs, Microsoft Office and WPS Office, Chrome, Edge, Safari. Even the average Android or Apple phone application is 50,000 to 100,000 lines of code – even without the libraries, like the UI frameworks, that they use.

Considering this, it is extremely important to be able to organize code. Although there are different programming styles, such as *functional*, *procedural*, and *object-oriented*, all separate code based on its functionality. For *object-oriented* programming, as Java programs are, code that performs a particular function is separated into into its own ***class***.

In Java, each `public class` must be contained within it's own file, and the file name of that file must match the name of the class, followed with a "`.java`" extension. Eclipse creates this file automatically when you ask it to create a new class.

Java also provides some standard libraries. One class that we've used before is the Java `Math` class. Look at example code which, although not exactly the same as what is in the real `Math` class, implements a portion of it, and is functionally equivalent.

***Code Block 1**: Code implementing functionality of the Java `Math` class*

```
1  public class Math {
2     public static final double PI = 3.14159265358979323846;
3     public static double abs(double a) {
4        if(a >= 0) {
5           return a;
6        } else {
7           return -a;
8        }
9     }
10 }
```

A ***constant*** is a value that, once set, cannot change during the execution of the program. In Java, a *constant* is defined and initialized in the same way a *variable* is, except the keyword `final` is added before the type of the data. Line `2` of the code above defines a value for π, assigning it to a *constant* of type `double` that has the identifier `PI`.

When a *constant* or a *variable* is defined at the top level within a class, it is called a ***field*** (or sometimes an ***attribute***).

We can access a `static` *field* that is in a different class than the current class by using the class identifier, a period (`.`), then the *field* identifier. For example, we can use the value of the `PI` constant from within a class other than the `Math` class by typing `Math.PI` in the code. *Code Block 2* shows an example of this.

Line `3` of *Code Block 1* defines a `static` method named `abs` that takes a single parameter of type `double`, and returns a value that is also of type `double`. We call this method from a class other than the `Math` class in a similar way: with the class name, a period, then the method identifier and the parameters required by the parameter list in the method signature. So to call the `abs` method

and pass a `double` value of `-5.25` to it, we would type: `Math.abs(-5.25)`. Again, *Code Block 2* shows an example of this (using a variable named `d` of type `double` to store the value passed.

***Code Block 2**: Code using the `Math` class, and corresponding output*

```
1 public class UsingMath {
2    public static void main(String[] args) {
3       double d = -5.25;
4       System.out.println("The value of pi is: " + Math.PI);
5       System.out.println("Absolute value of " + d +
6                          " is " + Math.abs(d));
7    }
8 }
```

```
Absolute value of -5.25 is 5.25
The value of pi is: 3.141592653589793
```

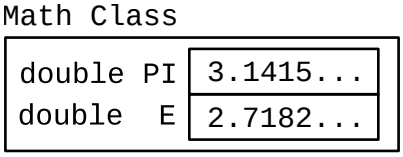
You should already have a solid understanding of how the code from *Code Block 2* works. Ensure you have studied *Code Block 1: Code implementing functionality of the Java Math class* sufficiently to understand how a Java class is written so that you can organize your own code by separating different functionality into different classes.

Static Versus Non-Static

In the previous section, we have discussed `static` fields and methods. Shortly, we will discuss non-`static` fields and methods. For `static` fields, there is one value stored in memory for each class. For example, the `Math` class need only have a single constant named `PI` that can be used by any other code. There is no need to create multiple instances of the `Math` class in order to have multiple values for `PI`.

However, the `String` class is different. We will surely want more than one value of string in almost every program we write. The `String` class defines the data structures and methods that operate on ***instances*** of the `String` class. How this is implemented in Java will hopefully be clear to you soon.

The following shows diagrammatically how the data for the `Math` class and for two instances of the `String` class are stored.

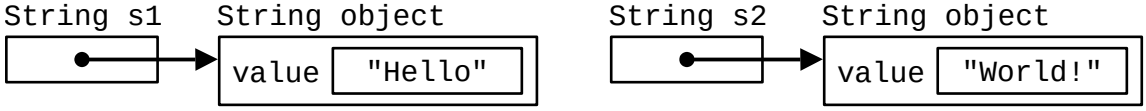


For the `Math` class, the `static` values are associated with the class. We do not create any *instances* of the Math class.

You may recall how we previously created instances of the String class with the code similar to:

```
String s1 = new String("Hello");
String s2 = new String("World!");
```

For the `String` class, *instances* of type `String` are created, and each instance stores its own data values. When *fields* are non-`static`, each ***instance*** of the class stores a different value.



With static fields, such as `Math.PI`, we use the class identifier to reference it. There is no ambiguity since there is only one value for the class. However, if we were to try to use the class identifier when referring to non-`static` fields, such as using `String.value` when referencing to a string object, how would the compiler know whether you are referring to the value "`Hello`" in the `s1` `String` object or the value "`World!`" in the `s2` `String` object?

To ensure there is no ambiguity, we must use the variable identifier (not the class identifier) when we refer to non-`static` fields and methods of an object. For example, we cannot use `String.length()`, that will return an error. We must use `s1.length()` to return the value of `5`, the length of the string "`Hello`", and `s2.length()` to return a value of `6`, the length of the string "`World!`".

This concept is fundamental, yet perhaps difficult, and it will hopefully be more clear as we go through more examples.

Using Classes to Define Complex Data Structures

Imagine we are writing some physics software that will manipulate vectors. One early requirement is that we are able to add two vectors. So let's consider how this might work. You start thinking of how you might write this and soon you find an inconvenience. Examine the code below and find the error. Then consider if you know of any easy and elegant way to solve the problem.

***Code Block 3**: First draft of method `addVector()` – contains an error!*

```
 1 public class TestVector {
 2    public static void main(String[] args) {
 3       double x1 = 5;
 4       double y1 = 4;
 5       double x2 = -1;
 6       double y2 = -1;
 7       double v = addVector(x1, y1, x2, y2);
 8       System.out.println("Vector value: " + v);
 9    }
10    public static double addVector(double x1, double y1,
11                                   double x2, double y2) {
12       double x3 = x1 + x2;
13       double y3 = y1 + y2;
14       return x3, y3;
15    }
16 }
```

Introduction to Classes: Organizing Code

In Java, we cannot have multiple return values from a function. This makes it very inconvenient to deal with data structures that are more than just a single primitive type. There are different solutions possible, but one solution is ***object-oriented programming***. The first step is to create an ***object*** that can store data that belongs logically together in one place. We start by defining the structure of the object we wish to create in a `class`.

In the above example, our data structure is a vector, and a two-dimensional vector consists of an x component and a y component. Each of the x and y components can be stored in their own variable of type `double`. The code below is how we might use a class named `Vector` to define what data a `Vector` object contains.

***Code Block 4**: First draft of the `Vector` class*

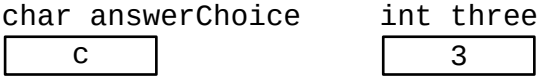
```
1 public class Vector {
2    public double x;
3    public double y;
4 }
```

The variables `x` and `y` are defined immediately inside the class at the top level (i.e.: not inside a method). These top-level variables defined in a class are called ***fields***.

Recall that we declare an initialize primitive type variables with statements such as these.

```
1 char answerChoice = 'c';
2 int three = 3;
```

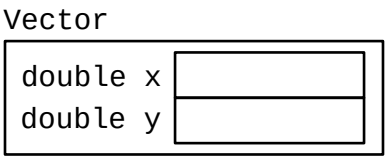
Memory is allocated and the value is stored in the memory location referred to by the variable identifier. These could be represented diagrammatically as:



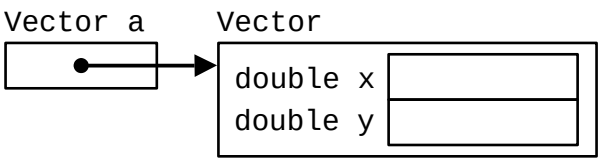
For objects, we declare and initialize an object variables with a statement such as this:

```
1 Vector a = new Vector();
```

The latter part of the line contains the keyword `new`, followed by the class name – in this case `Vector` – and a set of parentheses. This part of the code allocates the space to store the data inside the class. This data structure might be represented diagrammatically as:



The assignment (`=`) then stores the *reference* to (memory location of) that data into a variable with identifier `a`. The final data structure is shown diagrammatically below.



Notice that while primitive types store their value directly in the memory location given by the variable identifier, object variables only store a ***reference*** to (the location of) the data. The actual data is stored in a separate area of memory. The reason for this is that it allows all object variables to be the exact same size, no matter how big or small the actual object is.

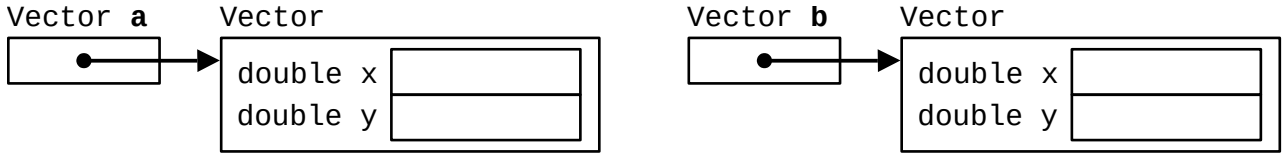
Using Public Fields From a Separate Class

Both *fields* in the Vector class in *Code Block 4* have the modifier `public`. The `public` modifier allows the fields to be accessed directly from any other class, as is shown with the following code block.

***Code Block 5**: Second draft of method `addVector()`*

```
1  public class TestVector {
2     public static void main(String[] args) {
3        Vector a = new Vector();
4        Vector b = new Vector();
5        a.x = 5;
6        a.y = 4;
7        b.x = -1;
8        b.y = -1;
9        a = addVector(a,b);
10       System.out.println("Vector value: (" +
11                          a.x +", " + a.y + ")");
12    }
13    public static Vector addVector(Vector v1, Vector v2) {
14       Vector v = new Vector();
15       v.x = v1.x + v2.x;
16       v.y = v1.y + v2.y;
17       return v;
18    }
19 }
```

Lines 3 and 4 will allocate memory for two separate Vector objects, and assign the location of one to variable `a`, and the location of the other to variable `b`. A diagrammatic representation of this is shown below.



Lines `5` through `8` show how the `public` ***instance*** *fields* within an *object* can be accessed. Be aware that we do <u>not</u> use the class identifier when accessing the data within an object, as we do to access `static` class fields, but we must use the object (variable) identifier. The object identifier, a period (`.`), and the identifier of the field to be accessed.

Line `13` contains the *method header* for `addVector`. The return type is an object of type `Vector`. This way, we are able to return a complex data structure from a method, not only a primitive type. We are also able to pass complex data types (objects) into a method in the parameter list, which can often simplify the parameter list and provide a level of abstraction.

Constructors to Initialize Objects

You might recall that when we *instantiated* a `String` object using the `new` keyword, we put the value for the string directly into the parentheses after the class name:

```
String s = new String("Hello");
```

We can also do this when we instantiate a `Vector` object by adding a ***constructor*** to our `Vector` class. A ***constructor*** is a method used to initialize the ***state*** of (the values stored in) an *instance* of the class. In Java, the identifier of every constructor must be the exact same as the identifier of the class is is in. In this case, our class identifier is `Vector`, so the constructor identifier must also be `Vector`. *Code Block 6*, below, has a constructor added to our previous version of the `Vector` class.

***Code Block 6**: Second draft of the `Vector` class*

```
1 public class Vector {
2    public double x;
3    public double y;
4    public Vector(double x, double y) {
5       this.x = x;
6       this.y = y;
7    }
8 }
```

For any `Vector` object, there are two fields: `x` and `y`, each of which is of type `double`. The constructor takes in two parameters, also named `x` and `y`, and also each of type `double`. The statement "`this.x = x;`" assigns the value passed into parameter `x` into the instance field `x`.

The "`this.x`" is required to remove the ambiguity that arises from using the same name for the instance field as for the constructor parameter. The "`this`" is not required if a different name were used for the parameter. For example the following code is also valid.

```
1 public class Vector {
2    public double x;
3    public double y;
4    public Vector(double xValue, double yValue) {
5       x = xValue;
6       y = yValue;
7    }
8 }
```

The following is updated code to use the constructor that was added to the `Vector` class. The constructor call has be highlighted using bold font.

***Code Block 7**: Test code for second draft of the `Vector` class*

```
1  public class TestVector {
2     public static void main(String[] args) {
3        Vector a = new Vector(5.0, 4.0);
4        Vector b = new Vector(-1.0, -1.0);
5        a = addVector(a,b);
6        System.out.println("Vector value: (" +
7                           a.x +", " + a.y + ")");
8     }
9     public static Vector addVector(Vector v1, Vector v2) {
10       Vector v = new Vector(0.0, 0.0);
11       v.x = v1.x + v2.x;
12       v.y = v1.y + v2.y;
13       return v;
14    }
15 }
```

If there is no constructor definition in a class, the Java compiler will add a "default constructor". The default constructor takes no parameters, and assigns a value of zero to all instance fields for any instantiated object. If a constructor is written for a class, then no default constructor will be added. Thus, although we could previously instantiate an object of type Vector using the line:

```
Vector a = new Vector();
```

After adding the constructor that requires two parameters of type `double` to the class, any attempt to call the constructor for `Vector` with no parameters (as in the line above) will result in a compile-time error.

Classes to Encapsulate Functionality

The method we have written, `addVector`, operates on two `Vector` objects. Therefore, we can only ever use it when we are manipulating objects of type `Vector`. We may also write other methods that will operate on `Vector` objects. It makes much more sense to keep all things related to manipulating `Vector` objects together in one place, and also together with the definition of what a `Vector` is. Meaning: we should move the method for adding `Vector` objects into the `Vector` class. This leads us to our next draft of the `Vector` class and associated test code.

***Code Block 8**: Third draft of the `Vector` class*

```
 1 public class Vector {
 2    public double x;
 3    public double y;
 4    public Vector(double x, double y) {
 5       this.x = x;
 6       this.y = y;
 7    }
 8    public void add(Vector v) {
 9       this.x += v.x;
10       this.y += v.y;
11    }
12 }
```

***Code Block 9**: Test code for the third draft of the `Vector` class*

```
public class TestVector {
   public static void main(String[] args) {
      Vector a = new Vector(5.0, 4.0);
      Vector b = new Vector(-1.0, -1.0);
      a.add(b);
      System.out.println("Vector value: (" +
                         a.x +", " + a.y + ")");
   }
}
```