Encapsulation and Data Hiding in Java

This is not an exhaustive treatment of encapsulation. It provides an explanation of the basic idea, since you should have already covered the idea before.

There's more to say about encapsulation, especially in the context of inheritance. We'll revisit it when we cover that topic.

Without Encapsulation

Before we look at what encapsulation and data-hiding actually are, let's take a look at a simple class.

```java
public class DogExample1 {
    public static void main(String[] args) {
        Dog shep = new Dog();

        System.out.println(shep);
    }
}

/**
 * A badly designed Dog class.
 * Why? the instance fields are public
 */
class Dog
{
    public String colour;
    public int age;

    public Dog()
    {
        this("brown", 4);
    }

    public Dog (String inColour, int inAge)
    {
        colour = inColour;
        age = inAge;
    }

    @Override
    public String toString() {

        return "Colour:[" + colour + "], Age:[" + age + "]";
    }
}
```

Running this will, of course output:

```
Colour:[brown], Age:[4]
```

Why is this poorly designed?

Hint: We can assign values to the instance fields directly, because they are `public`. For example,

```
    public static void main(String[] args) {
        Dog shep = new Dog();

        shep.age = 7; //possible because attribute is public

        System.out.println(shep);
    }
```

So, apart from the fact that we can modify attributes of objects from classes **outside** that object (and why do you think that is potentially bad?), we can assign any values we like to those instance-fields/attributes.

```
shep.age = -7; //what kind of a dog is this?
```

So, there are some obvious – and maybe not-so-obvious – issues with this:

1. The data (so-called *instance fields* or *attributes*) is not protected in any way. Code outside the class can easily put bad/wrong values into objects.

2. Depending on the context, there is also the potential for malicious data to be inserted into an object.
   Imagine having a `balance` field in a `BankAccount` object declared as `public`.

3. It can often be useful to log changes to data. Since the data can be changed in a de-centralised manner, there is no guarantee that each disparate piece of code that modifies the data will also log this.

4. If something goes wrong with our data, then we must find the source of the bug. Since any class in any package can potentially modify the data directly, it can be a nightmare to track down the offending piece of code.

What is encapsulation?

Encapsulation is often interchangeably used to mean data-hiding (or sometimes the two are conflated under the term 'encapsulation'.) In fact, definitions often differ as a result of this.

At its most basic explanation, encapsulation is one of the fundamentals of OOP (object-oriented programming). It refers to *the bundling of data with the methods that operate on that data* (as depicted in **Figure 1**).

However, it almost always also infers that the *data in the class cannot be directly accessible to code outside of the class*, as depicted in **Figure 2 :**. Effectively, this is information hiding.

And how is this achieved? Simply by **making the data private** to the class and only allowing **access to the data through public methods**, thus guarding the data from inappropriate access. Often this is through getter and setter methods, although other methods may also modify/mutate the data.

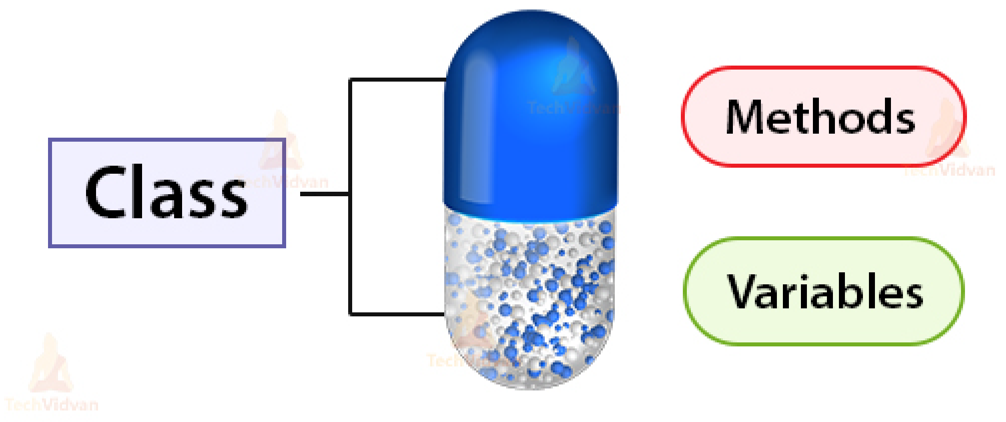


Figure 1: Encapsulation – only public methods can access/manipulate the data (variables)

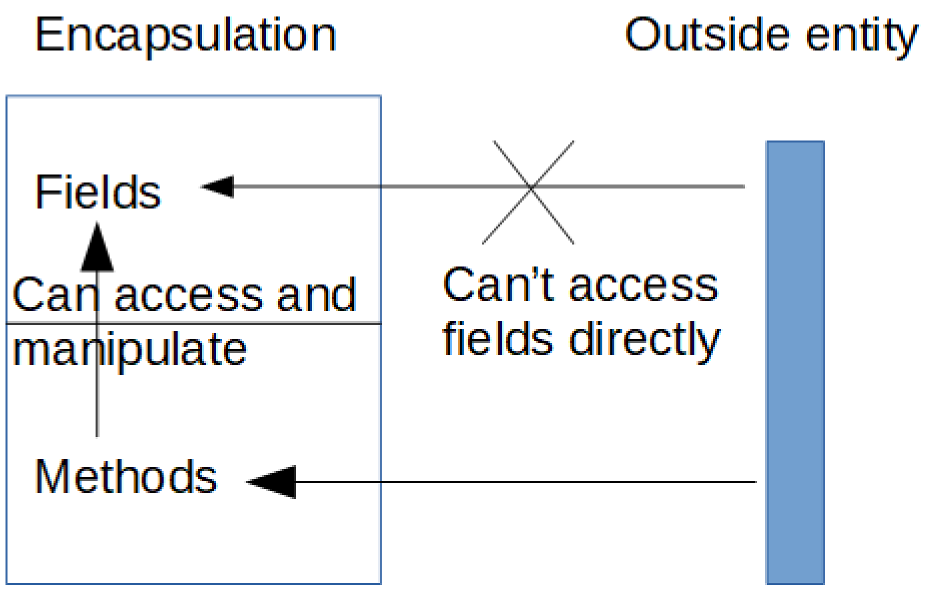


Figure 2 : A more code-like view of Encapsulation. Instance-Fields/variables/attributes/data are all interchangeable descriptors.

encapsulation
The characteristic of an object that limits access to the variables and methods contained in it.
All interaction with an object occurs through a well-defined interface that supports a modular design.

public, private and protected

As you already probably know, these are reserved words. They are known as access/visibility modifiers (because they control access to the members of a class) and can be applied to the variables and methods of a class (and even to classes themselves – for example, so-called inner classes are usually declared as `private`).

If a member of a class has ***`public`*** *visibility*, it can be directly referenced from outside of the object (remember that an *object* is an instance of a *class*).

If a member of a class has ***`private`*** *visibility*, it can be used anywhere inside the class definition but cannot be referenced externally.

You may also be aware of a third modifier – *protected* – but this is relevant in the context of inheritance, so we'll return to it later.

And just to note, *the **default** modifier is package-private*, which means that it is visible to any classes within the package, but private to class external from the package. To avoid potential confusion to yourself you should always provide access modifiers for class variables and methods – the basic rule-of-thumb is `private` for class variables and `public` for class methods and this suffices quite a lot of the time.

The table in **Figure 3** provides a handy little summary.

	public	private
Variables	Violate Encapsulation	Enforce Encapsulation
Methods	Provide services to clients	Support other methods in the class (helper methods)

Figure 3: The effects of public and private access modifiers

Exercise

```java
package com.dermot; // this is auto-generated. Yours will be different.

/**
 * A non-useful class that will allow us to play around with access-modifiers (public and private)
 *
 * Read through the class code before attempting the exercises below.
 * Try the following:
 *      1: Instantiate an instance of the class, i.e. an object, from another class (in the
 *          same package) that contains the main() method.
 *          Use system.out.println() to verify that the object has the correct values in it.
 *      2: In main() try to modify publicVar2 via the object that you created.
 *          Can you do it?
 *          print again to verify.
 *      3: In main() try to modify privateVar1 via the object that you created.
 *          Can you do it?
 *          What do you think you might do if you need privateVar1 to remain private but you would
 *          like to be able to modify it from outside the class, e.g. from main().
 *      4:  Call the printTotal() method from main() and verify that it works as expected.
 *          What happens if you change its access specifier to private?
 *      5: Try to change the access modifier of the class from 'public' to 'private'
 *          What happens? Can you speculate on why this is happening?
 *
 */
public class SomeClass {
    //these variables are sometimes called "instance-fields" or "member variables"
    //every instance (object) of our class will have their own versions/copies of these
    private int privateVar1;
    public int publicVar2;

    // This is what's known as an overloaded constructor
    public SomeClass(int val1, int val2)
    {
        privateVar1 = val1;
        publicVar2 = val2;
    }

    //I can talk briefly in class about the toString() method and
    //the @Override directive
    @Override
    public String toString() {
        return "SomeClass{" +
                "privateVar1=" + privateVar1 +
                ", publicVar2=" + publicVar2 +
                '}';
    }

    public void printTotal()
    {
        System.out.println("Var total is " + privateVar1 + publicVar2);
    }
    /*@Override
    public String toString()
    {
        return "blah";
    }*/
}
```

Appendix A – Explanation from a website

I downloaded this explanation some time ago, but I cannot find the website to cite the source. One of the main reasons I've included it is because I like the diagram it shows.

Description:

Object-oriented programming languages such as Java include features that support the object-oriented programming paradigm. These features include data abstraction, encapsulation, information hiding, polymorphism, and inheritance. Encapsulation refers to the bundling of both data and methods (behaviors) into a single entity called a class, which can be easily incorporated into different programs. A class defined in a Java program is used to declare objects of that class type. Using the feature of encapsulation, the programmer can restrict access to components of the class, which is a key strength of encapsulation in Java. Java classifies class members (data members and member methods) into one of three categories using access specifiers, which are also reserved words: private, public, and protected. Access specifiers determine the visibility or accessibility of an object's members, as specified in the table below:

Types of Threat	**Accesibility**
Private	Accessible only by member methods of the class
Public	Accessible by methods both inside and outside of the class
Protected	Accessible by member methods of the class and by member methods of any subclass

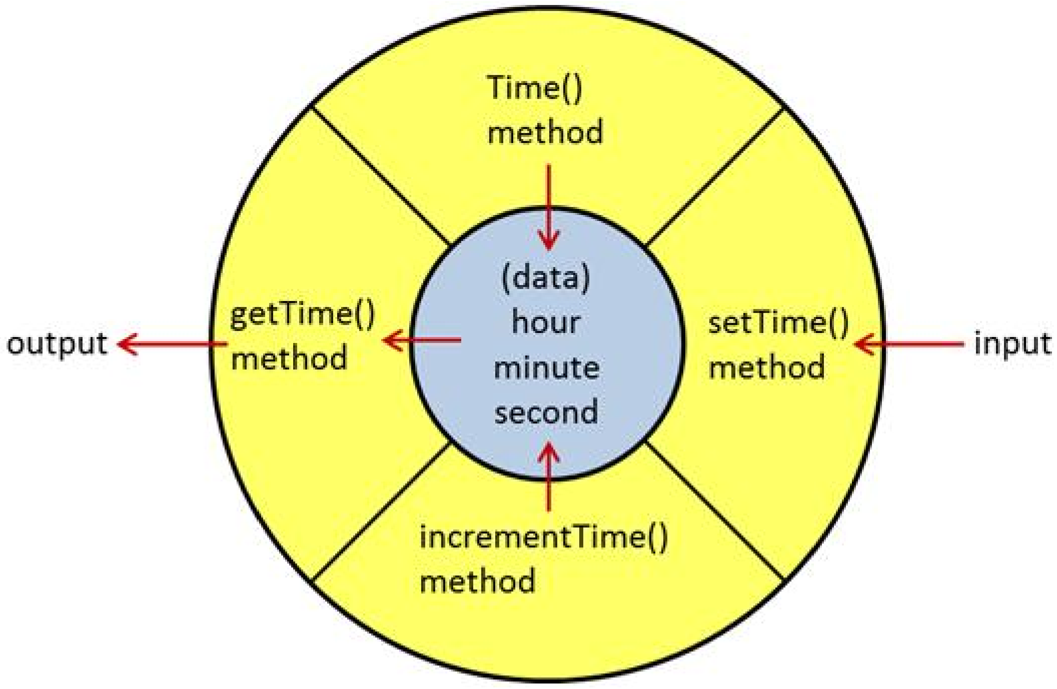
In Java, the access specifier is included in each of the class's variable and method declarations, preceding the data type. Be careful! When no modifier is given, the default access specifier is package-private, meaning that methods in any class within the same package will be able to use, access, or modify these members.

An example of a Java class declaration follows:

```
public class Time {
    private int hour;
    private int minute;
    private int second;

    public Time()
        { /* constructor implementation */ }
    public void setTime(int newHour, int newMinute, int newSecond)
        { /* mutator implementation */ }
    public int[] getTime()
        { /* accessor implementation */ }
    public void incrementTime()
        { /* mutator implementation */ }
}
```

In the example above, the data members hour, minute, and second and the member methods setTime(), getTime(), and incrementTime() are bundled into a single autonomous class named Time. This demonstrates the encapsulation feature of Java. When all data is declared as private, the data is only accessible through the methods provided by the class. This restricted access is illustrated below.



Such restricted access requires programmers to write specialized accessor methods like getTime() for acquiring the values of private data members and specialized mutator methods like setTime() and incrementTime() for performing operations on them. This allows programmers to validate changes to data members before making such a change. In this example, the setTime() method would be written to check for valid values for military time (hour is between 0 and 23 and minute and second are between 0 and 59.)

Encapsulation hides the private data members and the implementation details of a class, but it does not necessarily mean a class is completely isolated. Many objects must share information with other objects, usually with good reason. However, protecting private data by providing accessor and mutator methods ensures that an object is affected only in known ways by other objects in the system. Sharing should be minimized so that the class provides as few interfaces (public methods) as possible. Other methods required for the class should be private, so they are not available outside the class.

Using encapsulation properly and providing a controlled public interface is similar to a fast-food restaurant that has no indoor customer access and provides only a drive-up window. The customer can read the menu, place an order from the menu, pay for the order, and pick up the completed order. The customer does not know exactly what is occurring inside the restaurant; he or she knows only that placing an order and paying for it results in the food showing up in the completed order. The customer may return to the same fast-food restaurant every day for a month and interact with the restaurant in the same way. If the restaurant hires a consultant to cut costs and changes the way orders are filled, the customer has no knowledge of the change, and does not really need to know about it. As long as the customer interacts with the restaurant in the same way and the food tastes just as good each time, the customer does not really even care about the internal operations of the restaurant.