**Encapsulation - Data Binding/Data Hiding**



Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of fully encapsulated class.

**To achieve encapsulation in Java −**

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

**Example**

Following is an example that demonstrates how to achieve Encapsulation in Java −

**/\* File name : EncapTest.java \*/**

*public class EncapTest {*

*private String name;*

*private String idNum;*

*private int age;*

*public int getAge() {*

*return age;*

*}*

*public String getName() {*

*return name;*

*}*

*public String getIdNum() {*

*return idNum;*

*}*

*public void setAge( int newAge) {*

*age = newAge;*

*}*

*public void setName(String newName) {*

*name = newName;*

*}*

*public void setIdNum( String newId) {*

*idNum = newId;*

*}*

*}*

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program −

/\* File name : RunEncap.java \*/

*public class RunEncap {*

*public static void main(String args[]) {*

*EncapTest encap = new EncapTest();*

*encap.setName("James");*

*encap.setAge(20);*

*encap.setIdNum("12343ms");*

*System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());*

*}*

*}*

This will produce the following result −

Name : James Age : 20

**Benefits of Encapsulation**

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.
* The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

### Advantage of Encapsulation in java

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

### Simple example of encapsulation in java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

//save as Student.java

*package com.javatpoint;*

*public class Student{*

*private String name;*

*public String getName(){*

*return name;*

*}*

*public void setName(String name){*

*this.name=name*

*}*

*}*

//save as Test.java

*package com.javatpoint;*

*class Test{*

*public static void main(String[] args){*

*Student s=new Student();*

*s.setName("vijay");*

*System.out.println(s.getName());*

*}*

*}*

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay

**Just read the program twice. You may guess some values how they are printed.**

*public class Officer{*

*int salary;*

*public static void main(String args[]){*

*Officer o1 = new Officer();*

*Officer o2 = new Officer();*

*o1.salary = 5000;*

*o2.salary = 6000;*

*System.out.println("Salary of o1 before initially Rs." + o1.salary);****// 5000***

*System.out.println("Salary of o2 before initially Rs." + o2.salary);****// 6000***

*o1.salary = 7000;*

*System.out.println("Salary of o1 after changing o1 Rs.:" + o1.salary);****// 7000***

*System.out.println("Salary of o2 after changing o1 Rs." + o2.salary);****// 6000***

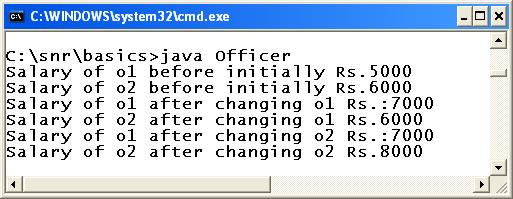
*o2.salary = 8000;*

*System.out.println("Salary of o1 after changing o2 Rs.:" + o1.salary);****// 7000***

*System.out.println("Salary of o2 after changing o2 Rs." + o2.salary);****// 8000***

*}*

*}*



**Explanation**

**salary** is an instance variable and should be accessed with an object only, through [static main()](http://way2java.com/oops-concepts/public-static-void-mainstring-args/) method. Two objects **o1** and **o2** are created and **salary** field is called. The object **o1**gets a salary of 5000 and **o2** gets 6000. If an instance variable is called with an object, a location is created in the memory. Now two locations are created in the memory for **o1.salary**and **o2.salary**. That is, one location of salary is binded with **o1** and another location of salary is binded with **o2**. This is is known as **data binding**.

Due to this binding habit, the salary location of **o1** is hidden from **o2** and similarly the salary location of **o2** is hidden from **o1**. This is known as **data hiding**.

Due to hiding, **o1** cannot access **o2** salary and **o2** cannot access**o1** salary. That is, the variable salary is encapsulated with object **o1** and becomes one unit of data. Similarly another unit **o2** is made with its own salary, entirely separate from **o1** unit. This is known as **encapsulation**.

Encapsulation is a pharmaceutical term where drug powder is placed in a capsule to hide its properties, like taste and color etc., from the patients. Due to data hiding, the properties (the value of fields(instance variables)) of an object are hidden from other objects. This exactly resembles a capsule. The advantage of encapsulation is **one instance variable can be given multiple values**, of course, by calling with different objects. Encapsulation is an object oriented concept or paradigm or design or principle; you may say anything, all amounts the same.

In the later part of the program, salary of **o1** is changed. Only the salary of **o1** is affected but not **o2** and when **o2** salary is changed, **o1** salary is not affected. It is due the concept of encapsulation. This is where object-oriented programming takes a lead over procedure-oriented programming, like C. Finally, **data binding** leads to **data hiding** and data hiding leads to **encapsulation**.

###### **Encapsulation – Want or Don’t Want**

Now let us see how to assign the properties of one object to another by modifying the previous Officer program.

*public class Officer1{*

*int salary;*

*public static void main(String args[]){*

*Officer1 o1 = new Officer1();*

*Officer1 o2 = new Officer1();*

*Officer1 o3 = new Officer1();*

*Officer1 o4 = new Officer1();*

*o1.salary = 5000;*

*o2.salary = o1.salary;* ***// assigning variable to variable***

*System.out.println("o1 initial salary Rs." + o1.salary); //* ***5000***

*System.out.println("o2 initial salary Rs." + o2.salary); //* ***5000***

*o1.salary = 6000;*

*System.out.println("o1 salary after changing o1 Rs." + o1.salary);* ***// 6000***

*System.out.println("o2 salary after changing o1 Rs." + o2.salary);* ***// 5000***

*o3.salary = 7000;*

*o4 = o3;* ***// assigning object to object***

*System.out.println("o3 initial salary Rs." + o3.salary);* ***// 7000***

*System.out.println("o4 inital salary Rs." + o4.salary);* ***// 7000***

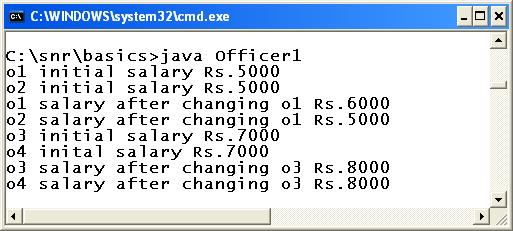
*o3.salary = 8000;*

*System.out.println("o3 salary after changing o3 Rs." + o3.salary);* ***// 8000***

*System.out.println("o4 salary after changing o3 Rs." + o4.salary);* ***// 8000***

*}*

*}*



The basic concept in the above code is to observe encapsulation by assigning variable to variable and object to object.

Four objects are created. **o1** salary is set to 5000 and later **o1 salary** is assigned to **o2** salary. They maintain separate locations and advantage is if **o1 salary** is changed to 6000, **o2 salary** is not affected. Observe the output. Later **o3** salary is set to 8000. Object **o4** is assigned with **o3**. In Java, when object to object is assigned, they maintain or refer the same location. That is, **o3** and **o4** refer or point to one salary only. The affect is if one object **o3** changes the salary, the other object **o4** also changes. Observe the output. Similarly, you can change **o4** salary and notice **o3** salary also changes; no encapsulation.

Final conclusion, we can make out is, if variable to variable is assigned perfect encapsulation is maintained and object to object is assigned no encapsulation is maintained. This must be kept in mind while assigning properties of one object to another. Another style of copying is [**cloning**](http://way2java.com/java-lang/cloning-%e2%80%93-duplicating-an-object-marker-interface/) which we get later.

#### Using Thin main() Method :

In the previous, Officer1 program, all the code is written in the **main()** method only. You may get output but it is against coding principles. Coding principles say, keep the main() method as thin as possible by placing less code. Limit the main() method just to create objects and calling the methods. Write any code in a method and call the method from the main(). This increases the **reusability**, **readability** and **debugging** becomes easier.

**Let us modify the previous program.**

*public class Officer2{*

*int salary;*

*public void display(int sal){*

*salary = sal;*

*}*

*public static void main(String args[]){*

*Officer2 o1 = new Officer2();*

*Officer2 o2 = new Officer2();*

*o1.display(5000);*

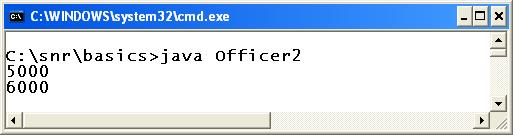
*o2.display(6000);*

*System.out.println(o1.salary);*  ***// 5000***

*System.out.println(o2.salary);*  ***// 6000***

*}*

*}*



The difference between **Officer1** and **Officer2** programs is assigning the values to objects **o1**and **o2**. **sal** is a local variable to **display()** method and **salary** is an instance variable (the other name of global variable). Local variable **sal** is assigned to instance variable **salary**. When**o1.display(5000)** is called, 5000 goes to **sal** and in turn **sal** gives to salary. Because **salary** is an instance variable, one location on the name **o1.salary** is created and is accessed though [main()](http://way2java.com/oops-concepts/public-static-void-mainstring-args/). But **o1** is not linked to **sal**because **sal** is a local variable and thereby no location is created.

## Advantage of Encapsulation in Java and OOPS

Here are few advantages of using **Encapsulation** while writing code in Java or any Object oriented programming language:

1. Encapsulated Code is more flexible and easy to change with new requirements.

2. Encapsulation in Java makes unit testing easy.

3. Encapsulation in Java allows you to control who can access what.

4. Encapsulation also helps to write immutable class in Java which is a good choice in multi-threading environment.

5. Encapsulation reduces coupling of modules and increases cohesion inside a module because all piece of one thing is encapsulated in one place.

6. Encapsulation allows you to change one part of code without affecting other parts of code.

**What should you encapsulate in code**

Anything which can be change and more likely to change in near future is a candidate of Encapsulation. This also helps to write more specific and cohesive code. An example of this is object creation code, code which can be improved in future like sorting and searching logic.

## Design Pattern based on Encapsulation in Java

Many design pattern in Java uses encapsulation concept, one of them is [Factory pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) which is used to create objects. Factory pattern is a better choice than new operator for creating an object of those classes whose creation logic can vary and also for creating different implementation of the same interface. BorderFactory class of JDK is a good example of encapsulation in Java which creates different types of Border and encapsulate creation logic of Border. [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) also encapsulate how you create an instance by providing getInstance() method. since an object is created inside one class and not from any other place in code you can easily change how you create an object without affect another part of code.

### Important points about encapsulation in Java.

1. "Whatever changes encapsulate it" is a famous design principle.

2. Encapsulation helps in loose coupling and high cohesion of code.

3. Encapsulation in Java is achieved using access modifier private, protected and public.

4. Factory pattern , Singleton pattern in Java makes good use of Encapsulation.

# ****1. What Is Encapsulation?****

It’s quite difficult to understand the concept of encapsulation from a definition. So let’s understand what encapsulation is by looking at some code examples.

Basically, there are two forms of encapsulation in OOP.

First, encapsulation is a technique that packages related data and behaviors into a single unit.

**Let’s see an example:**

|  |  |
| --- | --- |
|  | *class Person {*  *String name;*  *int age;*  *void talk() {*  *}*  *void think() {*  *}*  *void work() {*  *}*  *void play() {*  *}*  *}* |

Here, the common characteristics and behaviors of a person are packaged into a single unit: the Person class. This is the process of encapsulation.

The Person class is an encapsulation unit. A Person object exposes its attributes and behaviors to the outside world:

|  |  |
| --- | --- |
|  | *Person you = new Person();*  *you.name = "John";*  *you.work();* |

Here, encapsulation hides implementation details of the Person class from other objects.

Likewise, creating an interface is also the process of encapsulation:

|  |  |
| --- | --- |
|  | *interface Human {     void eat();     void talk();     void think(); }* |

Again, in terms of encapsulation, this interface groups the essential behaviours of human-being in a single unit.

Second, encapsulation is a technique for protecting data from misuse by the outside world, which is referred as ‘***information hiding***’ or ‘***data hiding***’

In Java, the access modifier **private** is used to protect data and behaviours from outside. Let’s modify the Person class above to prevent the attributes name and age from being modified by other objects:

|  |  |
| --- | --- |
|  | *class Person {*  *private String name;*  *private int age;*  *}* |

Here, the fields age and name can be only changed within the Person class. If someone attempts to make a change like this:

|  |  |
| --- | --- |
|  | *Person p = new Person();*  *p.name = "Tom"; // ERROR!* |

The code won’t compile because the field name is marked as private.

But what if we want the other objects to be able to read the name and age? In this case, we provide methods whose name in the form of getXXX() - so called **getters** in Java. Hence we add two getters to the Person class:

|  |  |
| --- | --- |
|  | *class Person {*  *private String name;*  *private int age;*  *public String getName() {*  *return name;*  *}*  *public String getAge() {*  *return age;*  *}*  *}* |

This is the process of hiding information. The other objects cannot access the data directly. Instead, they have to invoke the getters which are designed to protect the data from misuse or unwanted changes.

What if we want the outside world to be able to modify our data in a safety manner? In this case, we can provide methods in the pattern of setXXX() - the so called **setters** in Java. For example, creating a setter for the field name:

|  |  |
| --- | --- |
|  | *public void setName(String name) {*  *if (name == null || name.equals("")) {*  *throw new IllegalArgumentException("name cannot be null or empty!");*  *}*  *this.name = name;*  *}* |

Here, someone can change the name of a Person object via this setter method, but he cannot set an empty or null name - as the setter will throw an exception if doing so. This protects data from misuse or malicious changes. For example:

|  |  |
| --- | --- |
|  | *Person p = new Person();*  *p.setName("");  // ERROR: IllegalArgumentException will be thrown*  *p.setName("Tom");   // OK, legal* |

Likewise, we can implement a setter for the age attribute that allows the caller to set age in a valid range. For example:

|  |  |
| --- | --- |
|  | *public void setAge(int age) {*  *if (age < 18 || age > 55) {*  *throw IllegalArgumentException("Age must be from 18 to 55");*  *}*  *this.age = age;*  *}* |

So far you’ve got an understanding about what encapsulation is in OOP and how it is implemented in Java. In short, encapsulation is a technique for hiding implementation details and restricting access for the outside world.

# ****2. Why Is Encapsulation?****

It’s because encapsulation has a number of advantages that increase the reusability, flexibility and maintainability of the code.

* **Flexibility**: It’s more flexible and easy to change the encapsulated code with new requirements. For example, if the requirement for setting the age of a person changes, we can easily update the logic in the setter method setAge().
* **Reusability**: Encapsulated code can be reused throughout the application or across multiple applications. For example, the Person class can be reused whenever such type of object is required.
* **Maintainability**: Application ode is encapsulated in separate units (classes, interfaces, methods, setters, getters, etc) so it’s easy to change or update a part of the application without affecting other parts, which reduces the time of maintenance.

# ****3. How Is Encapsulation Done in Java?****

As you can see in the above examples, encapsulation is implemented in Java using interfaces, classes, access modifiers, setters and getters.

A class or an interface encapsulates essential features of an object.

Access modifiers (private, public, protected) restrict access to data at different levels.

Setters and getters prevent data from misuse or unwanted changes from other objects.

That’s all for encapsulation topic today. Right now, take a pencil and a paper to take note what you have learned.

# ****4. Encapsulation vs. Abstraction****

So far you got a proper understanding about [abstraction](http://www.codejava.net/java-core/the-java-language/what-is-abstraction-in-java-the-why-and-the-truth) and encapsulation in Object-Oriented Programming with Java. To summary:

* Abstraction is the process of modelling real things in the real word into programming language. In Java, the process of abstraction is done using interfaces, classes, abstract classes, fields, methods and variables. Everything is an abstraction.
* Encapsulation is the process of hiding information details and protecting data and behaviour of an object from misuse by other objects. In Java, encapsulation is done using access modifiers (public, protected, private) with classes, interfaces, setters, getters.

So, what are the commons and differences between abstraction and encapsulation?

Many programmers have been wondering about this.

If you are wondering about this too, here’s my answer:

Hey, there are NO commons and differences between abstraction and encapsulation. Why? It’s because they are not comparable.

They are about different things, so we cannot compare them.

As I told you in the article [What is Abstraction in Java - the WHY and the Truth](http://www.codejava.net/java-core/the-java-language/what-is-abstraction-in-java-the-why-and-the-truth), abstraction is the fundamental concept on which other things rely on such as encapsulation, inheritance and polymorphism. In other words, if there is no abstraction, encapsulation would not exist.

Encapsulation is done based on abstraction. Imagine you are building a house. The house is made by bricks. Abstraction is the bricks and encapsulation is the house. Got it?

That’s all about abstraction and encapsulation today. You know, it took me years to really understand and figure out this mater. Now you got it in just some minutes.

**What is encapsulation and why is it used.?**

Encapsulation is one of the four **fundamental OOP concepts**.Encapsulation is the **technique of making the fields in a class private and providing access to the fields via public methods.** If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Encapsulation can be described as a protective barrier that **prevents the code and data being randomly accessed by other code** defined outside the class. Access to the data and code is tightly controlled by an interface.

The **main benefit of encapsulation is the ability to modify our implemented code without breaking the code** of others who use our code. With this feature **Encapsulation gives maintainability, flexibility and extensibility** to our code.

Take a small example:

*public class EncapTest{*

*private String name;*

*private String idNum;*

*private int age;*

*public int getAge(){*

*return age;*

*}*

*public String getName(){*

*return name;*

*}*

*public String getIdNum(){*

*return idNum;*

*}*

*public void setAge( int newAge){*

*age = newAge;*

*}*

*public void setName(String newName){*

*name = newName;*

*}*

*public void setIdNum( String newId){*

*idNum = newId;*

*}*

*}*

The above methods are called Accessors(aka getters and setters). Now you might ask,

**Why should you use accessors..?** There are actually many good reasons to consider using accessors rather than directly exposing fields of a class.Getter and Setters make APIs more stable.

For instance, consider a **field public in a class which is accessed by other classes**. Now later on, you want to add any extra logic while getting and setting the variable. This will impact the existing client that uses the API. So any changes to this public field will require change to each class that refers it. On the contrary, with accessor methods, one can easily add some logic like cache some data, lazily initialize it later. Moreover, one can fire a property changed event if the new value is different from the previous value. All this will be seamless to the class that gets value using accessor method.

**Note:**

**Data Binding** =Assemble of instance variable and object of class  
**Data Hiding** = obj1 dont know information of obj2 and vice versa.  
**Encapsulation** = 1 variable can be assign multiple value.

**Q. Officer o1=new Officer();  
 Officer o2=new Officer();  
 o1.sal=5000;  
 o2.sal=o1.sal;  
Here, o1.sal–>5000 or o1.sal–>address of 5000 in both of them what is true  
then how it reflect to o2.sal=o1.sal; here where o2.sal is getting the value can u please tell me am not catching the concept ?**

A. It simply like a variable assignment. o1.salary is assigned to o2.salary. At runtime, o1.salary is replaced by 5000 and then assigned to o2.salary.

**Q. What is the difference between Encapsulation and Abstraction?**

Using objects Java maintains encapsulation. Books may write many pages on encapsulation, the straight answer is “one variable multiple values”.

Abstraction varies a little bit. I can give number of examples of my own. In C-lang, you are using printf() function. Without known the internal code of printf(), you are writing to DOS prompt. How it is possible? printf() function code is now shown to you and this we say technically, printf() code is abstracted from you. Similarly, you are riding a motor bike without knowing the piston and accelerator mechanism. We say, the mechanisms are abstracted from the user. To design a auto mobile, we require an auto mobile engineer, but to use, anyone can do. Similarly, TV mechanism is abstracted from you and still you are using.

**Why and how Assigning Variable to variable they maintain separate locations, and assigning object to object they refer same location?**

Java is designed on the rule: Variables are assigned by value and objects are assigned by reference. With object assignment only, Java achieved pointers indirectly.

**Why we use private in a class to show encapsulation?**

One of the programming techniques is declaring variables private and methods public. Other classes can access the private variables through public methods. In public public methods, you can control the input.

*private int birthdate;  
public setBirthDate(int birthdate){  
if(birthdate < 1 || birthdate > 31)  
 birtthdate = 0;  
else  
 this.birthdate = birthdate;  
}*

If the variable is not private, the other class can assign its own value, which may be wrong.

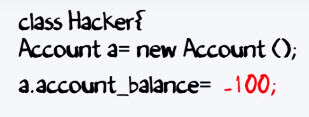
**Can overloaded methods be override too?**

Yes, you can.

In Java, encapsulation is a mechanism of wrapping data (variables) and code together as a single unit.

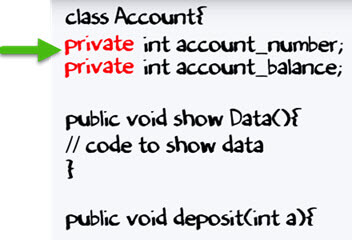
Suppose a hacker managed to gain access to the code of your bank account. Now, he tries to deposit amount 100 into your account by two ways. Let see his first method or approach.

**Approach 1:** He tries to deposit an invalid amount (say -100) into your bank account by manipulating the code.

[](http://cdn.guru99.com/images/java/052016_0638_LearnJavaEn1.jpg)

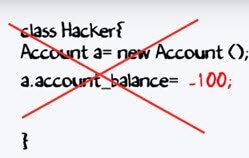
**Now, the question is – Is that possible?**

Let see whether it is possible or not? Usually, a variable and class is set as a "Private" as shown below. It can only be accessed with the corresponding methods, and no other alternatives methods are allowed to carry out the operations.

[](http://cdn.guru99.com/images/java/052016_0638_LearnJavaEn2.jpg)

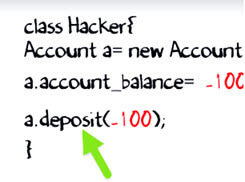
If a data member is private, it means it can only be accessed within the same class. No outside class can access private data member or variable of other class.

So in our case hacker cannot deposit amount 100 to your account.

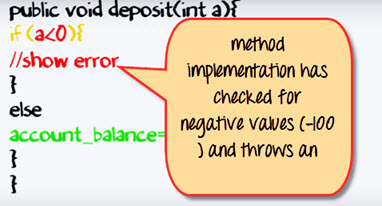
[](http://cdn.guru99.com/images/java/052016_0638_LearnJavaEn3.jpg)

**Approach 2**:

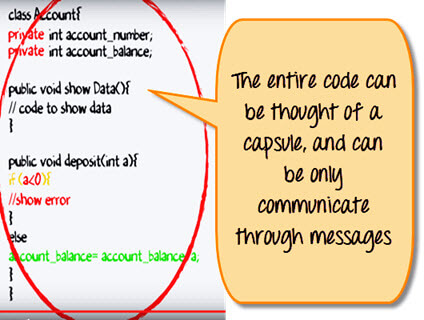
Hacker's first approach failed to deposit the amount. Next, he tries to do same by using "deposit" method. To carry out this operation he uses "deposit" method.

[](http://cdn.guru99.com/images/java/052016_0638_LearnJavaEn4.jpg)

But method implementation has check for negative values. So the second approach also fails in depositing amount 100 into your account.

[](http://cdn.guru99.com/images/java/052016_0638_LearnJavaEn5.jpg)

Thus, you never expose your data to an external party. Which makes your application secure.

[](http://cdn.guru99.com/images/java/052016_0638_LearnJavaEn6.jpg)

The entire code can be thought of a capsule, and you can only communicate through the messages. Hence the name encapsulation.

**Few notes on Encapsulation in Java:**

* In Java, encapsulation is binding the data with its related functionalities
* Here functionalities mean "methods" and data means "variables"
* So we keep variable and methods in one place. That place is "class."
* "Class" is the base for encapsulation.
* With Java Encapsulation, you can hide (restrict access) to critical data members in your code, which improves security
* As we discussed earlier, if a data member is declared "private", then it can only be accessed within the same class. No outside class can access data member (variable) of other class.
* However if you need to access these variables, you have to use **public "getter" and "setter"** methods.
* Setup public "getter" and "setter" method to update and read the private data field. This will allow data access from private class.

Frequently, java encapsulation is referred for **data hiding**. But more than data hiding, encapsulation concept is meant for better management or grouping of related data.

To achieve a lesser degree of encapsulation in Java you can use modifiers like "protected" or "public". With encapsulation, developers can change one part of the code easily without affecting other.

**Often encapsulation is misunderstood with Abstraction. To get clear on this,**

* Encapsulation is more about "How" to achieve that functionality
* Abstraction is more about "what" a class can do.

A simple example to understand this difference is a mobile phone. Where the complex logic in the circuit board is encapsulated in a touch screen, and the interface is provided to abstract it out.