Encapsulation in Java — How to master OOPs with Encapsulation?

Object-Oriented Programming or better known as OOPs is one of the major pillars of Java that has leveraged its power and ease of usage. To become a professional Java developer, you must get flawless control

over the various **Java OOPs concepts like** Inheritance, Abstraction, Encapsulation, and Polymorphism. Through the medium of this article, I will give you a complete insight into one of the most important concepts of OOPs i.e Encapsulation in Java, and how it is achieved.

Below are the topics, I will be discussing in this article:

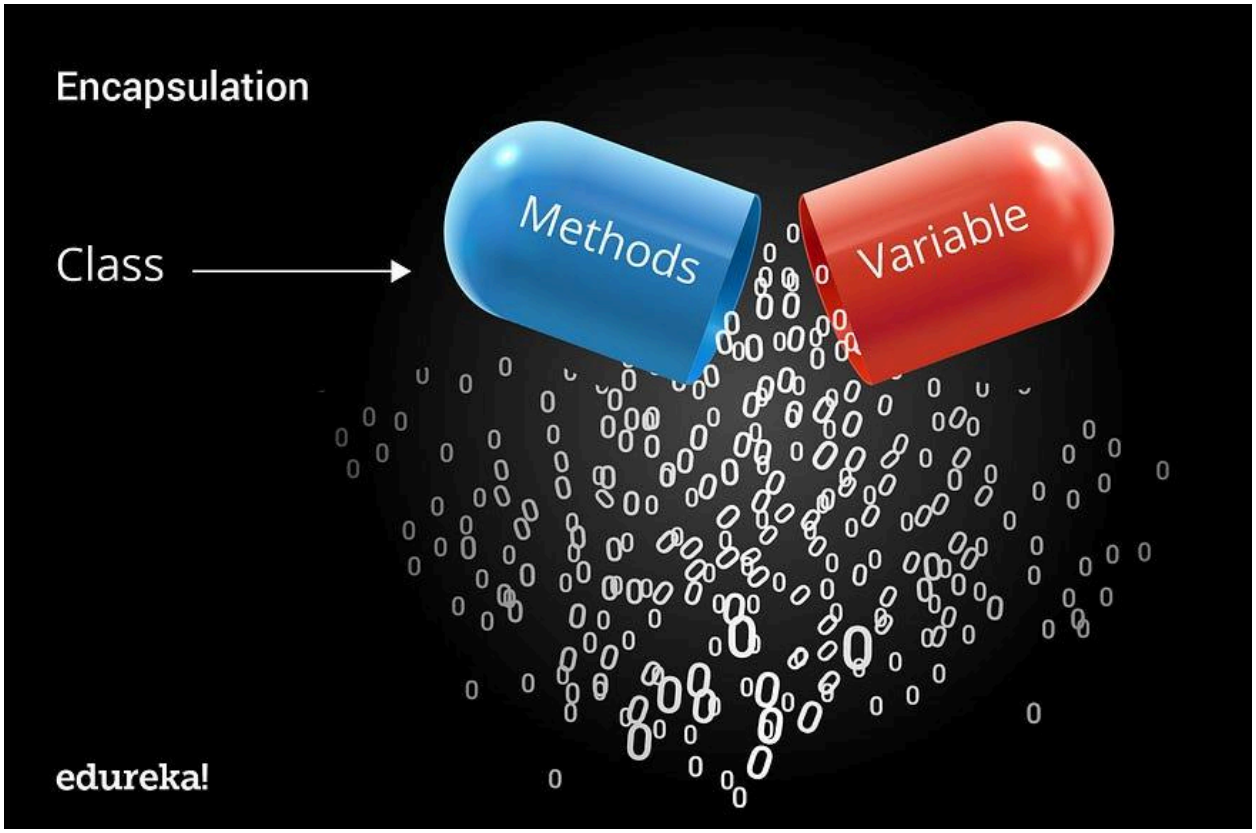
- Introduction to Encapsulation
- Why we need Encapsulation in Java?
- Benefits of Encapsulation
- A Real-Time Example

Introduction to Encapsulation

Encapsulation refers to wrapping up of data under a single unit. It is the mechanism that binds code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

In this, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

Now, let's take the example of a medical capsule, where the drug is always safe inside the capsule. Similarly, through encapsulation, the methods and variables of a class are well hidden and safe.



Encapsulation in Java can be achieved by:

- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the variables values.

Now, let's look at the code to get a better understanding of encapsulation:

```
public class Student {

private String name;

public String getName() {

return name;

}
```

```
public void setName(String name) {

this.name = name;

}

}

class Test{

public static void main(String[] args) {

Student s=new Student();

s.setName("Harry Potter");

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

}
```

```
}
```

As you can see in the above code, I have created a class Student which has a private variable name**.** Next, I have created a getter and setter to get and set the name of a student. With the help of these methods, any class which wishes to access the name variable has to do it using these getter and setter methods.

Now let's see one more example and understand Encapsulation in depth. In this example, the Car class has two fields -name and top speed. Here, both are declared as private, meaning they can not be accessed directly outside the class. We have some getter and setter methods like get Name, set Name, set Top Speed etc., and they are declared as public. These methods are exposed to "outsiders" and can be used to change and retrieve data from the Car object. We have one method to set the top speed of the vehicle and two getter methods to retrieve the max speed value either in MPH or KMHt. So basically, this

is what encapsulation does — it hides the implementation and gives us the values we want. Now, let's look at the code below.

```
package Edureka;

public class Car {

private String name;

private double topSpeed;

public Car() {}

public String getName(){

return name;

}
```

```
public void setName(String name){

this.name= name;

}

public void setTopSpeed(double speedMPH){

topSpeed = speedMPH;

}

public double getTopSpeedMPH(){

return topSpeed;

}

public double getTopSpeedKMH(){
```

```
return topSpeed*1.609344;

}

}
```

Here, the main program creates a Car object with a given name and uses the setter method to store the top speed for this instance. By doing this, we can easily get the speed in MPH or KMH without caring about how speed is converted in the Car class.

```
package Edureka;

public class Example{

public static void main(String args[])

Car car =new Car();
```

```
car.setName("Mustang GT 4.8-litre V8");
```

```
car.setTopSpeed(201);
```

```
System.out.println(car.getName()+ " top speed in MPH is " +
car.getTopSpeedMPH());
```

```
System.out.println(car.getName() + " top speed in KMH is " +
car.getTopSpeedKMH());
```

So, this is how Encapsulation can be achieved in Java. Now, let's move further and see why do we need Encapsulation.

Why we need Encapsulation in Java?

Encapsulation is essential in Java because:

- It controls the way of data accessibility
- Modifies the code based on the requisites
- Helps us to achieve a loose couple
- Achieves simplicity of our application

- It also allows you to change the part of the code without disrupting any other functions or code present in the program

Now, let's consider a small example that illustrates the need for encapsulation.

```
class Student {

int id;

String name;

}

public class Demo {

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

```
Student s = new Student();


s.id = 0;


s.name="";


s.name=null;


}


}
```

In the above example, it contains two instance variables as access modifier. So any class within the same package can assign and change values of those variables by creating an object of that class. Thus, we don’t have control over the values stored in the Student class as variables. In order to solve this problem, we encapsulate the Student class.

So, these were the few pointers that depict the need of Encapsulation. Now, let's see some benefits of encapsulation.

Benefits of Encapsulation

- **Data Hiding:** Here, a user will have no idea about the inner implementation of the class. Even users will not be aware of how the class is storing values in the variables. He/she will only be aware that we are passing the values to a setter method and variables are getting initialized with that value.
- () etc. or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program. **Increased Flexibility:** Here, we can make the variables of the class as read-only or write-only depending on our requirement. In case you wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge

- **Reusability:** It also improves the re-usability and easy to change with new requirements.

Now that we have understood the fundamentals of encapsulation, let's dive into the last topic of this article and understand Encapsulation in detail with the help of a real-time example.

A Real-Time Example of Encapsulation

Let's consider a television example and understand how internal implementation details are hidden from the outside class. Basically, in this example, we are hiding inner code data i.e. circuits from the external world by the cover. Now in Java, this can be achieved with the help of access modifiers. Access modifiers set the access or level of a class, constructors variables, etc. As you can see in the below code, I have used a private access modifier to restrict the access level of the class. Variables declared as private are accessible only within the Television class.

```
public class Television{

private double width;

private double height;

private double Screensize;

private int maxVolume;

print int volume;

private boolean power;

public Television(double width, double height, double
screenSize)

{
```

```
this.width=width;

this.height=height;

this.screenSize=ScreenSize;

}

public double channelTuning(int channel){

switch(channel){

case1: return 34.56;

case2: return 54.89;

case3: return 73.89;

case1: return 94.98;
```

```
}return 0;

}

public int decreaseVolume(){

if(0<volume) volume --;

return volume;

}

public void powerSwitch(){

this.power=!power;

}

public int increaseVolume(){
```

```
if(maxVolume>volume) volume++;

return volume;

}

}

class test{

public static void main(String args[]){

Television t= new Television(11.5,7,9);

t.powerSwitch();

t.channelTuning(2);

t.decreaseVolume();
```

```
t.increaseVolume();


television.width=12; // Throws error as variable is private and
cannot be accessed outside the class


}


}
```

In the above example, I have declared all the variables as private and methods, constructors and class as public. Here, constructors, methods can be accessed outside the class. When I create an object of Television class, it can access the methods and constructors present in the class, whereas variables declared with private access modifier are hidden. That's why when you try to access an error ***width variable*** in the above example, it throws . That's how internal implementation details are hidden from the other classes. This is how Encapsulation is achieved in Java.