Encapsulation:

In Java, encapsulation refers to the principle of bundling data (variables) and methods (functions that operate on the data) together as a single unit, called a class. It restricts direct access to some of the object's components, usually to enforce proper usage and to prevent unintended interference. Here are the key aspects of encapsulation in Java:

1. **Private Instance Variables**: Instance variables of a class are typically declared as private to hide their visibility from other classes. This prevents direct access and modification, ensuring data integrity and security.

```
java
Copy code
public class MyClass {
    private int myVariable;
}
```

2. **Public Methods**: Public methods are provided to access and modify the private variables. These methods are often called getter and setter methods.

```
java
Copy code
public class MyClass {
    private int myVariable;

    public int getMyVariable() {
        return myVariable;
    }

    public void setMyVariable(int newValue) {
        myVariable = newValue;
    }
}
```

3. **Access Modifiers**: By using access modifiers (private, protected, public), you control the visibility of classes, methods, and variables. Private variables can only be accessed within the same class, while public methods can be accessed from outside the class.
4. **Encapsulation Benefits**:
    - **Controlled Access**: Protects data from unintended or unauthorized access.
    - **Flexibility**: Allows the internal implementation of a class to be changed without affecting the overall code that uses the class.
    - **Modularity**: Encapsulation promotes modularity by organizing code into self-contained units (classes), making it easier to manage and understand.
5. **Example**:

```
java
Copy code
public class Car {
    private String model;
    private int year;

    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
```

```
        public int getYear() {
            return year;
        }

        public void setYear(int year) {
            this.year = year;
        }
    }
```

The example demonstrates encapsulation in Java using a `Car` class:

1. **Private Variables**:
   - `model` and `year` are declared as private variables. This means they can only be accessed and modified within the `Car` class itself.
2. **Constructor**:
   - The `Car` class has a constructor that initializes the `model` and `year` of the car when an object is created.
3. **Getter and Setter Methods**:
   - Public getter methods (`getModel()` and `getYear()`) allow other classes to retrieve the values of `model` and `year`.
   - Public setter methods (`setModel(String model)` and `setYear(int year)`) allow other classes to modify the values of `model` and `year`, while providing controlled access.
4. **Encapsulation Benefits**:
   - Data (`model` and `year`) is encapsulated within the `Car` class, ensuring that it can be accessed and modified only through defined methods.
   - This protects the integrity of the data and allows the `Car` class to change its internal implementation (e.g., how `model` and `year` are stored or validated) without affecting the code that uses `Car` objects.

Encapsulation here helps in creating a modular and maintainable `Car` class, where the internal details (like `model` and `year`) are hidden and accessed only through controlled interfaces (the getter and setter methods).

Here's an example of how you can access and modify the `Car` class's variables (`model` and `year`) from another class using its public getter and setter methods:

```
java
Copy code
// Car.java
public class Car {
    private String model;
    private int year;

    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
```

```
    }
}
```

Now, let's create another class `Main` where we'll instantiate a `Car` object, access its methods to retrieve and modify its properties:

```
java
Copy code
// Main.java
public class Main {
    public static void main(String[] args) {
        // Create a Car object
        Car myCar = new Car("Toyota Camry", 2023);

        // Access and print current values using getter methods
        System.out.println("Current Car Model: " + myCar.getModel());
        System.out.println("Current Car Year: " + myCar.getYear());

        // Modify values using setter methods
        myCar.setModel("Honda Civic");
        myCar.setYear(2022);

        // Access and print modified values using getter methods
        System.out.println("Modified Car Model: " + myCar.getModel());
        System.out.println("Modified Car Year: " + myCar.getYear());
    }
}
```

Explanation:

- In the `Main` class, we first create a `Car` object named `myCar` with initial values `"Toyota Camry"` and `2023`.
- We then use the `getModel()` and `getYear()` methods of the `Car` class to retrieve and print the current values of `model` and `year`.
- Next, we use the `setModel(String model)` and `setYear(int year)` methods of the `Car` class to modify the values of `model` to `"Honda Civic"` and `year` to `2022`.
- Finally, we again use the `getModel()` and `getYear()` methods to retrieve and print the modified values of `model` and `year`.

This demonstrates how encapsulation allows controlled access to the internal state (`model` and `year` in this case) of an object (`Car`) through well-defined interfaces (getter and setter methods), ensuring data integrity and flexibility in the implementation.