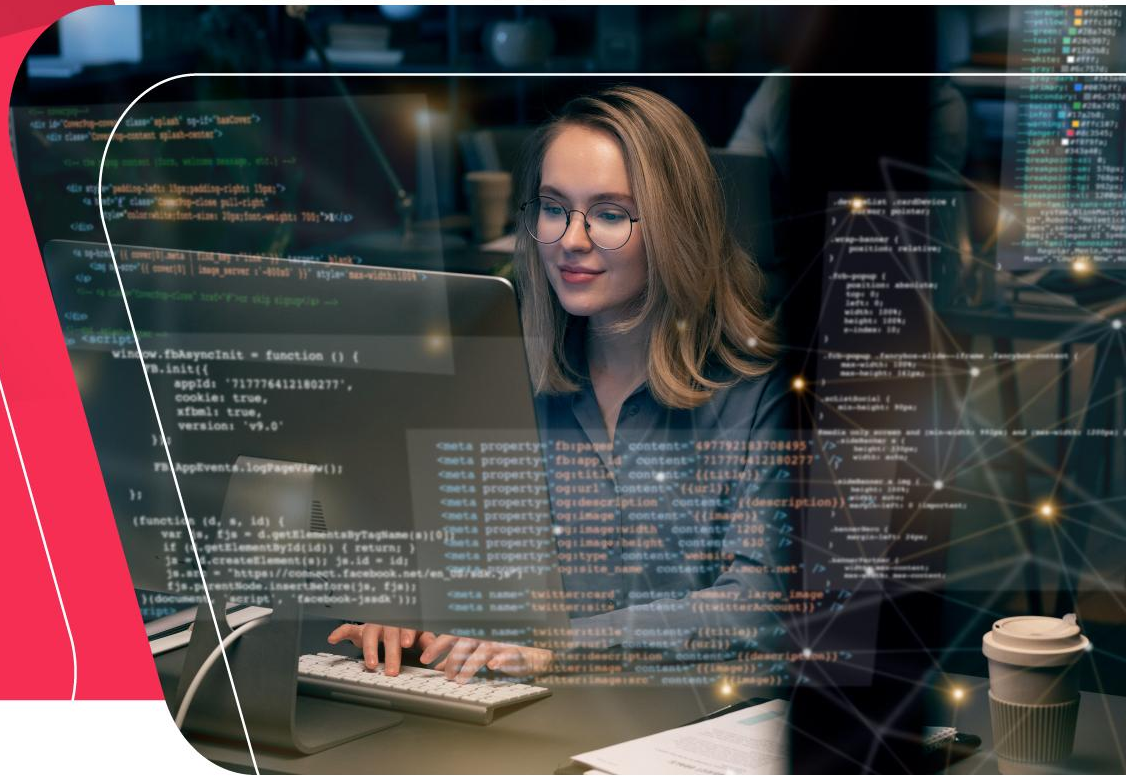
TIA-Academy

Java Bootcamp

Day 18



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- Writing Code using Notepad++ (For Windows) or Vim (For Linux and Mac)
- Compiling with Command Prompt (For Windows) or Terminal (For Linux and Mac)
- ***Forbidden of using IDE Based Development Tools as of Intellij IDEA or Eclipse***

TIA-Academy

Java Abstract

Java Abstract Class

- The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

Java Abstract Class

- An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Java Abstract Method

- A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. For example,

```
abstract void display();
```

- Here, display() is an abstract method. The body of display() is replaced by ;.
- If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

  // abstract method
  abstract void method1();
}
```

Java Abstract Class and Method

- Though abstract classes cannot be instantiated, we can create subclasses from it.
- We can then access members of the abstract class using the object of the subclass. For example,
- Output:

```
This is Java programming
```

```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java
Programming");
  }
}

class Main extends Language {
  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

Java Abstract Class and Method

- In the above example, we have created an abstract class named Language. The class contains a regular method display().
- We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```

- Here, obj is the object of the child class Main. We are calling the method of the abstract class using the object obj.

Implementing Abstract Methods

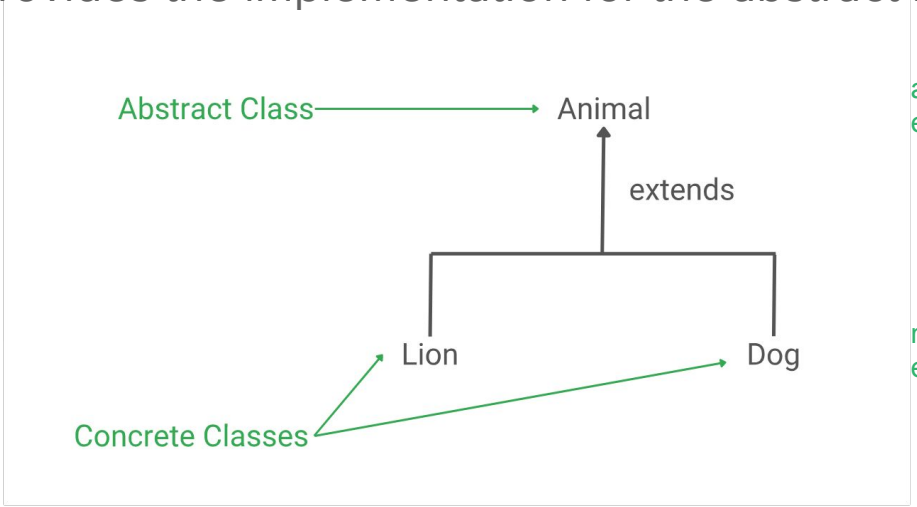
- If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,
- Output:

```
Bark bark
I can eat.
```

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Implementing Abstract Methods

- In the above example, we have created an abstract class Animal. The class contains an abstract method makeSound() and a non-abstract method eat().
- We have inherited a subclass Dog from the superclass Animal. Here, the subclass Dog provides the implementation for the abstract method makeSound().



Implementing Abstract Methods

- We then used the object d1 of the Dog class to call methods makeSound() and eat().
- Note: If the Dog class doesn't provide the implementation of the abstract method makeSound(), Dog should also be declared as abstract. This is because the subclass Dog inherits makeSound() from Animal.

Java Abstraction

- The major use of abstract classes and methods is to achieve abstraction in Java.
- Abstraction is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information.
- This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

Java Abstraction

- A practical example of abstraction can be motorbike brakes. We know what brake does. When we apply the brake, the motorbike will stop. However, the working of the brake is kept hidden from us.
- The major advantage of hiding the working of the brake is that now the manufacturer can implement brake differently for different motorbikes, however, what brake does will be the same.

Java Abstraction

- Example 3: Java Abstraction
- Output:

```
MountainBike Brake
SportsBike Brake
```

```
abstract class MotorBike {
  abstract void brake();
}

class SportsBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("SportsBike Brake");
  }
}

class MountainBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("MountainBike Brake");
  }
}

class Main {
  public static void main(String[] args) {
    MountainBike m1 = new MountainBike();
    m1.brake();
    SportsBike s1 = new SportsBike();
    s1.brake();
  }
}
```

Java Abstraction

- In the above example, we have created an abstract super class MotorBike. The superclass MotorBike has an abstract method brake().
- The brake() method cannot be implemented inside MotorBike. It is because every bike has different implementation of brakes. So, all the subclasses of MotorBike would have different implementation of brake().
- So, the implementation of brake() in MotorBike is kept hidden.
- Here, MountainBike makes its own implementation of brake() and SportsBike makes its own implementation of brake().
- Note: We can also use interfaces to achieve abstraction in Java.

Key Points to Remember

- We use the abstract keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- *We can access the static attributes and methods of an abstract class using the reference of the abstract class. For example,*

```
Animal.staticMethod();
```

TIA-Academy

ASSIGNMENT 01

TIA-Academy

Java Interface

Java Interface

- An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).
- We use the interface keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

- Here,
  - Language is an **interface**.
  - It includes abstract methods: **getType() and getVersion().**

Implementing an Interface

- Like abstract classes, we cannot create objects of interfaces.
- To use an interface, other classes must implement it.
- We use the **implements keyword** to implement an interface.

Implementing an Interface

- Example 1: Java Interface
- Output :

```
The area of the rectangle is 30
```

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Implementing an Interface

- In the above example, we have created an interface named Polygon. The interface contains an abstract method getArea().
- Here, the Rectangle class implements Polygon. And, provides the implementation of the getArea() method.

Implementing an Interface

- Example 2: Java Interface
- Output :

```
Programming Language: Java
```

```
// create an interface
interface Language {
  void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Implementing an Interface

- In the above example, we have created an interface named Language. The interface includes an abstract method getName().
- Here, the ProgrammingLanguage class implements the interface and provides the implementation for the method.

Implementing Multiple Interface

- In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Extending an Interface

- Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

- Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

Extending Multiple Interface

- An interface can extend multiple interfaces. For example,

```
interface A {
    ...
}

interface B {
    ...
}

interface C extends A, B {
    ...
}
```

Advantages of Interface in Java

Similar to abstract classes, interfaces help us to achieve abstraction in Java.

- Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.

Interfaces provide specifications that a class (which implements it) must follow.

- In our previous example, we have used getArea() as a specification inside the interface Polygon. This is like setting a rule that we should be able to get the area of every polygon.
- Now any class that implements the Polygon interface must provide an implementation for the getArea() method.

Advantages of Interface in Java

- Interfaces are also used to achieve multiple inheritance in Java. For example,

```
interface Line {
…
}

interface Polygon {
…
}

class Rectangle implements Line, Polygon {
…
}
```

- Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

Advantages of Interface in Java

- All the methods inside an interface are implicitly public and all fields are implicitly public static final. For example,

```
interface Language {

  // by default public static final
  String type = "programming language";

  // by default public
  void getName();
}
```

default methods in Java Interfaces

- With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.
- To declare default methods inside interfaces, we use the **default** keyword. For example,

```
public default void getSides() {
    // body of getSides()
}
```

Why default methods?

- Let's take a scenario to understand why default methods are introduced in Java.
- Suppose, we need to add a new method in an interface.
- We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.
- If **a large number of classes were implementing this interface**, **we need to track all these classes and make changes to them**. This is not only tedious but error-prone as well.
- To resolve this, Java **introduced default methods**. Default methods are inherited like ordinary methods.

Why default methods?

- Let's take an example to have a better understanding of default methods.

```
interface Polygon {
  void getArea();

  // default method
  default void getSides() {
    System.out.println("I can get sides of a polygon.");
  }
}

// implements the interface
class Rectangle implements Polygon {
  public void getArea() {
    int length = 6;
    int breadth = 5;
    int area = length * breadth;
    System.out.println("The area of the rectangle is " + area);
  }

  // overrides the getSides()
  public void getSides() {
    System.out.println("I have 4 sides.");
  }
}
```

Why default methods?

- Output :

```
The area of the rectangle is 30
I have 4 sides.
The area of the square is 25
I can get sides of a polygon.
```

```
// implements the interface
class Square implements Polygon {
  public void getArea() {
    int length = 5;
    int area = length * length;
    System.out.println("The area of the square is " + area);
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Rectangle
    Rectangle r1 = new Rectangle();
    r1.getArea();
    r1.getSides();

    // create an object of Square
    Square s1 = new Square();
    s1.getArea();
    s1.getSides();
  }
}
```

Why default methods?

- In the above example, we have created an interface named Polygon. It has a default method getSides() and an abstract method getArea().
- Here, we have created two classes Rectangle and Square that implement Polygon.
- The Rectangle class provides the implementation of the getArea() method and overrides the getSides() method. However, the Square class only provides the implementation of the getArea() method.
- Now, while calling the getSides() method using the Rectangle object, the overridden method is called. However, in the case of the Square object, the default method is called.

Practical Example of Interface

```
// To use the sqrt function
import java.lang.Math;

interface  Polygon {
   void getArea();

 // calculate the perimeter of a Polygon
   default void getPerimeter(int... sides) {
      int perimeter = 0;
      for (int side: sides) {
         perimeter += side;
      }

   System.out.println("Perimeter: " + perimeter);
   }
}

class Triangle implements Polygon {
   private int a, b, c;
   private double s, area;

   // initializing sides of a triangle
   Triangle(int a, int b, int c) {
      this.a = a;
      this.b = b;
      this.c = c;
      s = 0;
   }

   // calculate the area of a triangle
   public void getArea() {
      s = (double) (a + b + c)/2;
      area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
      System.out.println("Area: " + area);
   }
}
```

Practical Example of Interface

- Output :

```
Area: 2.9047375096555625
Perimeter: 9
```

```
class Main {
   public static void main(String[] args) {
      Triangle t1 = new Triangle(2, 3, 4);

      // calls the method of the Triangle class
      t1.getArea();

      // calls the method of Polygon
      t1.getPerimeter(2, 3, 4);
   }
}
```

Practical Example of Interface

- In the above program, we have created an interface named Polygon. It includes a default method getPerimeter() and an abstract method getArea().
- We can calculate the perimeter of all polygons in the same manner so we implemented the body of getPerimeter() in Polygon.
- Now, all polygons that implement Polygon can use getPerimeter() to calculate perimeter.
- However, the rule for calculating the area is different for different polygons. Hence, getArea() is included without implementation.
- Any class that implements Polygon must provide an implementation of getArea().

TIA-Academy

Java Polymorphism

Java Polymorphism

- Polymorphism is an important concept of object-oriented programming. **It simply means more than one form**.
- That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Java Polymorphism

```
class Polygon {

  // method to render a shape
  public void render() {
    System.out.println("Rendering Polygon...");
  }
}

class Square extends Polygon {

  // renders Square
  public void render() {
    System.out.println("Rendering Square...");
  }
}

class Circle extends Polygon {

  // renders circle
  public void render() {
    System.out.println("Rendering Circle...");
  }
}
```

Java Polymorphism

- Output :

```
Rendering Square...
Rendering Circle...
```

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

    // create an object of Square
    Square s1 = new Square();
    s1.render();

    // create an object of Circle
    Circle c1 = new Circle();
    c1.render();
  }
}
```

Java Polymorphism

- In the above example, we have created a superclass: **Polygon** and two subclasses: **Square and Circle**. Notice the use of the render() method.
- The main purpose of the render() method is to render the **shape**. However, the process of rendering a square is different than the process of rendering a circle.
- Hence, the **render() method behaves differently in different classes**. Or, we can say **render() is polymorphic**.

Java Method Overriding

- During inheritance in Java, if the same method is present in both the superclass and the subclass. Then, the **method in the subclass overrides the same method in the superclass**. This is called **method overriding.**
- In this case, the same method will perform one operation in the superclass and another operation in the subclass. For example,

Java Polymorphism

- Example 1: Polymorphism using method overriding
- Output :

```
Java Programming Language
Common English Language
```

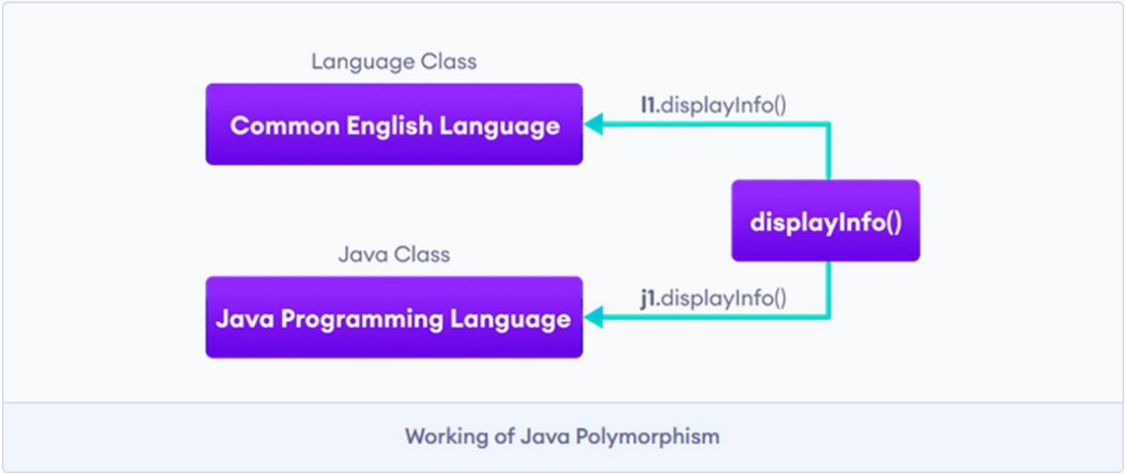
```
class Language {
  public void displayInfo() {
    System.out.println("Common English Language");
  }
}

class Java extends Language {
  @Override
  public void displayInfo() {
    System.out.println("Java Programming Language");
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Java class
    Java j1 = new Java();
    j1.displayInfo();

    // create an object of Language class
    Language l1 = new Language();
    l1.displayInfo();
  }
}
```

Java Method Overriding

- In the above example, we have created a superclass named **Language** and a subclass named **Java**. Here, the method **displayInfo()** is present in both Language and Java.
- The use of displayInfo() is to print the information. However, it is printing different information in Language and Java.
- **Based on the object used to call the method**, the corresponding information is printed.

Java Method Overriding

- **Note**: The method that is called is determined during the execution of the program. Hence, **method overriding** is a **run-time polymorphism**.



Working of Java Polymorphism

Java Method Overloading

- In a Java class, we can create methods with the same name if they differ in parameters. For example,

```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

- This is known as method overloading in Java. Here, the same method will perform different operations based on the parameter.

Java Method Overloading

- Example 3: Polymorphism using method overloading
- Output :

```
**********

##########
```

```
class Pattern {

  // method without parameter
  public void display() {
    for (int i = 0; i < 10; i++) {
      System.out.print("*");
    }
  }

  // method with single parameter
  public void display(char symbol) {
    for (int i = 0; i < 10; i++) {
      System.out.print(symbol);
    }
  }
}

class Main {
  public static void main(String[] args) {
    Pattern d1 = new Pattern();

    // call method without any argument
    d1.display();
    System.out.println("\n");

    // call method with a single argument
    d1.display('#');
  }
}
```

Java Method Overloading

- In the above example, we have created a class named Pattern. The class contains a method named display() that is overloaded.

```
// method with no arguments
display() {...}

// method with a single char type argument
display(char symbol) {...}
```

Java Method Overloading

- Here, the main function of **display()** is to print the pattern. However, based on the arguments passed, the method is performing different operations:
  - prints a pattern of *, if **no argument is passed** or
  - prints pattern of the parameter, **if a single char type argument is passed**.
- Note: The method that is called is determined by the compiler. Hence, it is also known as **compile-time polymorphism**.

TIA-Academy

Java Encapsulation

Java Encapsulation

- Encapsulation is one of the key features of object-oriented programming. **Encapsulation refers to the bundling of fields and methods inside a single class.**
- It **prevents outer classes from accessing and changing fields** and methods of a class. This also helps to achieve **data hiding**.

Java Encapsulation

- Example 1: Java Encapsulation
- Output :

```
Area: 30
```

```
class Area {

  // fields to calculate area
  int length;
  int breadth;

  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }

  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}

class Main {
  public static void main(String[] args) {

    // create object of Area
    // pass value of length and breadth
    Area rectangle = new Area(5, 6);
    rectangle.getArea();
  }
}
```

Java Encapsulation

- In the above example, we have created a class named Area. The main purpose of this class is to calculate the area.
- To calculate an area, we need two variables: length and breadth and a method: getArea(). Hence, we bundled these fields and methods inside a single class.
- Here, the fields and methods can be accessed from other classes as well.

  Hence, this is not data hiding.

Java Encapsulation

- People often consider encapsulation as data hiding, but that's not entirely true.
- Encapsulation refers to the bundling of related fields and methods together. This can be used to achieve data hiding. **Encapsulation in itself is not data hiding**.

Why Encapsulation?

- In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.
- It helps to control the values of our data fields. For example,

```
class Person {
  private int age;

  public void setAge(int age) {
    if (age >= 0) {
      this.age = age;
    }
  }
}
```

Here, we are making the age variable **private** and applying logic inside the **setAge()** method. Now, age cannot be negative.

Why Encapsulation?

- The getter and setter methods **provide read-only or write-only** access to our class fields. For example,

```
getName()  // provides read-only access
setName() // provides write-only access
```

- It helps to decouple components of a system. For example, we can encapsulate code into multiple bundles.
  - These decoupled components (bundle) can be developed, tested, and debugged independently and concurrently. And, any changes in a particular component do not have any effect on other components.
- We can also achieve data hiding using encapsulation. In the above example, if we change the length and breadth variable into private, then the access to these fields is restricted.
  - And, they are kept hidden from outer classes. This is called data hiding.

Data Hiding

- Data hiding is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.
- We can use access modifiers to achieve data hiding.

Data Hiding

- Example 2: Data hiding using the private specifier
- Output :

```
My age is 24
```

```
class Person {

  // private field
  private int age;

  // getter method
  public int getAge() {
    return age;
  }

  // setter method
  public void setAge(int age) {
    this.age = age;
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Person
    Person p1 = new Person();

    // change age using setter
    p1.setAge(24);

    // access age using getter
    System.out.println("My age is " + p1.getAge());
  }
}
```

Data Hiding

- In the above example, we have a private field age. Since it is private, it cannot be accessed from outside the class.
- In order to access age, we have used public methods: getAge() and setAge(). These methods are called getter and setter methods.
- Making age private allowed us to restrict unauthorized access from outside the class. This is data hiding.
- If we try to access the age field from the Main class, we will get an error.

```
// error: age has private access in Person
p1.age = 24;
```

TIA-Academy

ASSIGNMENT 02
(HOME ASSIGNMENT)

TIA-Academy

Thank You