Java Inheritance

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.

The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).

The `extends` keyword is used to perform inheritance in Java. For example,

```
class Animal {
  // methods and fields
}

// use of extends keyword
// to perform inheritance
class Dog extends Animal {

  // methods and fields of Animal
  // methods and fields of Dog
}
```

In the above example, the Dog class is created by inheriting the methods and fields from the Animal class.

Here, Dog is the subclass and Animal is the superclass.

Example 1: Java Inheritance

```
class Animal {

  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }
}

// inherit from Animal
class Dog extends Animal {

  // new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }
}
```

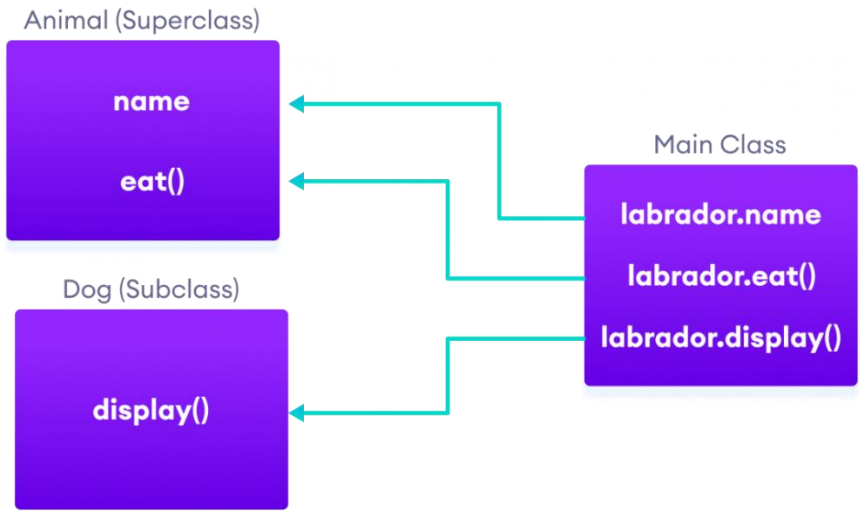
```
class Main {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();

    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();

    // call method of superclass
    // using object of subclass
    labrador.eat();

  }
}
```

Output

```
My name is Rohu
I can eat
```

Here, labrador is an object of Dog. However, name and eat() are the members of the Animal class.

Since Dog inherits the field and method from Animal, we are able to access the field and method using the object of the Dog.



Java Inheritance Implementation

is-a relationship

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Method Overriding in Java Inheritance

In **Example 1**, we see the object of the subclass can access the method of the superclass.

However, if the same method is present in both the superclass and subclass, what will happen?

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Java.

Example 2: Method overriding in Java Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
class Dog extends Animal {

  // overriding the eat() method
  @Override
  public void eat() {
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
```

```
}

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

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

Output

```
I eat dog food
I can bark
```

In the above example, the eat() method is present in both the superclass Animal and the subclass Dog.

Here, we have created an object labrador of Dog.

Now when we call eat() using the object labrador, the method inside Dog is called. This is because the method inside the derived class overrides the method inside the base class.

This is called method overriding.

Note: We have used the @Override annotation to tell the compiler that we are overriding a method. However, the annotation is not mandatory.

super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass.

In such a situation, the super keyword is used to call the method of the parent class from the method of the child class.

Example 3: super Keyword in Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
class Dog extends Animal {

  // overriding the eat() method
  @Override
  public void eat() {

    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}

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

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

Output

```
I can eat
I eat dog food
I can bark
```

In the above example, the eat() method is present in both the base class Animal and the derived class Dog. Notice the statement,

```
super.eat();
```

Here, the super keyword is used to call the eat() method present in the superclass.

We can also use the super keyword to call the constructor of the superclass from the constructor of the subclass.

protected Members in Inheritance

In Java, if a class includes protected fields and methods, then these fields and methods are accessible from the subclass of the class.

Example 4: protected Members in Inheritance

```
class Animal {
  protected String name;

  protected void display() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal {

  public void getInfo() {
    System.out.println("My name is " + name);
  }
}

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

    // create an object of the subclass
    Dog labrador = new Dog();

    // access protected field and method
    // using the object of subclass
    labrador.name = "Rocky";
    labrador.display();

    labrador.getInfo();
  }
}
```

Output

```
I am an animal.
My name is Rocky
```

In the above example, we have created a class named Animal. The class includes a protected field: name and a method: display().

We have inherited the Dog class inherits Animal. Notice the statement,

```
labrador.name = "Rocky";
labrador.display();
```

Here, we are able to access the protected field and method of the superclass using the labrador object of the subclass.

Why use inheritance?

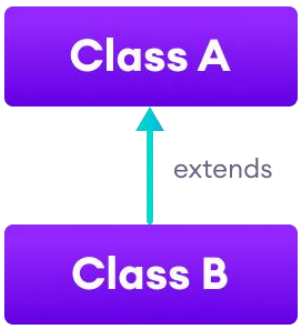
- The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.
- Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

Types of inheritance

There are five types of inheritance.

1. Single Inheritance

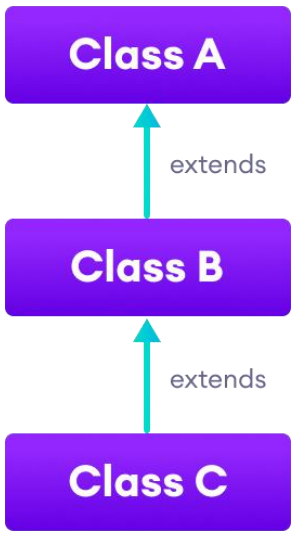
In single inheritance, a single subclass extends from a single superclass. For example,



Java Single Inheritance

2. Multilevel Inheritance

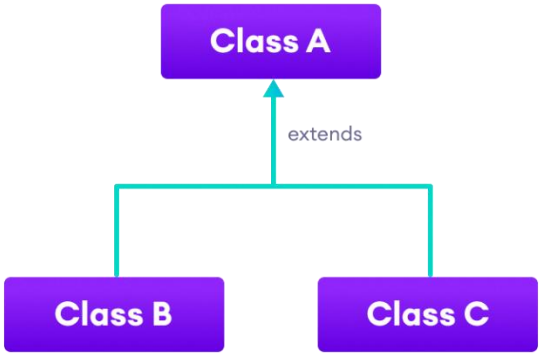
In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,



Java Multilevel Inheritance

3. Hierarchical Inheritance

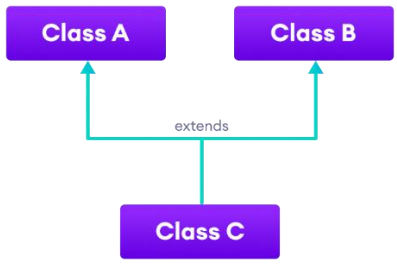
In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



Java Hierarchical Inheritance

4. Multiple Inheritance

In multiple inheritance, a single subclass extends from multiple superclasses. For example,

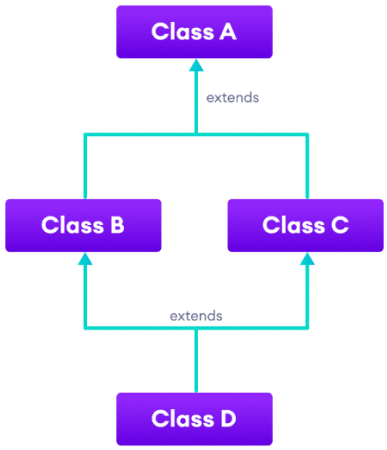


Java Multiple Inheritance

Note: Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces.

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example,



Java Hybrid Inheritance

Here, we have combined hierarchical and multiple inheritance to form a hybrid inheritance.

Java Method Overriding

In the last tutorial, we learned about inheritance. Inheritance is an OOP property that allows us to derive a new class (subclass) from an existing class (superclass). The subclass inherits the attributes and methods of the superclass.

Now, if the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method overriding.

Example 1: Method Overriding

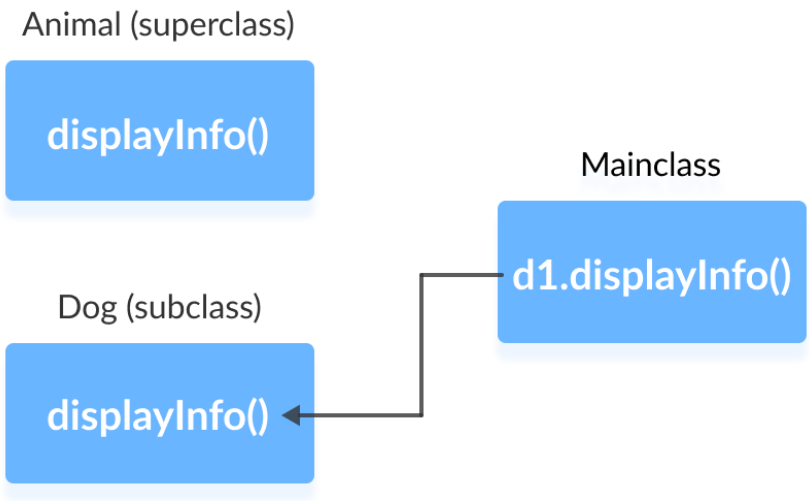
```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am a dog.
```

In the above program, the displayInfo() method is present in both the *Animal* superclass and the *Dog* subclass.

When we call displayInfo() using the *d1* object (object of the subclass), the method inside the subclass *Dog* is called. The displayInfo() method of the subclass overrides the same method of the superclass.



Notice the use of the @Override annotation in our example. In Java, annotations are the metadata that we used to provide information to the compiler. Here, the @Override annotation specifies the compiler that the method after this annotation overrides the method of the superclass.

It is not mandatory to use @Override. However, when we use this, the method should follow all the rules of overriding. Otherwise, the compiler will generate an error.

Java Overriding Rules

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as final and static.
- We should always override abstract methods of the superclass (will be discussed in later tutorials).

super Keyword in Java Overriding

A common question that arises while performing overriding in Java is:

Can we access the method of the superclass after overriding?

Well, the answer is **Yes**. To access the method of the superclass from the subclass, we use the `super` keyword.

Example 2: Use of super Keyword

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      super.displayInfo();
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am an animal.
I am a dog.
```

In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal.

When we call the method displayInfo() using the d1 object of the Dog subclass, the method inside the Dog subclass is called; the method inside the superclass is not called.

Inside displayInfo() of the Dog subclass, we have used super.displayInfo() to call displayInfo() of the superclass.

It is important to note that constructors in Java are not inherited. Hence, there is no such thing as constructor overriding in Java.

However, we can call the constructor of the superclass from its subclasses.

Access Specifiers in Method Overriding

The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.

We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,

Suppose, a method myClass() in the superclass is declared protected. Then, the same method myClass() in the subclass can be either public or protected, but not private.

Example 3: Access Specifier in Overriding

```
class Animal {
   protected void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am a dog.
```

In the above example, the subclass *Dog* overrides the method displayInfo() of the superclass *Animal*.

Whenever we call displayInfo() using the *d1* (object of the subclass), the method inside the subclass is called.

Notice that, the displayInfo() is declared protected in the *Animal* superclass. The same method has the public access specifier in the *Dog* subclass. This is possible because the public provides larger access than the protected.

Overriding Abstract Methods

In Java, abstract classes are created to be the superclass of other classes. And, if a class contains an abstract method, it is mandatory to override it.

Java super

The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

Uses of super keyword

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

1. Access Overridden Methods of the superclass

If methods with the same name are defined in both superclass and subclass, the method in the subclass overrides the method in the superclass. This is called method overriding.

Example 1: Method overriding

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
```

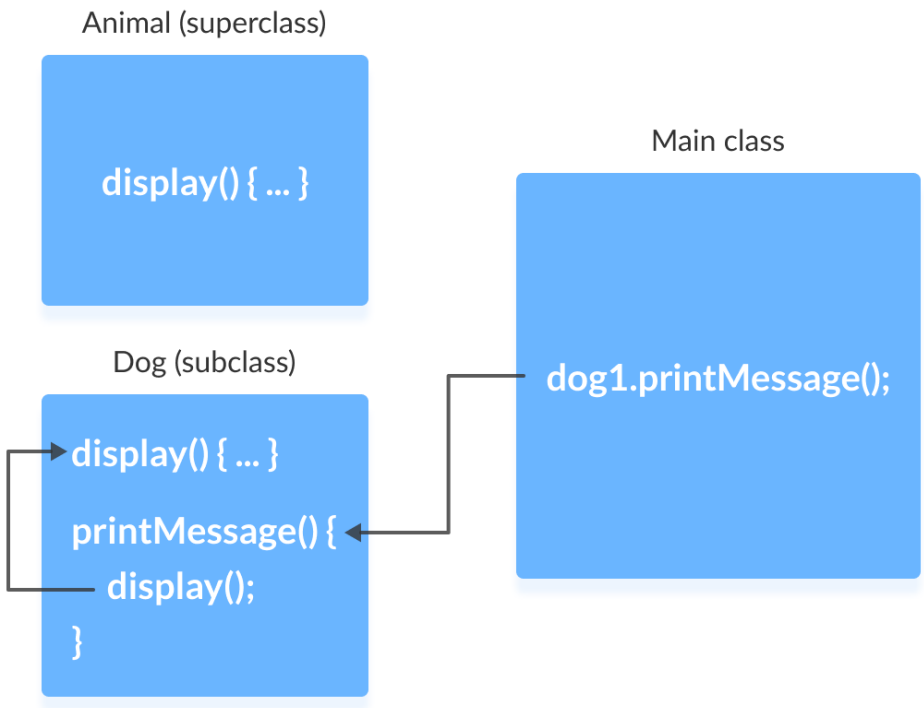
```
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Output

```
I am a dog
```

In this example, by making an object dog1 of Dog class, we can call its method printMessage() which then executes the display() statement.

Since display() is defined in both the classes, the method of subclass Dog overrides the method of superclass Animal. Hence, the display() of the subclass is called.



What if the overridden method of the superclass has to be called?

We use super.display() if the overridden method display() of superclass Animal needs to be called.

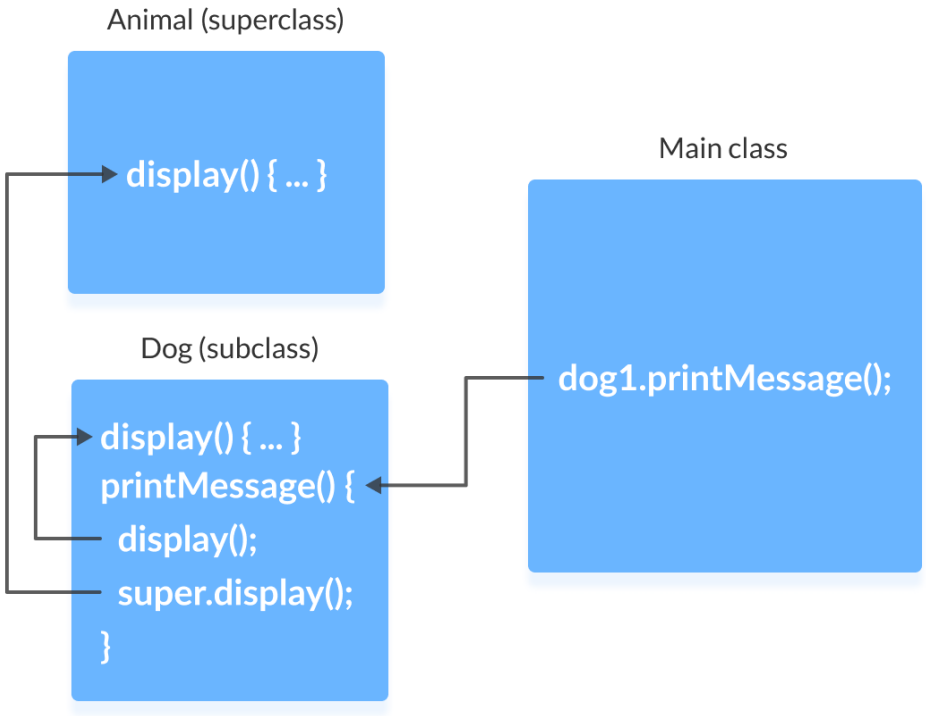
Example 2: super to Call Superclass Method

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Output

```
I am a dog
I am an animal
```

Here, how the above program works.



2. Access Attributes of the Superclass

The superclass and subclass can have attributes with the same name. We use the super keyword to access the attribute of the superclass.

Example 3: Access superclass attribute

```
class Animal {
  protected String type="animal";
}

class Dog extends Animal {
  public String type="mammal";

  public void printType() {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

Output:

```
I am a mammal
I am an animal
```

In this example, we have defined the same instance field *type* in both the superclass *Animal* and the subclass *Dog*.

We then created an object *dog1* of the *Dog* class. Then, the printType() method is called using this object.

Inside the printType() function,

- *type* refers to the attribute of the subclass *Dog*.
- *super.type* refers to the attribute of the superclass Animal.

Hence, System.out.println("I am a " + type); prints I am a mammal. And, System.out.println("I am an " + super.type); prints I am an animal.

3. Use of super() to access superclass constructor

As we know, when an object of a class is created, its default constructor is automatically called.

To explicitly call the superclass constructor from the subclass constructor, we use super(). It's a special form of the super keyword.

super() can be used only inside the subclass constructor and must be the first statement.

Example 4: Use of super()

```
class Animal {

  // default or no-arg constructor of class Animal
  Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // default or no-arg constructor of class Dog
  Dog() {
```

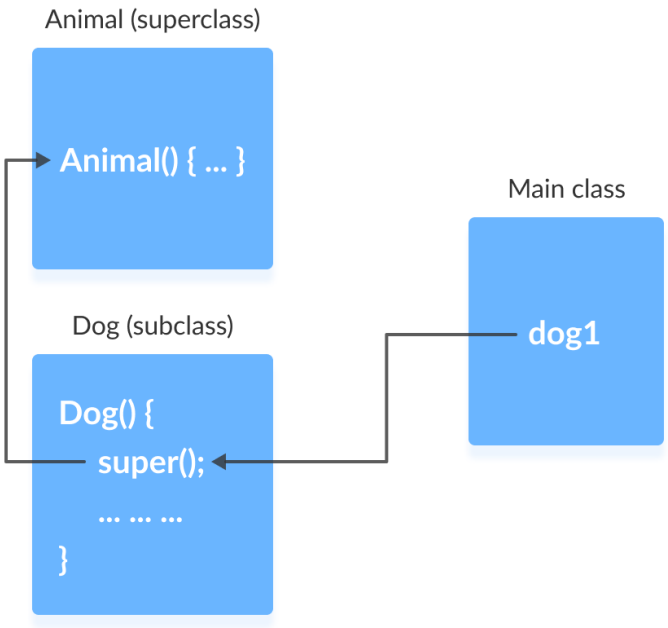
```
    // calling default constructor of the superclass
    super();

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Output

```
I am an animal
I am a dog
```

Here, when an object dog1 of Dog class is created, it automatically calls the default or no-arg constructor of that class.

Inside the subclass constructor, the super() statement calls the constructor of the superclass and executes the statements inside it. Hence, we get the output I am an animal.



The flow of the program then returns back to the subclass constructor and executes the remaining statements. Thus, I am a dog will be printed.

However, using super() is not compulsory. Even if super() is not used in the subclass constructor, the compiler implicitly calls the default constructor of the superclass.

So, why use redundant code if the compiler automatically invokes super()?

It is required if the parameterized constructor (a constructor that takes arguments) of the superclass has to be called from the subclass constructor.

The parameterized super() must always be the first statement in the body of the constructor of the subclass, otherwise, we get a compilation error.

Example 5: Call Parameterized Constructor Using super()

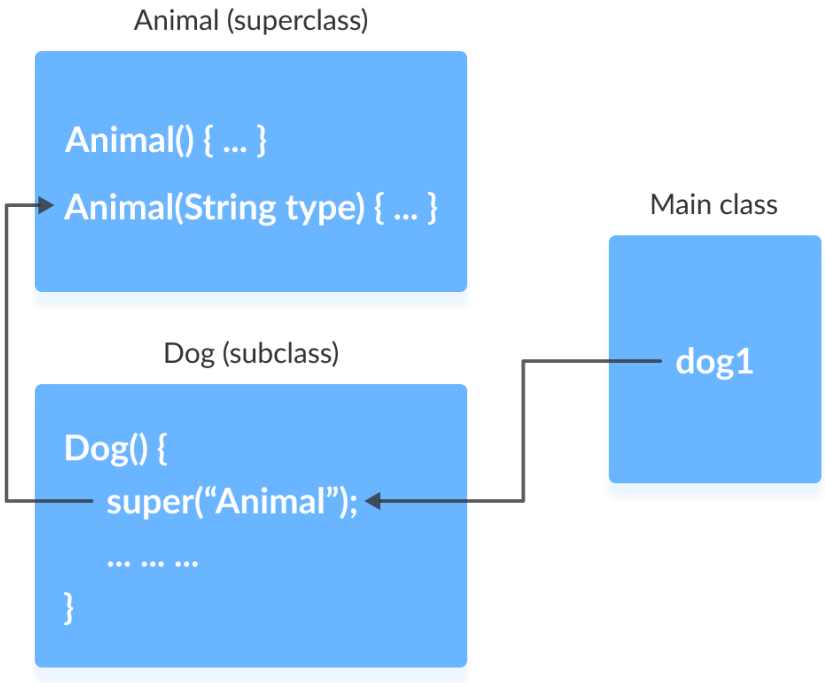
```
class Animal {

  // default or no-arg constructor
  Animal() {
    System.out.println("I am an animal");
  }

  // parameterized constructor
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}

class Dog extends Animal {

  // default constructor
  Dog() {

    // calling parameterized constructor of the superclass
    super("Animal");

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Output

```
Type: Animal
I am a dog
```

The compiler can automatically call the no-arg constructor. However, it cannot call parameterized constructors.

If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.



Note that in the above example, we explicitly called the parameterized constructor super("Animal"). The compiler does not call the default constructor of the superclass in this case.

Java Abstract Class and Abstract Methods

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();
```

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();
}
```

Example: 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,

```
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();
  }
}
```

Output

```
This is Java programming
```

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,

```
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();
  }
}
```

Output

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

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().

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.

Accesses Constructor of Abstract Classes

An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the super keyword. For example,

```
abstract class Animal {
   Animal() {
```

```
        ….
    }
}

class Dog extends Animal {
    Dog() {
        super();
        ...
    }
}
```

Here, we have used the super() inside the constructor of *Dog* to access the constructor of the *Animal*.

Note that the super should always be the first statement of the subclass constructor.

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.

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.

Let's take an example that helps us to better understand Java abstraction.

Example 3: Java Abstraction

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

}

class Dog extends Animal {

  // implementation of abstract method
```

```
  public void makeSound() {
    System.out.println("Bark bark.");
  }
}

class Cat extends Animal {

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

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

    Cat c1 = new Cat();
    c1.makeSound();
  }
}
```

Output:

```
Bark bark
Meows
```

In the above example, we have created a superclass Animal. The superclass Animal has an abstract method makeSound().

The makeSound() method cannot be implemented inside Animal. It is because every animal makes different sounds. So, all the subclasses of Animal would have different implementation of makeSound().

So, the implementation of makeSound() in Animal is kept hidden.

Here, Dog makes its own implementation of makeSound() and Cat makes its own implementation of makeSound().

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();
  ```

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.

Example 1: Java Interface

```
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);
  }
}
```

Output

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

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.

Example 2: Java Interface

```
// 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");
  }
}
```

Output

```
Programming Language: Java
```

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 Interfaces

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 Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
   ...
}
interface B {
   ...
}

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

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used 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.
- 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. **Note**: 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.

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

Example: Default Method in Java Interface

```
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.");
  }
}

// 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();
  }
}
```

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.
```

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.

private and static Methods in Interface

The Java 8 also added another feature to include static methods inside an interface.

Similar to a class, we can access static methods of an interface using its references. For example,

```
// create an interface
interface Polygon {
  staticMethod(){..}
}

// access static method
Polygon.staticMethod();
```

Note: With the release of Java 9, private methods are also supported in interfaces.

We cannot create objects of an interface. Hence, private methods are used as helper methods that provide support to other methods in interfaces.

Practical Example of Interface

Let's see a more practical example of Java 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);
   }
}

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);
```

```
    }
}
```

Output

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

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().

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.

Example: 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...");
  }
}

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();
  }
}
```

Output

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

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.

Why Polymorphism?

Polymorphism allows us to create consistent code. In the previous example, we can also create different methods: renderSquare() and renderCircle() to render *Square* and *Circle*, respectively.

This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.

To solve this, polymorphism in Java allows us to create a single method render() that will behave differently for different shapes.

Note: The print() method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc

We can achieve polymorphism in Java using the following ways:

1. Method Overriding
2. Method Overloading
3. Operator Overloading

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,

Example 1: Polymorphism using method overriding

```
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();
  }
}
```

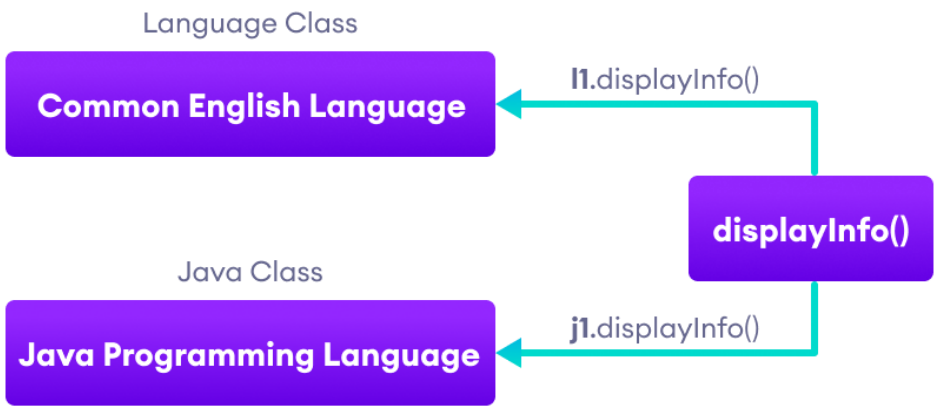
Output:

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

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.



Working of Java Polymorphism

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

2. 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.

Example 3: Polymorphism using method overloading

```
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('#');
  }
}
```

Output:

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

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

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) {...}
```

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.

3. Java Operator Overloading

Some operators in Java behave differently with different operands. For example,

- + operator is overloaded to perform numeric addition as well as string concatenation, and
- operators like &, |, and ! are overloaded for logical and bitwise operations.

Let's see how we can achieve polymorphism using operator overloading.

The + operator is used to add two entities. However, in Java, the + operator performs two operations.

1. When + is used with numbers (integers and floating-point numbers), it performs mathematical addition. For example,

```
int a = 5;
int b = 6;

// + with numbers
int sum = a + b;  // Output = 11
```

2. When we use the + operator with strings, it will perform string concatenation (join two strings). For example,

```
String first = "Java ";
String second = "Programming";

// + with strings
name = first + second;  // Output = Java Programming
```

Here, we can see that the + operator is overloaded in Java to perform two operations: **addition** and **concatenation**.

Note: In languages like C++, we can define operators to work differently for different operands. However, Java doesn't support user-defined operator overloading.

Polymorphic Variables

A variable is called polymorphic if it refers to different values under different conditions.

Object variables (instance variables) represent the behavior of polymorphic variables in Java. It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

Example: Polymorphic Variables

```
class ProgrammingLanguage {
  public void display() {
    System.out.println("I am Programming Language.");
  }
}

class Java extends ProgrammingLanguage {
  @Override
  public void display() {
    System.out.println("I am Object-Oriented Programming Language.");
  }
}

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

    // declare an object variable
    ProgrammingLanguage pl;

    // create object of ProgrammingLanguage
    pl = new ProgrammingLanguage();
    pl.display();
```

```
    // create object of Java class
    pl = new Java();
    pl.display();
  }
}
```

Output:

```
I am Programming Language.
I am Object-Oriented Programming Language.
```

In the above example, we have created an object variable *pl* of the *ProgrammingLanguage* class. Here, *pl* is a polymorphic variable. This is because,

- In statement `pl = new ProgrammingLanguage()`, *pl* refer to the object of the *ProgrammingLanguage* class.
- And, in statement `pl = new Java()`, *pl* refer to the object of the *Java* class.

This is an example of upcasting in Java.

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**.

Example 1: Java Encapsulation

```
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();
  }
}
```

Output

```
Area: 30
```

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**.

This is only **encapsulation**. We are just keeping similar codes together.

Note: 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.

 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. For example,

Example 2: Data hiding using the private specifier

```
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());
  }
}
```

Output

```
My age is 24
```

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;
```