

**Lab Manual- Java Inheritance , Polymorphism and Abstract class**

**Prepared for**:

**Date:** 18th June 2024

**Prepared by:**

Document Name: Lab Manual **Document Number** SDLab333

**Contributor:**

Contents

[1. What is Inheritance 3](#_Toc172435451)

[2. Inheritance Example 3](#_Toc172435452)

[1. Define the Superclass 3](#_Toc172435453)

[2. Define the Subclass 4](#_Toc172435454)

[3. Using the Subclass 5](#_Toc172435455)

[4. Key Points 6](#_Toc172435456)

[5. Compile and Run 6](#_Toc172435457)

[3. Instanceof and getClass() 7](#_Toc172435458)

[Simplified Example 7](#_Toc172435459)

[1. Create Main1.java 7](#_Toc172435460)

[2. Compile and Run Main1.java 8](#_Toc172435461)

[4. Polymorphism 8](#_Toc172435462)

[1. Animal.java 9](#_Toc172435463)

[2. Dog.java 10](#_Toc172435464)

[3. Cat.java 10](#_Toc172435465)

[4. Cow.java 10](#_Toc172435466)

[5. AnimalMain.java 10](#_Toc172435467)

[6. Steps to Compile and Run 11](#_Toc172435468)

[5. Abstract Class 11](#_Toc172435469)

[6. Encapsulation 13](#_Toc172435470)

[1. Concept 13](#_Toc172435471)

[2. TV Remote Example 14](#_Toc172435472)

[2.1 How TV Remote Works: 14](#_Toc172435473)

[2.2 Encapsulation in Action with TV Remote: 14](#_Toc172435474)

[3. Book.java and ExampleBooks.java 15](#_Toc172435475)

[3.1 Book.java 15](#_Toc172435476)

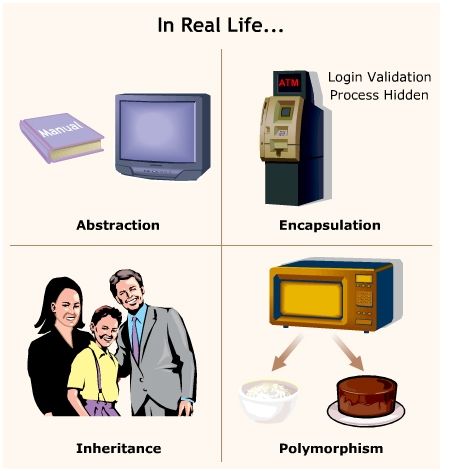
[3.2 .1 Explanation of Encapsulation 16](#_Toc172435477)

[3.3 ExampleBooks.java 16](#_Toc172435478)

[3.4 Compile and Run 17](#_Toc172435479)

# What is Inheritance

Inheritance allows a new class (called a subclass or derived class) to inherit fields and methods from an existing class (called a superclass or base class). The subclass can add new fields and methods or override existing ones.



# Inheritance Example

## Define the Superclass

First, let's define the **Book** class that the **ExampleBooks** will extend. We already have the Book class from the previous example:

**Book.java**

public class Book {

    private String title;

    private String author;

    private int pages;

    private String isbn;

    private static int bookCount = 0;

    public Book(String title, String author, int pages, String isbn) {

        this.title = title;

        this.author = author;

        this.pages = pages;

        this.isbn = isbn;

        bookCount++;

    }

    public Book(String title, String author, int pages) {

        this(title, author, pages, "");

    }

    public static int getBookCount() {

        return bookCount;

    }

    public String getTitle() {

        return title;

    }

    public String getAuthor() {

        return author;

    }

    public int getPages() {

        return pages;

    }

    public String getIsbn() {

        return isbn;

    }

}

## Define the Subclass

Now, let's define the **ExampleBooks** class that extends **Book.**

**ExampleBooks.java**

public class ExampleBooks extends Book

{

    private String area;

    // Constructor for ExampleBooks

    public ExampleBooks(String title, String author, int pages, String isbn, String area)

{

        super(title, author, pages, isbn); // Call the constructor of the superclass

        this.area = area;

    }

    // Getter for area

    public String getArea()

{

        return area;

    }

}

**Inheritance**:

* The **ExampleBooks** class uses the e**xtends** keyword to inherit from the Book class.
* This means **ExampleBooks** has access to all public and protected fields and methods of the **Book** class.

**Constructor Inheritance**:

* The constructor of **ExampleBooks** takes additional parameters specific to **ExampleBooks**, like **area**.
* Inside the constructor, the **super** keyword is used to call the constructor of the Book class. This ensures that the Book part of a ExampleBooks object is properly initialized.

## Using the Subclass

To create an instance of ExampleBooks and use its methods and fields:

**Main.java**

public class Main {

    public static void main(String[] args) {

        // Create an instance of ExampleBooks

        ExampleBooks eb = new ExampleBooks(

            "Neural Networks",

            "Simon Haykin",

            696,

            "0-02-352761-7",

            "Artificial Intelligence"

        );

        // Accessing methods from the superclass

        System.out.println("Title: " + eb.getTitle());

        System.out.println("Author: " + eb.getAuthor());

        System.out.println("Pages: " + eb.getPages());

        System.out.println("ISBN: " + eb.getIsbn());

        // Accessing methods from the subclass

        System.out.println("Area: " + eb.getArea());

    }

}

**Using the Subclass**:

* In the **Main** class, we create an instance of **ExampleBooks.**
* We can call methods from both the **Book** class and **ExampleBooks** class on this instance.

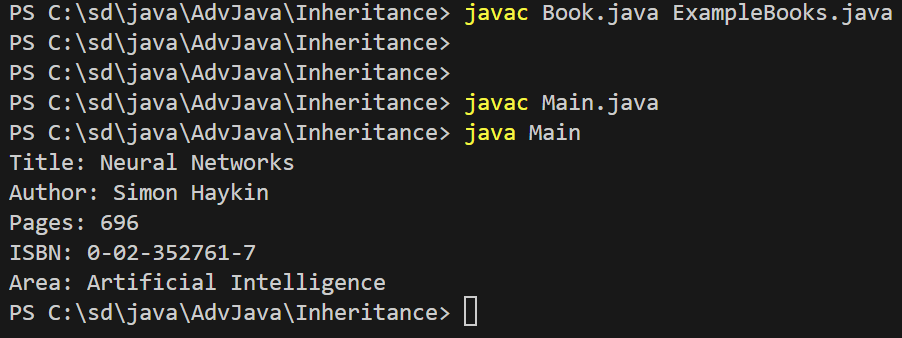
## Key Points

* **Inheritance** allows ScientificBook to use and extend the functionality of Book.
* **Constructors** in the subclass must call the superclass constructors to ensure the superclass part of the object is initialized.
* **Super Keyword**: super is used to call the superclass constructor and must be the first statement in the subclass constructor.
* **Encapsulation**: Both classes use private fields with public getters (and setters if needed) to maintain encapsulation.

## Compile and Run

**javac Book.java ExampleBooks.java**

**javac Main.java**



# Instanceof and getClass()

 **instanceof**:

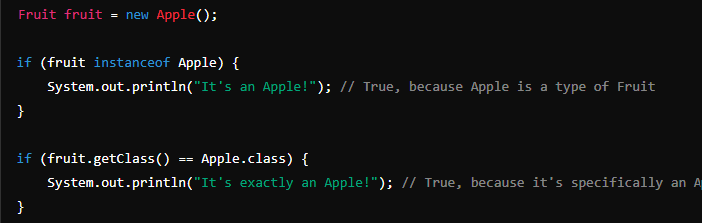
* Checks if an object is a **certain type or a subtype**.
* Useful for safely converting (casting) an object to another type.

 **getClass()**:

* Checks if an object is **exactly a specific type.**
* **Does not consider subtypes**.

### Simplified Example

Imagine you have a general category called **Fruit**, and specific types like **Apple** and **Banana**.



In this example:

* **instanceof** checks if **fruit** is an **Apple or any subclass** of Apple.
* **getClass()** checks if **fruit** is **exactly an Apple**, not any subclass

Let's say you have a class hierarchy with Animal, Dog, and Cat:

## Create Main1.java

class Animal { }

class Dog extends Animal { }

class Cat extends Animal { }

public class Main1 {

    public static void main(String[] args) {

        Animal animal = new Dog();

        if (animal instanceof Dog) {

            System.out.println("It's a Dog");

        }

        if (animal.getClass() == Dog.class) {

            System.out.println("It's exactly a Dog");

        }

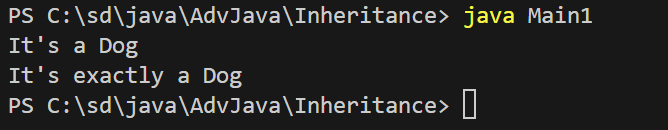
    }

}

## Compile and Run Main1.java

**javac Main1.java**

**java Main1**

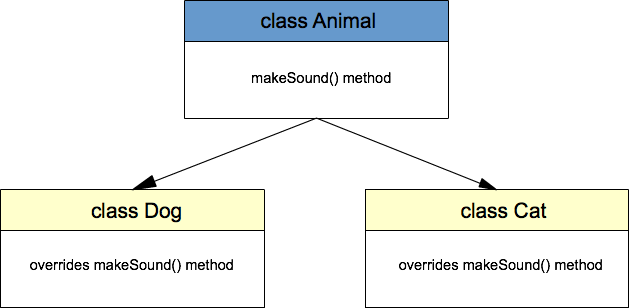
****

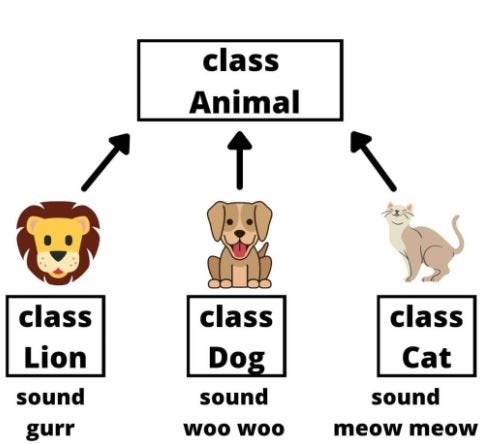
In this example:

* instanceof will print "It's a Dog" because animal is an instance of Dog (considering the hierarchy).
* getClass() will print "It's exactly a Dog" because the runtime class of animal is Dog.

# Polymorphism

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class rather than their actual class. The most common use of polymorphism is method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.





### ****Animal.java****

This file contains the base class Animal:

// Animal.java

public class Animal {

    public void speak() {

        System.out.println("Animal makes a sound");

    }

}

### ****Dog.java****

This file contains the Dog class:

// Dog.java

public class Dog extends Animal {

    @Override

    public void speak() {

        System.out.println("Dog barks");

    }

}

### ****Cat.java****

This file contains the Cat class:

// Cat.java

public class Cat extends Animal {

    @Override

    public void speak() {

        System.out.println("Cat meows");

    }

}

### ****Cow.java****

This file contains the Cow class:

// Cow.java

public class Cow extends Animal {

    @Override

    public void speak() {

        System.out.println("Cow moos");

    }

}

### ****AnimalMain.java****

This file contains the Main class with the main method:

// Main.java

public class AnimalMain {

    public static void animalSound(Animal animal) {

        animal.speak();

    }

    public static void main(String[] args) {

        Animal dog = new Dog();

        Animal cat = new Cat();

        Animal cow = new Cow();

        animalSound(dog); // Outputs: Dog barks

        animalSound(cat); // Outputs: Cat meows

        animalSound(cow); // Outputs: Cow moos

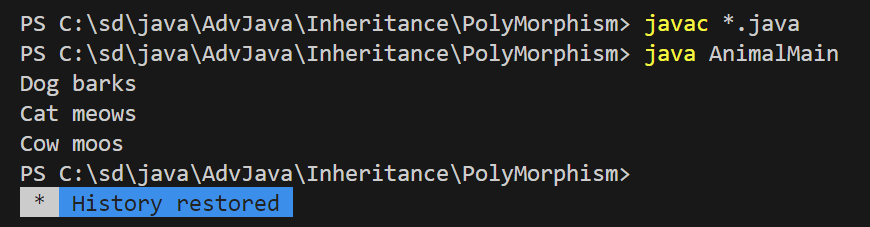
    }

}

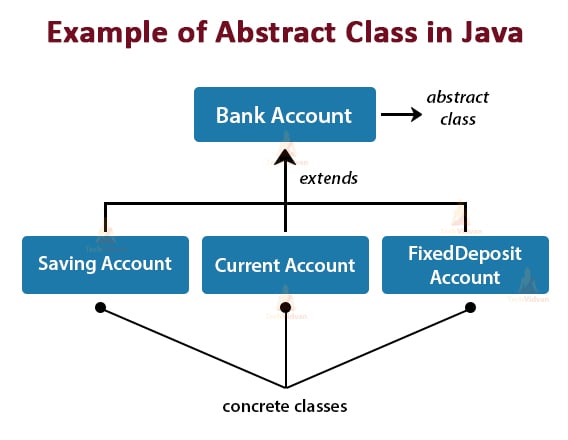
### **Steps to Compile and Run**

**javac \*.java**

**java AnimalMain**

****

# Abstract Class



**Book.java**

**Abstract Class (Book)**:

* The Book class is abstract and cannot be instantiated directly.
* It has two properties: title and author.
* It includes an abstract method display() that must be implemented by any subclass.

 **Derived Classes (Fiction and NonFiction)**:

* Both classes extend Book and provide specific implementations of the display() method.
* They call the constructor of the Book class using super() to set the title and author.

// Book.java

abstract class Book {

    String title;

    String author;

    Book(String title, String author) {

        this.title = title;

        this.author = author;

    }

    abstract void display();

}

class Fiction extends Book {

    Fiction(String title, String author) {

        super(title, author);

    }

    @Override

    void display() {

        System.out.println("Fiction Book: " + title + " by " + author);

    }

}

class NonFiction extends Book {

    NonFiction(String title, String author) {

        super(title, author);

    }

    @Override

    void display() {

        System.out.println("Non-Fiction Book: " + title + " by " + author);

    }

}

**ExampleBooks.java**

**Main Method (ExampleBooks.java)**:

* In the main method, we create instances of Fiction and NonFiction and call their display() methods to print book details.

// Example.java

public class ExampleBooks {

    public static void main(String[] args) {

        Book fictionBook = new Fiction("1984", "George Orwell");

        Book nonFictionBook = new NonFiction("Sapiens", "Yuval Noah Harari");

        fictionBook.display(); // Outputs: Fiction Book: 1984 by George Orwell

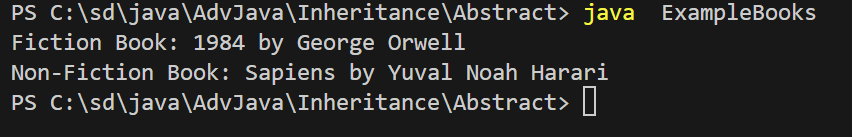
        nonFictionBook.display(); // Outputs: Non-Fiction Book: Sapiens by Yuval Noah Harari

    }

}

**javac \*.java**

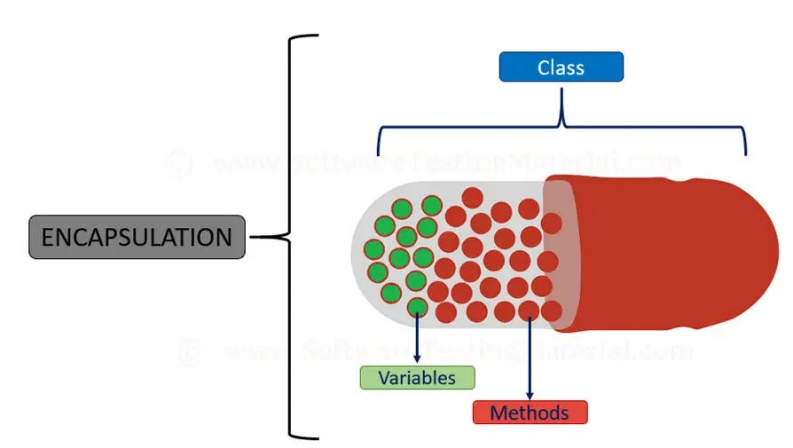
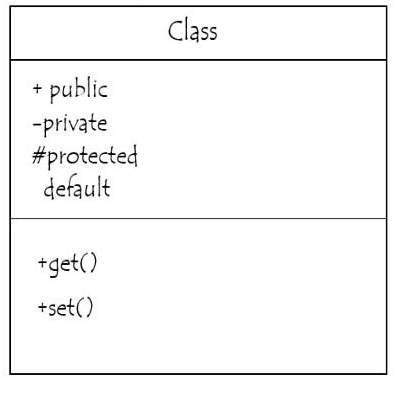
**java ExampleBooks**

****

# Encapsulation

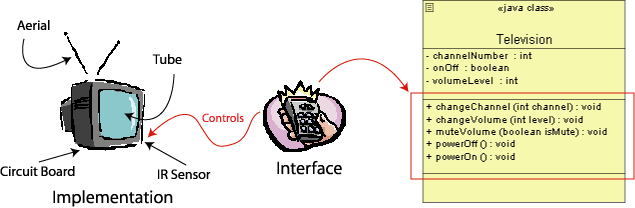
### Concept

Encapsulation is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

****

### TV Remote Example

A great real-life example of encapsulation is a **television remote control**.



### How TV Remote Works:

1. **Inside the Remote (Encapsulation)**:
   * The remote has a lot of complex electronics and circuits inside that control the TV. You don't need to know how these circuits work or how they connect to the TV. The inner workings are hidden from you.
2. **Public Interface**:
   * The remote has buttons like Power, Volume Up/Down, Channel Up/Down, etc. These buttons are the public interface. You press these buttons to interact with the TV.
   * Each button performs a specific function without needing you to understand how the remote communicates with the TV internally.

### Encapsulation in Action with TV Remote:

* **Data and Methods**: The remote contains internal data (like which channel you're on) and methods (like sending signals to change the channel or volume). These are all bundled together inside the remote.
* **Hidden Details**: You don't need to worry about the complex electronics or the signal transmission. You just use the buttons to control the TV.
* **Controlled Access**: The remote provides a controlled way to interact with the TV. You can’t access the TV’s internals directly, but you can use the buttons to achieve your goal.

So, the television remote encapsulates the complexity of its functions and only exposes a simple set of controls to the user, making it a perfect real-life example of encapsulation!

### Book.java and ExampleBooks.java

### Book.java

We’ll make the data fields **title** and **author** private and provide **getter** and **setter** methods to **access** and **modify** them.

// Book.java

abstract class Book {

    private String title;

    private String author;

    // Constructor

    Book(String title, String author) {

        this.title = title;

        this.author = author;

    }

    // Getter and Setter for title

    public String getTitle() {

        return title;

    }

    public void setTitle(String title) {

        this.title = title;

    }

    // Getter and Setter for author

    public String getAuthor() {

        return author;

    }

    public void setAuthor(String author) {

        this.author = author;

    }

    // Abstract method

    abstract void display();

}

class Fiction extends Book {

    Fiction(String title, String author) {

        super(title, author);

    }

    @Override

    void display() {

        System.out.println("Fiction Book: " + getTitle() + " by " + getAuthor());

    }

}

class NonFiction extends Book {

    NonFiction(String title, String author) {

        super(title, author);

    }

    @Override

    void display() {

        System.out.println("Non-Fiction Book: " + getTitle() + " by " + getAuthor());

    }

}

### .1 Explanation of Encapsulation

1. **Private Data Fields**:
   * title and author in the Book class are marked as private. This restricts direct access to these fields from outside the class.
2. **Public Getters and Setters**:
   * Methods like getTitle(), setTitle(), getAuthor(), and setAuthor() provide controlled access to the private fields. They allow other classes to read or modify the values of title and author without directly accessing the fields.
3. **Encapsulation Benefits**:
   * **Controlled Access**: Only the methods provided can access or modify the fields, which helps in maintaining the integrity of the data.

### ExampleBooks.java

This file will remain the same, as it demonstrates how to use the encapsulated Book objects.

// ExampleBooks.java

public class ExampleBooks {

    public static void main(String[] args) {

        Book fictionBook = new Fiction("1984", "George Orwell");

        Book nonFictionBook = new NonFiction("Sapiens", "Yuval Noah Harari");

        fictionBook.display(); // Outputs: Fiction Book: 1984 by George Orwell

        nonFictionBook.display(); // Outputs: Non-Fiction Book: Sapiens by Yuval Noah Harari

        // Access and modify the title and author through getters and setters

        fictionBook.setTitle("Animal Farm");

        fictionBook.setAuthor("George Orwell");

        fictionBook.display(); // Outputs: Fiction Book: Animal Farm by George Orwell

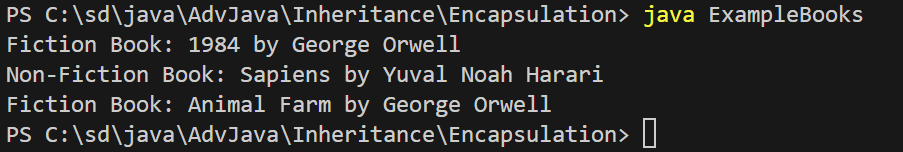
    }

}

### Compile and Run

**javac \*.java**

**java ExampleBooks**

****