JAVA ENTERPRISE APPLICATION DEVELOPMENT

Ms. Archana A

Department of Computer Applications

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Inheritance and Multithreading

Ms. Archana A
Department of Computer Applications

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).

```
class Subclass-name extends Superclass-name
{
//methods and felds
}
```

Java Inheritance

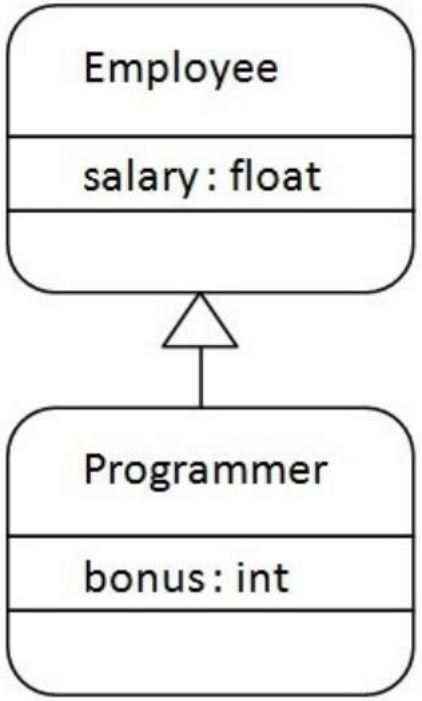
- **The extends keyword is used to perform inheritance in Java.**

- The extends keyword indicates that you are making a new class that derives from an existing class.

- The meaning of "**extends**" is **to increase the functionality.**

Java Inheritance

```
class Employee{
        float salary=40000;
}
class Programmer extends Employee{
        int bonus=10000;
        public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
        }
}
```



Java Inheritance

Another Example:

```
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);
  }
}
```

cont…

Java Inheritance

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

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

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

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

  }
}
```

Java Inheritance

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.

Java Inheritance

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.
- For **Method overriding,** which is also known as **runtime polymorphism**.
- Hence, we can achieve Polymorphism in Java with the help of inheritance.

Types of Inheritance

Inheritance can be further divided into the following types:

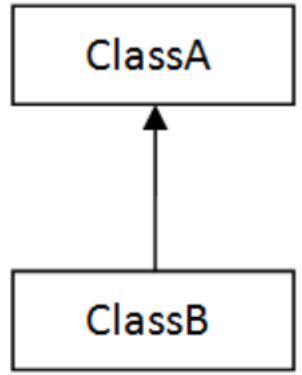
- **Single level**
- **Multi-level**
- **Hierarchical**
- **Multiple**
- **Hybrid**

Multiple and hybrid inheritance **is not directly supported** in java, instead it is achieved through the use of **interfaces** in java.

Java Inheritance

1. Single Inheritance

- When a single class inherits the attributes and methods of another class, it is known as single inheritance.



1) Single

```
class FundamentalForce {
    void Force() {
        System.out.println("There are four fundamental forces.");
    }
}

class Gravitational extends FundamentalForce {
    void Gravity() {
        System.out.println("Fruits fall to the ground due to gravitational Force.");
    }
}
```

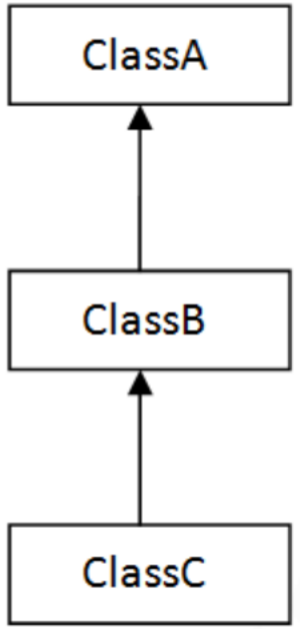
Java Inheritance

```
class SingleInheritance {
    public static void main(String[] args) {
        Gravitational G = new Gravitational();
        G.Force();
        G.Gravity();
    }
}
```

2. Multi-level Inheritance

- When a **classC** inherits attributes and methods from **classB** which in turn inherits its attributes and methods from **classA**, it is called a multi-level inheritance.

- It forms a child-parent-grandparent (or a parent-child-grandchild) relationship. Meaning that child inherits from the parent while the parent inherits from the grandparent.



2) Multilevel

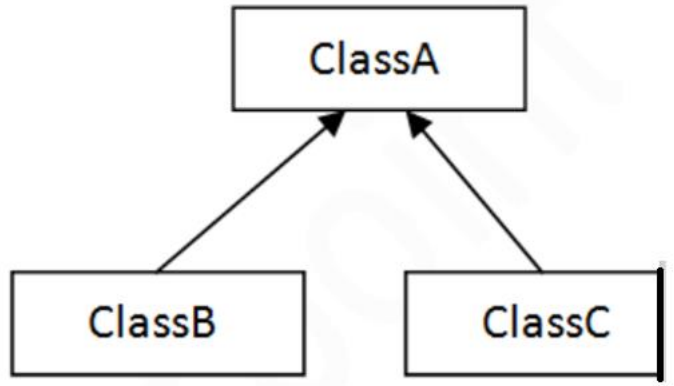
Java Inheritance

```
class NuclearForce extends FundamentalForce {
    void Nuclear() {
        System.out.println("Nuclear Forces are of two types;");
        System.out.println("Strong Nuclear Force");
        System.out.println("Weak Nuclear Force");
    }
}

class StrongNuclearForce extends NuclearForce {
    void Strong() {
        System.out.println("Strong Nuclear Force is responsible for the underlying stability of
matter.");
    }
}
```

Java Inheritance

```
class MultilevelInheritance {
    public static void main(String[] args) {
        StrongNuclearForce st = new StrongNuclearForce();
        st.Force();
        st.Nuclear();
        st.Strong();
    }
}
```



3) Hierarchical

3. Hierarchical Inheritance

- Hierarchical inheritance is when two or more classes inherit from a single class. This can be easily visualized as a parent with more than one child. Here each child can inherit the properties of a parent.

```
class FundamentalForce {
    void Force() {
        System.out.println("There are four fundamental forces.");
    }
}
class Gravitational extends FundamentalForce {
    void Gravity() {
        System.out.println("Fruits fall to the ground due to gravitational Force.");
    }
}
class Electromagnetic extends FundamentalForce {
    void Particles() {
        System.out.println("The electromagnetic force acts between charged particles");
    }
}
```

Java Inheritance

```
class HierarchicalInheritance {
    public static void main(String[] args) {
        System.out.println("Child 1:");
        Gravitational G = new Gravitational();
        G.Force();
        G.Gravity();

        System.out.println();
        System.out.println("Child 2");
        Electromagnetic em = new Electromagnetic();
        em.Force();
        em.Particles();
    }
}
```

Try This:

1. Create a class Vehicle with attributes make and model. Add a method displayInfo() that prints out the make and model of the vehicle. Create a subclass Car that inherits from Vehicle and adds an additional attribute year. Create another subclass SportsCar that inherits from Car and adds an additional attribute topSpeed. Override the displayInfo() method in SportsCar to also print out the year and top speed.

Try These:

2. Create a class Shape with attributes color and area. Add a method calculateArea() that sets the area to a default value (e.g., 0.0). Create two subclasses Circle and Rectangle that inherit from Shape. Implement the calculateArea() method in each subclass to calculate the area based on the respective shapes. Add a method displayInfo() to print out the color and area of the shape.

Try These:

3. Create a class Person with attributes name, age, and email. Add a method displayInfo() that prints out the name, age, and email of the person. Create a subclass Student that inherits from Person and adds an additional attribute studentId. Override the displayInfo() method in Student to also print out the student ID.

Multiple Inheritance

Why multiple inheritance is not supported in java through class?

- In multiple inheritance, one class extends two superclasses or base classes. But in Java, one class cannot simultaneously extend more than one class. **One class at a time can extend to only one class**. Hence, to reduce ambiguity Java does not support multiple inheritance through classes.

- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there **will be ambiguity to call the method** of A or B class..

Java Inheritance

```
class A{
        void msg(){System.out.println("Hello");}
        }
class B{
        void msg(){System.out.println("Welcome");}
        }

class C extends A,B{              //suppose if it were
        public static void main(String args[]){
        C obj=new C();
        obj.msg();          //Now which msg() method would be invoked?
}
}
```

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Method Overriding

Ms. Archana A
Department of Computer Applications

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding** in Java.

- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as **method overriding.**

- Method overriding is used for **runtime polymorphism**

Method Overriding

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Method Overriding

Points to remember:

Can we override static method?

- No, a static method cannot be overridden. It can be proved by runtime polymorphism.

Why we cannot override static method?

- It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

- No, because the main is a static method.

Try These:

Create a class **Employee** with attributes **name, employeeId, and salary**. Add a method **displayInfo()** that prints out the name, employee ID, and salary of the employee. Create a subclass **Manager** that inherits from Employee and adds an additional attribute **department**. Create another subclass **Director** that inherits from Manager and adds an additional attribute **bonus**.

Override the displayInfo() method in Director and also print out the department and bonus.

Java Inheritance

Try These:

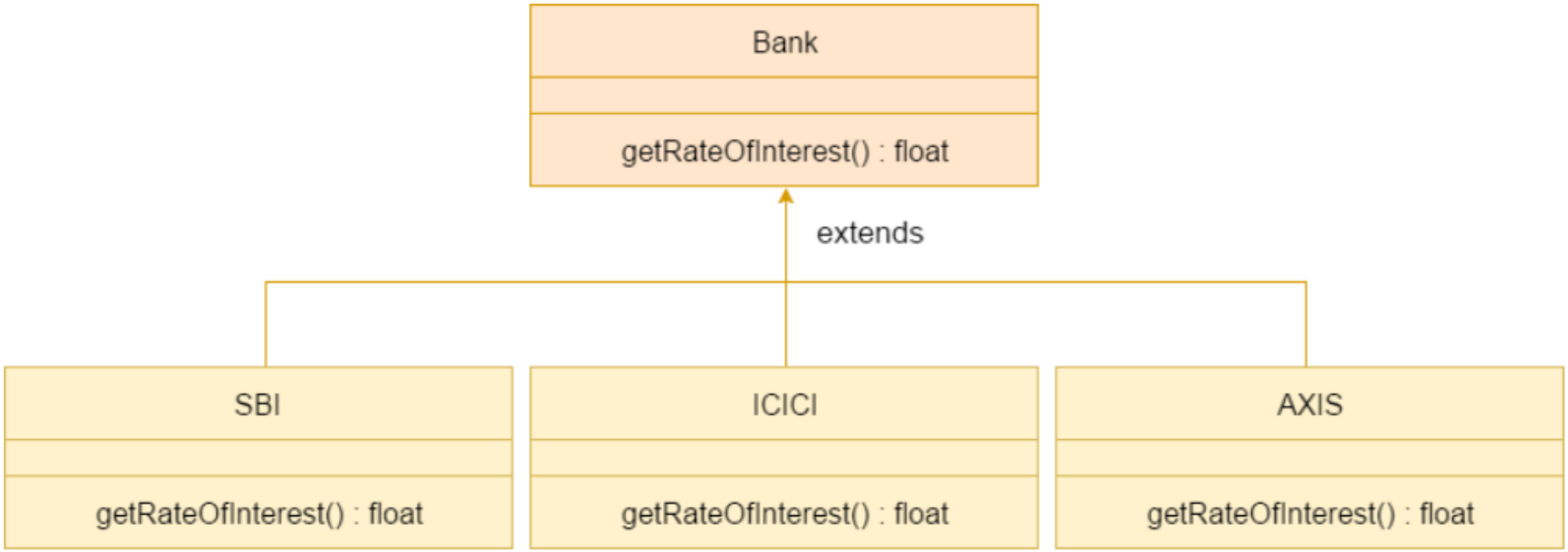
Create a class **Person** with attributes **name and age**. Add a method **displayInfo()** that prints out the name and age of the person. Create a subclass **Student** that inherits from Person and adds an additional attribute **studentId**. Create another subclass **Employee** that inherits from Person and adds an additional attribute **employeeId.** Finally, create a subclass **TeachingAssistant** that inherits from Employee. Add an additional attribute **course** for the class the teaching assistant is helping with.

Override the displayInfo() method in TeachingAssistant and also print out the employee ID, and course.

Method Overriding

Try this:

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

super keyword

Ms. Archana A
Department of Computer Applications

Access Parent class member

Super Keyword in Java

- The **super** keyword in Java is a **reference variable** which is used to refer **immediate parent class object.**

- Whenever you create the instance of subclass, an instance of parent class is created **implicitly** which is referred by **super reference variable**

PES UNIVERSITY

Usage of Java super Keyword

1. **super** can be used to refer **immediate parent class instance variable**.
2. **super** can be used to invoke **immediate parent class method**.
3. **super()** can be used to invoke **immediate parent class constructor**

1) super is used to refer immediate parent class instance variable.

- We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

For example:

- Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

Java Inheritance

```
class Animal{
        String color="white";
}
class Dog extends Animal{
        String color="black";
        void printColor(){
                System.out.println(color);      //prints color of Dog class
                System.out.println(super.color);        //prints color of Animal class
        }
}
```

```
class TestSuper1{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

}

}
```

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class.

In other words, it is used if method is overridden

Java Inheritance

```
class Animal{
        void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
        void eat(){System.out.println("eating bread...");}
        void bark(){System.out.println("barking...");}
        void work(){
                super.eat();
                bark();
        }
}
```

```
class TestSuper2{
        public static void main(String args[]){
                Dog d=new Dog();
                d.work();
        }
}
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class **by default** because **priority is given to local**. To call the parent class method, we need to use **super keyword**

3) super is used to invoke parent class constructor.

- The super keyword can also be used to invoke the parent class constructor.

Let's see a **simple example:**

```
class Animal{
        Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
        Dog(){
        super();
        System.out.println("dog is created");
        }
}
```

```
class TestSuper3{
        public static void main(String args[]){
        Dog d=new Dog();
        }
}
```

Output:

animal is created

dog is created

THANK YOU

Ms. Archana A
Department of Computer Applications

archana@pes.edu

+91 80 6666 3333 Extn 392

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Final variables and methods, Final class

Ms. Archana A
Department of Computer Applications

Final variable and methods

- All methods and variables can be overridden by default in subclasses.
- If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword **final** as modifier

Example,

```
final int size=100;

final void show(){ ……}
```

Final variable and methods

- Making a method final ensures that the **functionality defined** in this method **will never be altered** in any way.
- Similarly, the **value of a final variable can never be changed**.
- Final variables, behave like class variables and they do not take any space on individual objects of the class.

Final variable example

Consider for eg1:

```
public final int MY_CONSTANT = 10;
```

- The MY_CONSTANT variable is **final**, which means that its value cannot be changed after it is initialized.

Example2:

```
public class MyClass {
private final String myName;
public MyClass(String name) {
        myName = name;
        }
        public String getName() {
                return myName;
        }
}
```

Final variable example

Ex3:

```
public class FinalVariableExample {
    final int MAX_VALUE = 100; // This is a final variable

    public static void main(String[] args) {
        FinalVariableExample example = new FinalVariableExample();
        System.out.println("MAX_VALUE: " + example.MAX_VALUE);

        // Uncommenting the line below will result in a compilation error
        // example.MAX_VALUE = 200;
    }
}
```

Final method example

Similarly, consider an example of final method

```
public class MyClass {
        public final void myMethod() {
        System.out.println("This is my final method.");
        }
}
```

- Here myMethod() method in the MyClass class is final, which means that it **cannot be overridden** by subclasses

Final method example2

```
class Parent {
    final void display() {
        System.out.println("This is the parent class");
    }
}

class Child extends Parent {
    // Uncommenting the following method will result in a compilation error
    // void display() { }
}

public class FinalMethodExample {
    public static void main(String[] args) {
        Child child = new Child();
        child.display(); // Calls the display method from the parent class
    }
}
```

Final Classes

Sometimes we may like to prevent class being further subclassed for security reasons. A class that cannot be subclassed is called final class. This is achieved in java using keyword **final** as follow:

final class Vehicle{…….}

final class Bike extends someotherclass{……..}

Final Classes

```
public final class MyClass {
    public void myMethod() {
        System.out.println("This is my method.");
    }
}
```

- Here MyClass class is final, which means that it cannot be extended by subclasses

Final Classes

In Java, a final class is a class that **cannot be subclassed**. Here's a another simple example:

```
final class FinalClass {
    // This is a final class
}

// Uncommenting the following line will result in a compilation error
// class SubClass extends FinalClass { }
```

Final Classes

In this example, we have a class FinalClass declared with the final keyword.

This means that FinalClass cannot be extended by any other class.

If you uncomment the line class SubClass extends FinalClass { }, it will result in a compilation error because FinalClass is a final class and cannot be subclassed.

Attempting to extend a final class will result in a compilation error, preventing any further subclassing.

Final method example3

```
class Parent {
    final void display() {
        System.out.println("This is the parent class");
    }
}

class Child extends Parent {
    // Uncommenting the following method will result in a compilation error
    void display() { }
}

public class FinalMethodExample {
    public static void main(String[] args) {
        Child child = new Child();
        child.display(); // Calls the display method from the parent class
    }
}
```

Final method example4

```
public class MyClass {
    public void myMethod() {
        System.out.println("This is my method.");
    }
}

public class MySubClass extends MyClass {
    // This will not compile because the MyClass class is final
    public class MyClass extends MyClass {
        public void myMethod() {
            System.out.println("This is my overridden method.");
        }
    }
}
```

Final -Summary

- **Final variables, final methods, and final classes** can be used to make your code more secure and reliable.
- For example, final variables can be used to ensure that important constants are **not accidentally changed**.
- Final methods can be used to **prevent subclasses from overriding** important functionality.
- final classes can be used to **prevent subclasses from extending** and **modifying** the class

Finalizer methods

- We have seen that a constructor method is used to initialize an object when it is declared. This process is known as initialization.

- Similarly, java supports a concept called finalization, which is just opposite to initialization.
- We know that java runtime is an automatic garbage collecting system. It automatically frees up the memory resources we used by the objects. But **objects may hold other non-object resources such as file descriptors or window system fonts**. The **garbage collector cannot free these resources**. In order to free these resources we must use a finalizer method. This is similar to destructors in C++.
- The finalizer method is simply finalize().

Syntax:

Protected void finalize() {……}

Final Classes

```
// FinalVariableDemo.java

// Final class cannot be extended
final class FinalClass {
    final int MAX_VALUE = 100; // Final variable
    final String NAME; // Final variable without initialization

    // Constructor to initialize the final variable
    public FinalClass(String name) {
        NAME = name;
    }

    final void display() { // Final method
        System.out.println("FinalClass Display");
    }
}
```

Final Classes

```
// This line would cause an error since FinalClass cannot be extended
// class SubClass extends FinalClass { }

class SubClass {
    // Final variable can be assigned only once
    final int MAX_VALUE = 200; // This is allowed because it's a new variable

    // Final method cannot be overridden
    // Uncommenting the following line would cause an error
    // final void display() { }
}
```

Final Classes

```
public class FinalVariableDemo {
    public static void main(String[] args) {
        FinalClass obj = new FinalClass("OpenAI");
        System.out.println("Name: " + obj.NAME);
        System.out.println("MAX_VALUE: " + obj.MAX_VALUE);
        obj.display();

        SubClass subObj = new SubClass();
        System.out.println("MAX_VALUE in SubClass: " + subObj.MAX_VALUE);
    }
}
```

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Abstract Class

Ms. Archana A
Department of Computer Applications

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();
```

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");
  }
}
```

Abstract Class

Rules for Abstact class

- Use abstract keyword
- You cannot instantiate an abstract class
- An abstract class contains both abstract and non-abstract methods
- You can also include constructors and non-abstract static methods in your abstract class

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."

Abstract Class

For example,

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

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

Abstract Class

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");
  }
}
```

Abstract Class

```
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();
  }
}
```

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

Try These:

1. Create an abstract class **Shape** with an abstract method **calculateArea().** Then create two subclasses **Circle and Rectangle** that extend the Shape class. Implement the calculateArea() method in each subclass to calculate the area of a circle and a rectangle respectively.

2. Create an abstract class **BankAccount** with fields **accountNumber, balance, and ownerName**. Define an abstract method **withdraw()** and a concrete method **deposit()** that adds a specified amount to the balance. Then create two subclasses **SavingsAccount** and **CheckingAccount** that extend the BankAccount class. Implement the withdraw() method in each subclass to handle withdrawals specific to savings and checking accounts.

Try This:

Exercise 3:
Create an abstract class **Employee** with fields **name, id, and salary**. Define an abstract method **calculateBonus()** that calculates the bonus based on the salary. Then create two subclasses **Manager and Developer** that extend the Employee class. Implement the calculateBonus() method in each subclass.

Exercise 4:
Create an abstract class **Vehicle** with abstract methods **startEngine()** and **stopEngine()**. Then create two subclasses **Car and Motorcycle** that extend the Vehicle class. Implement the startEngine() and stopEngine() methods to simulate starting and stopping the engines of a car and a motorcycle.

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Interface

Ms. Archana A
Department of Computer Applications

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.

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);
  }
}
```

Interface

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

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 {
   ...
}
```

Try these:

1. Create an interface called Shape with a method calculateArea(). Implement this interface in classes Circle and Rectangle. The classes should have appropriate properties and methods.

2. Create two interfaces, Person and Employee. Implement these interfaces in a class called EmployeeDetails. The Person interface should have methods for setting and getting the person's name, and the Employee interface should have methods for setting and getting the employee ID.

3. Create an interface called Engine with a method start(). Then create another interface called Vehicle that extends Engine and adds a method stop(). Implement these interfaces in a class called Car.

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 us consider 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. The 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.

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);
  }
}
```

Interface

```
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();
  }
}
```

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Adapter class

Ms. Archana A
Department of Computer Applications

Interface… Adapter Class

- In Java, an **adapter class** is a class that provides an **empty or default implementation of certain methods of an interface.**

- It allows you to create a class that implements an interface by extending the adapter class and only overriding the methods that are necessary.

- This is particularly useful when you want to implement an interface that has a large number of methods, but you are only interested in providing a specific implementation for a subset of those methods.

THANK YOU

Ms. Archana A
Department of Computer Applications

archana@pes.edu

+91 80 6666 3333 Extn 392

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Java Packages

Ms. Archana A
Department of Computer Applications

Package

Java Package

- A package is simply a container that groups related types (Java classes, interfaces, enumerations, and annotations).

- All classes in java belong to some package. When no package statement is specified, the default package is used. The default package has no name, which makes the default package transparent.

- Package is a mechanism for organizing a group of related files in the same directory. It has the ability to reuse the classes and interfaces declared/defined in the package, again and again.

- Package in java can be categorized in two form, **built-in package** and **user-defined package.**

Package – builtin packages

- There are many built-in packages such as:

Package Name	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.awt	Contains classes that support the Abstract Window Toolkit
Java.util	Contains some abstract datatypes and interfaces
Java.swing	Contains second generation graphics elements
Java.applet	Contains classes for implementing applets

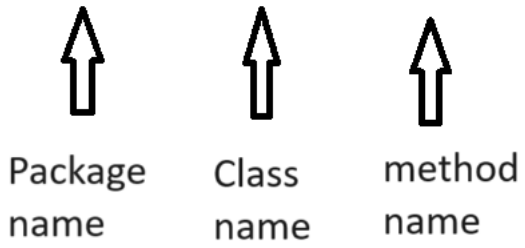
Package – user-defined packages

Package Naming Conventions

- Packages begin with lowercase letters. This makeseasy for users to distinguish package names from class names when looking at an explicit reference to a class.

For eg:

```
double y = java.lang.Math.sqrt(x);
```



How to import packages in Java?

Java has an **import** statement that allows you to import an entire package (as in earlier examples), or use only certain classes and interfaces defined in the package.

The general form of import statement is:

import package_name.ClassName; // To import a certain class only
import package_name.* // To import the whole package

Creating Packages

- To create user defined package, we first declare the name of the package using the package keyword followed by a package name. this must be the first statement in a java source file then we define a class . For example:

```
package firstPackage;  //package declaration
public class Firstclass{   //class definition
--------------------
}
```

Package -Example for Accessing a Package

```
package p1;
public class A{
        public void displayA(){
            System.out.println("Class A");
        }
}
import p1.A;
class PackageDemo{
        public static void main(String args[]){
            A  obj = new A();
             obj.displayA();
        }
}
```

Hiding classes

When we import a package using asterisk (*), all public classes are imported. We may prefer to "not import" certain classes, such class should not be declared as public. For Eg:

```
package p1;
public class A {                    //public class available outside
          ------------          }
Class B{                            //non-public class, hidden
   ---------------------  }
}

Import p1.*;
A  a= new A();  // class A is available here
B   b= new B();   //error, B is not available outside
```

Package

Try These:

- WAP to implement single package in java
- WAP to implement multiple packages in java

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Java Exception Handling

Ms. Archana A
Department of Computer Applications

Exception Handling

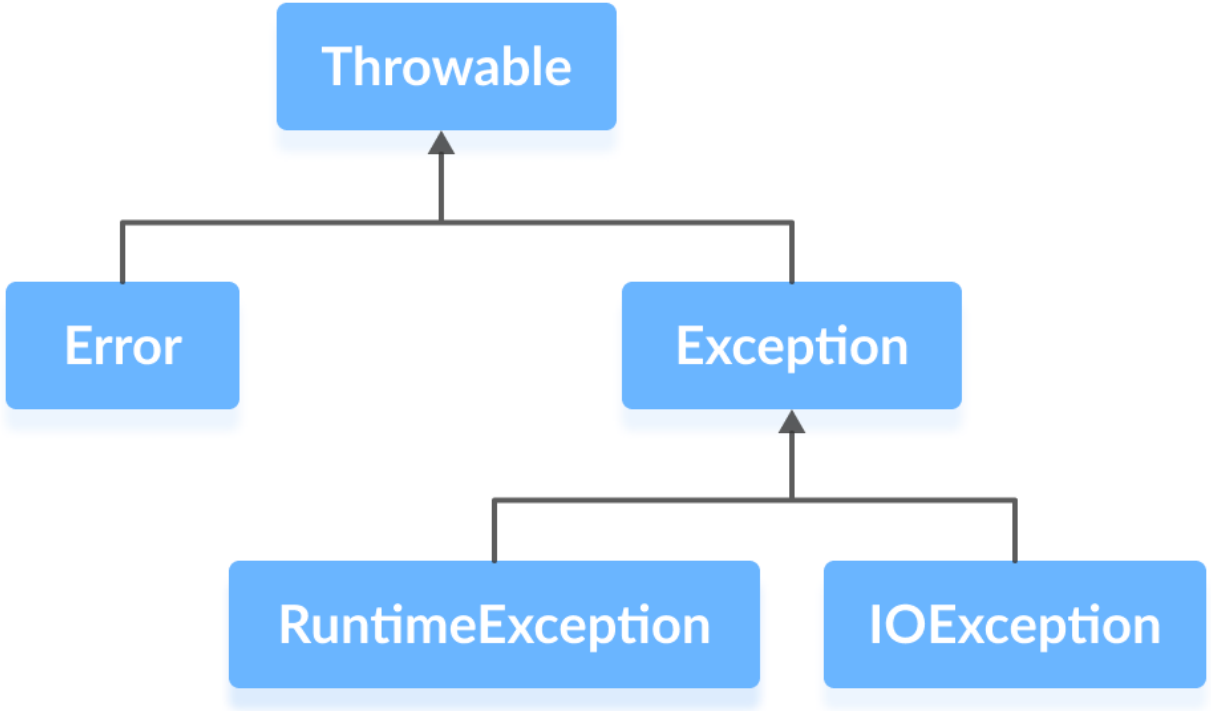
An exception is an **unexpected event** **that occurs during program execution**.
It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Exception Handling

Java Exception hierarchy
Here is a simplified diagram of the exception hierarchy in Java.



- As you can see from the image above, the **Throwable class** is the **root class** in the hierarchy.

Errors

- Errors represent irrecoverable conditions such as Java virtual machine (JVM) **running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.**
- Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions can be caught and handled by the program.

- When an exception occurs within a method, it creates an object. This object is called the exception object.

- It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

Java Exception Types

The exception hierarchy also has two branches: **RuntimeException** and **IOException**.

1. RuntimeException

- A runtime exception happens due to a programming error. They are also known as **unchecked exceptions.**

- These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - IllegalArgumentException
- Null pointer access (missing the initialization of a variable) - NullPointerException
- Out-of-bounds array access - ArrayIndexOutOfBoundsException
- Dividing a number by 0 - ArithmeticException

2. IOException

- An IOException is also known as a **checked exception**. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

- Trying to open a file that doesn't exist results in **FileNotFoundException**
- Trying to read past the end of a file

Exception Handling

We know that **exceptions abnormally terminate** the execution of a program.

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- **try...catch block**
- **finally block**
- **throw and throws keyword**

1. Java try...catch block

- The try-catch block is used to handle exceptions in Java. Here's the syntax of try...catch block:

```
try {
  // code
}
catch(Exception e) {
  // code
}
```

- Here, we have placed the code that might generate an **exception** inside the **try block**. Every try block is followed by a catch block.
- When an exception occurs, it is caught by the catch block. **The catch block cannot be used without the try block.**

Exception Handling

Example:

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

    try {
      // code that generate exception
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
}
```

Try these:

1. Write a program that reads a text file line by line and prints each line to the console. If the file does not exist or cannot be opened, handle the exception appropriately and display a meaningful error message.

2. Write a program that takes two integers as input and calculates their quotient. If the divisor is zero, handle the exception appropriately and display an error message indicating that division by zero is not allowed.

3. Write a program that takes an integer array as input and prints the element at a specified index. If the index is out of bounds, handle the exception appropriately and display an error message indicating an invalid index.

2. Java finally block

In Java, the **finally block is always executed** no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

The basic syntax of finally block is:

```
try {
  //code
}
catch (ExceptionType1 e1) {
  // catch block
}
finally {
  // finally block always executes
}
```

- If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.

Exception Handling – using finally block

```
class Main {
  public static void main(String[] args) {
    try {
      // code that generates exception
      int divideByZero = 5 / 0;
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }

    finally {
      System.out.println("This is the finally block");
    }
  }
}
```

3. Java throw and throws keyword

- The Java throw keyword is used to **explicitly throw a single exception**.
- When we throw an exception, the flow of the program moves from the try block to the catch block.

```
class Main {
  public static void divideByZero() {

    // throw an exception
    throw new ArithmeticException("Trying to divide by 0");
  }

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

Exception Handling

In the previous example, we are explicitly throwing the ArithmeticException using the **throw** keyword.

Similarly, the **throws** keyword is used to declare the **type of exceptions** that might occur within the method. **It is used in the method declaration.**

```
import java.io.*;

class Main {
  // declareing the type of exception
  public static void findFile() throws IOException {

    // code that may generate IOException
    File newFile = new File("test.txt");
    FileInputStream stream = new FileInputStream(newFile);
  }
```

Exception Handling

```
public static void main(String[] args) {
  try {
    findFile();
  }
  catch (IOException e) {
    System.out.println(e);
  }
 }
}
```

- When we run this program, if the file test.txt does not exist, FileInputStream throws a FileNotFoundException which extends the IOException class.

- The findFile() method specifies that an IOException can be thrown. The main() method calls this method and handles the exception if it is thrown.

- If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause.

Throwing our own Exceptions

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:

throw new Throwable_subclass;

For example:
throw new ArithmeticException();
throw new NumberFormatException();

```
import java.lang.Exceptions;

class MyException extends Exception{
   MyException(String msg){
            super(msg);
            }
}
class TestMyException{
     public static void main(String args[]){
            int x=5,y=1000;
            try{
               float z= (float) x / (float) y;
                if(z < 0.01){
                        throw new MyException("Number is too small");
                        }
```

```
}
catch(MyException e){
        System.out.println("Caught my exception");
        System.out.println(e.getMessage());
        }
finally{
        System.out.println("I am always here");
        }
    }
}
```

1. WAP to authenticateUser method which uses the throw keyword to throw a custom exception (InvalidCredentialsException) when the provided username and password are not valid. The main method catches this exception and handles the authentication failure.

2. Write an exception handling program for the given scenario: The calculateGrade method uses the throw keyword to throw a custom exception (InvalidGradeException) when the provided marks are not within the valid range. The main method catches this exception and handles the invalid grade scenario.

Experiential Learning Exercises on Exception Handling

Scenario 1: ATM Withdrawal

Design an ATM withdrawal system that allows users to withdraw money from their accounts. Implement exception handling to handle various scenarios, such as insufficient funds, invalid withdrawal amount, and network connectivity issues.

Scenario 2: Online Shopping Cart

Create an online shopping cart system that allows users to add items to their cart, calculate the total price, and proceed to checkout. Implement exception handling to handle scenarios such as out-of-stock items, invalid payment information, and shipping address validation.

Experiential Learning Exercises on Exception Handling…

Scenario 3: Error Logging and Reporting

Implement a robust error logging and reporting mechanism for a web application. Capture exceptions that occur during various application processes, such as database operations, user interactions, and external API calls. Log the exceptions with detailed information, including timestamps, error messages, and stack traces. Generate comprehensive error reports that can be analyzed to identify recurring issues and improve application stability.

PES UNIVERSITY

throws keyword Vs. try...catch...finally

- There might be several methods that can cause exceptions. Writing try...catch for each method will be tedious and code becomes long and less-readable.

- throws is also useful when you have checked exception (an exception that must be handled) that you don't want to catch in your current method..

THANK YOU

Ms. Archana A
Department of Computer Applications

archana@pes.edu

+91 80 6666 3333 Extn 392

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Multithreaded Programming

Ms. Archana A
Department of Computer Applications

Introduction

Java programs that we have discussed so far contain **only single sequential flow of control**. The program begins, runs through a sequence of executions and finally ends. At any given point of time **there is only one statement under execution**.

A **thread** is similar to a program that has a **single flow of control**. It has **beginning, a body, and an end** and executes commands sequentially.

A unique property of java is its support for **multithreading**. That is, Java enables us to use **multiple flows of control** in developing programs. Each flow of control may be thought of as a separate tiny program (or module) known as a **thread** that **runs in parallel** to others.

A program that contains **multiple flows of control** is known as **multithreaded program**.

Creating Threads

Threads are implemented in the form of objects that contain a method called **run().** The run() method is the heart and soul of any thread. It makes up entire body of a thread and is the only method in which the thread's behavior can be implemented.

A typical run() would appear as follows:

```
public void run(){
        ----------------
        ------------------
}
```

The run() should be invoked by **an object** of the concerned thread. This can be achieved by **creating the thread and initiating it** with the help of another thread method called **start()**

A **new thread** can be created in **two ways**:

1. **By creating a thread class:** define a class that extends **Thread class** and override its **run()** method with the code required by the thread.

2. **By converting a class to a thread:** define a class that implements **Runnable interface**.

- The **Runnable interface** has only one method **run(),** that is to be defined in the method with the code to be executed by the thread.

Extending the Thread class

We can make our class runnable as a thread **by extending** the class **java.lang.Thread**. This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the Thread class.
2. Implement the run() method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the start() method to initiate the thread execution.

Declaring the class:

```
class MyThread extends Thread{
        ----------
        ----------
}
```

Implementing the run() method

- The run() method has been inherited by the class MyThread. We have to override this method in order to implement the code to be executed by our thread. The basic implementation look like this:

```
public void run() {
        ----------
        ----------
}
```

When we start the new thread, Java calls the thread's run() method, so it is the run() method where the action takes place.

Starting New Thread

To create and run an instance of thread class, do as shown below:

```
MyThread athread = new MyThread();
athread.start();
```

First line instantiates a new object of class MyThread.
Note that this statement just creates the object and the thread that will run this object is **not yet running**. The thread is in **newborn state**.

The second line calls the **start()** method causing the thread to move into the runnable state. Then, the java runtime will schedule the thread to run by invoking its run() method.
Now the thread is said to be in the running state.

Example: creating threads **using the Thread class**

```
class A extends Thread{
    public void run(){
        for(int i=1;i<=5;i++){
        System.out.println("from Thead A: i= " +i);
        }
        System.out.println("Exit from A");
        }
}
class B extends A{
        public void run(){
        for(j=1;j<=5;j++){
        System.out.println("from thread B : j =" +j)
        }
        System.out.println("exit from B");
        }
}
```

```
class C extends A{
        public void run(){
        for(k=1;k<=5;k++){
        System.out.println("from thread C : k =" +j)
        }
        System.out.println("exit from C");
        }
}
Class ThreadTest{
    public static void main(String args[]){
        new A().start();
        new B().start();
        new C().start();
        }
}
```

Example: creating threads using Runnable interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("MyRunnable is running...");
    }
}

public class ThreadTest2 {
    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

Stopping and blocking a Thread

Whenever we want to stop a thread from running further, we need to call its stop() method

athread.stop();

This statement causes the thread to move to the **dead state.** A thread will also move to the dead state automatically when it reaches the end of its method.

The **stop()** method may be used when the **premature death** of a thread is desired.

Blocking a thread

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

sleep() // blocked for a specified time
suspend() // blocked until further orders
wait() // blocked until certain condition occurs

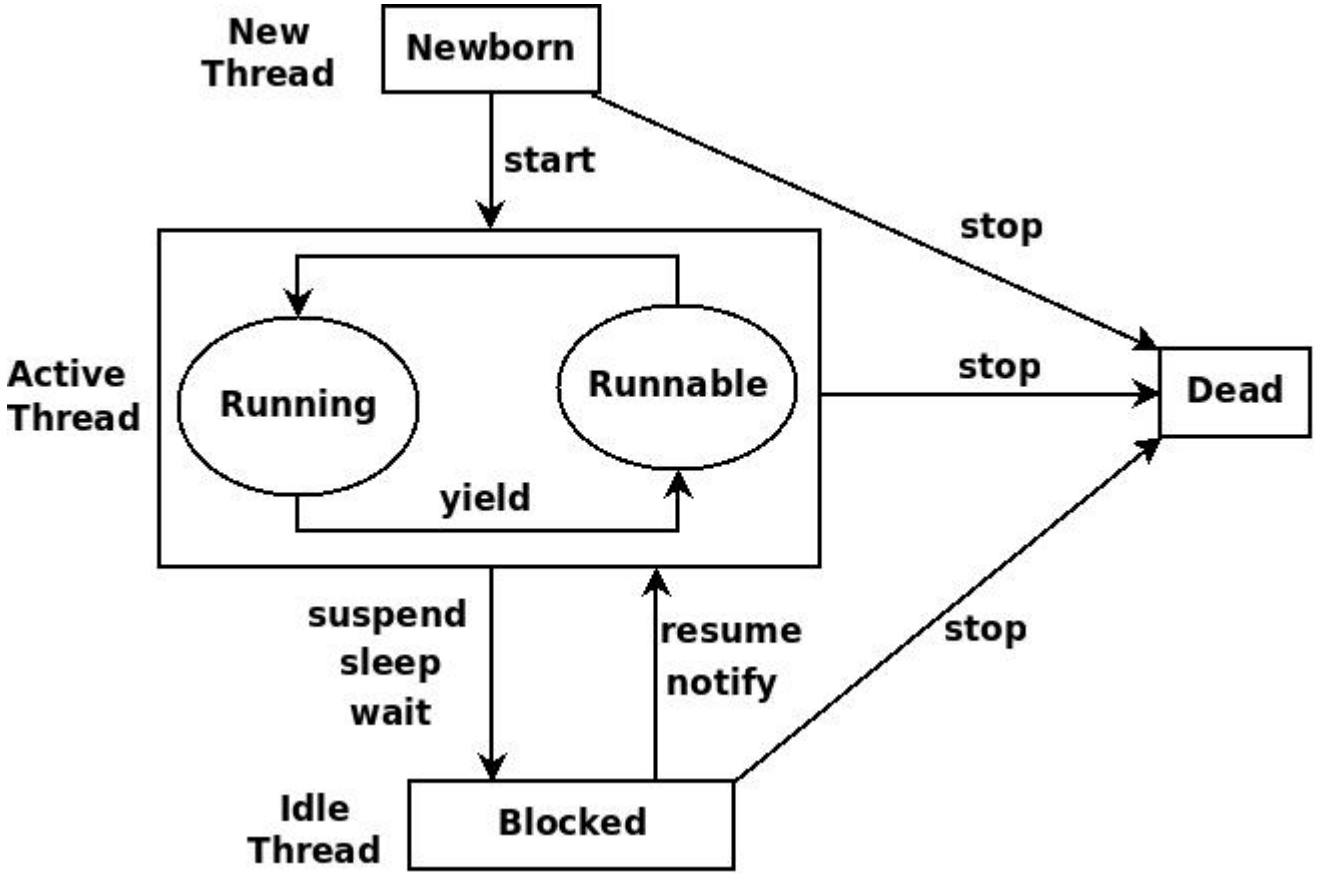
These methods cause the thread to go into the blocked (or not-runnable) state. The thread will return to the runnable state when the **specified time is elapsed** in the case of **sleep()** method, the **resume()** method is invoked in the case of **suspend() ,** and **notify()** method is called in case of **wait()** method.

Life cycle of a thread

During the life time of a thread, there are many states it can enter. They are:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

State transition diagram: **Lifecycle of a thread**



Newborn state:

- When we create a thread object, the thread is born and is said to be in **newborn state**.
  - The thread is **not yet** **scheduled for running**. At this state, we can do only one of the following things with it:

- **Schedule it for running using start() method**
- **Kill it using stop() method.**

If Scheduled, it moves to the **runnable state**. If we attempt to use any other method at this stage, an **exception will be thrown**.

Runnable state:

- The runnable state means that the thread is **ready for execution** and **is waiting for the availability of the processor(CPU)**. That is, the thread has joined the queue of threads that are waiting for execution.

- If all threads have **equal priority,** then they are given **time slots for execution** in **round robin** fashion i.e., **first come first serve manner**.

- The thread that **relinquishes control joins the queue at the end and again waits for its turn**. This process of **assigning time to threads** is known as **time-slicing.**

Running state:

- Running means that the processor has given its time to the thread for its execution.

The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

- A running thread may relinquish its control in one of the following situations:

1. It has been suspended using **suspend()** method. A suspended thread can be reviewed by using **resume()** method. This approach is useful when we want to suspend a thread for some time due to certain reason, but not want to kill it.

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep(time)** where time in **milliseconds**.

This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.

3. It has been told to wait until some event occurs. This is done using the **wait()** method. The thread can be scheduled to run again using the **notify()** method.

PES UNIVERSITY

Blocked state:

- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state.
- This happens when the thread is **suspended, sleeping or waiting** in order to satisfy certain requirements.
- A blocked thread is considered "**not runnable**" **but not dead** and therefore fully qualified to run again

Dead State:

- Every thread has a life cycle. A running thread ends its life when it has completed executing its **run()** method. It is a **natural death.**
- We can kill it by sending the **stop()** message to it at any state thus causing a premature death to it.
- A thread can be **killed** as soon **it is born, or while it is running, or even when it is in "not runnable"** (blocked) condition.

Methods used in Threads

start()	The start method initiates the execution of a thread
currentThread()	The currentThread method returns the reference to the currently executing thread object.
run()	The run method triggers an action for the thread
join()	Stop the current thread until the called thread gets terminated. Used in multithreading.
sleep()	The sleep method is used to suspend the thread temporarily
suspend()	The suspend method is used to instantly suspend the thread execution

Methods used in Threads…

getId()	It gives the id of a thread
getName()	Returns the name of a thread. And always starts from thread-0
setname(string)	Here thread name will be replaced with given string
getPriority()	It returns priority of the thread
setPriority(integer)	We can set the priority of a thread. Highest priority is 10 and minimum priority is 1.
yield()	The yield method is used to send the currently executing threads to standby mode and runs different sets of threads on higher priority
isAlive()	This will gives the status of a thread. Returns true if thread is still running and false if thread completes its execution. The isAlive method is invoked to verify if the thread is alive or dead

Inter-Thread Communication

1. **wait()**
2. **Notify()**
3. **NotifyAll()**

All three Methods belongs to **Object Class**

- **wait(), notify(), and notifyAll()** methods only from synchronized area otherwise we will get runtime exception saying **IllegalMonitorStateException**.
- To call wait (), notify () and notifyAll () methods compulsory the **current thread should be the owner** of that object
- we can call wait(), notify(), and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying IllegalMonitorStateException.

Thread Synchronization

- **wait():** This method is used to **make the particular Thread wait** until it gets a notification. This method pauses the current thread to the waiting room dynamically.

- **notify():** This method is used to **send the notification to one of the waiting thread** so that thread enters into a running state and execute the remaining task. This method **wakeup a single thread into the active state** (that acts on the common object).

- **notifyAll():** This method is used to **send the notification to all the waiting threads** so that **all thread enters into the running state and execute simultaneously**. This method wakes up all the waiting threads that act on the common objects.

Example of Thread Synchronization

```
class BankAccount {
    private int balance = 1000;
    public synchronized void withdraw(int amount)
 {
        if (balance >= amount) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            balance -= amount;
            System.out.println("Withdrawal successful. New balance: " + balance);
        } else {
            System.out.println("Insufficient funds. Balance: " + balance);
        }
    }
}
```

Thread synchronization…

```
public class Example5 {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread thread1 = new Thread(() -> {
            account.withdraw(500);
        });

        Thread thread2 = new Thread(() -> {
            account.withdraw(500);
        });

        thread1.start();
        thread2.start();
    }
}
```

Try these:

Exercise 1: Basic Thread Creation

Create two threads, and each thread should print numbers from 1 to 5. Ensure that the output is interleaved.

Exercise 2: Synchronization

Modify Exercise 1 to use synchronization to ensure that the output is sequential and not interleaved.

Exercise 3: Producer-Consumer Problem

Implement a simple producer-consumer scenario using two threads. One thread (the producer) should produce items (e.g., numbers) and put them into a shared buffer. The other thread (the consumer) should consume the items from the buffer.

Advantages of Multithreading in Java

- Multithreading in Java improves the performance and reliability
- Multithreading in Java minimizes the execution period drastically
- There is a smooth and hassle-free GUI response while using Multithreading in Java
- The software maintenance cost is lower
- The CPU and other processing resources are modes judiciously used

Disadvantages:

- Multithreading in Java poses complexity in code debugging
- Multithreading in Java increases the probability of a deadlock in process execution
- The results might be unpredictable in some worst-case scenarios
- Complications might occur while code is being ported

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Enumeration - enum class

Ms. Archana A
Department of Computer Applications

Java enums

In Java, an enum (short for enumeration) is a type that has a **fixed set of constant values**. We use the **enum** keyword to declare **enums**. For example,

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}
```

- Here, we have created an enum named Size. It contains fixed values SMALL, MEDIUM, LARGE, and EXTRALARGE.

These values inside the braces are called **enum constants** (values).

Note: The enum constants are usually represented in **uppercase.**

Java enum

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}

class Main {
  public static void main(String[] args) {
    System.out.println(Size.SMALL);
    System.out.println(Size.MEDIUM);
  }
}
```

Java enum

Enum Class in Java

In Java, enum types are considered to be a special type of class. It was introduced with the release of Java 5.

An **enum class** can include **methods and fields** just like regular classes.

```
enum Size {
   constant1, constant2, ..., constantN;

   // methods and fields
}
```

Java enum class - for example

```
enum Size{
  SMALL, MEDIUM, LARGE, EXTRALARGE;

  public String getSize() {

    // this will refer to the object SMALL
    switch(this) {
      case SMALL:
        return "small";

      case MEDIUM:
        return "medium";

      case LARGE:
        return "large";

      case EXTRALARGE:
        return "extra large";
```

Java enum

```
      default:
        return null;
      }
    }

  public static void main(String[] args) {

    // call getSize()
    // using the object SMALL
    System.out.println("The size of the pizza is " + Size.SMALL.getSize());
  }
}
```

JAVA ENTERPRISE APPLICATION DEVELOPMENT (JEAD)

Wrapper class

Ms. Archana A
Department of Computer Applications

Wrapper Class

The wrapper classes in Java are used to **convert primitive types** (int, char, float, etc) into **corresponding objects**.

Each of the 8 primitive types has corresponding wrapper classes.

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Wrapper Class…

Convert Primitive Type to Wrapper Objects

We can also use the **valueOf()** method to convert primitive types into corresponding objects

For example:

Wrapper Class…

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

    // create primitive types
    int a = 5;
    double b = 5.65;

    //converts into wrapper objects
    Integer aObj = Integer.valueOf(a);
    Double bObj = Double.valueOf(b);

    if(aObj instanceof Integer) {
      System.out.println("An object of Integer is created.");
    }

    if(bObj instanceof Double) {
      System.out.println("An object of Double is created.");
    }
  }
}
```

Wrapper Class…

In the above example, we have used the **intValue() and doubleValue()** method to convert the Integer and Double objects into corresponding primitive types.

However, the Java compiler can automatically convert objects into corresponding primitive types. For example,

Wrapper Class…

```
Integer aObj = Integer.valueOf(2);
// converts into int type
int a = aObj;

Double bObj = Double.valueOf(5.55);
// converts into double type
double b = bObj;
```

This process is known as **unboxing.**

Wrapper Class…

```
int a = 56;

// autoboxing
Integer aObj = a;
```

Autoboxing has a great advantage while working with Java collections.

Wrapper Class…

```
import java.util.ArrayList;

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

    ArrayList<Integer> list = new ArrayList<>();

    //autoboxing
    list.add(5);
    list.add(6);

    System.out.println("ArrayList: " + list);
  }
}
```

Wrapper Class…

In the previous example, we have created an array list of Integer type. Hence the array list can only hold objects of **Integer type**.

Notice the line,

list.add(5);

Here, we are passing primitive type value. However, due to autoboxing, the primitive value is automatically converted into an Integer object and stored in the array list.

Unboxing - Wrapper Objects to Primitive Types

In unboxing, the Java compiler automatically converts wrapper class objects into their corresponding primitive types. For example,

```
// autoboxing
Integer aObj = 56;

// unboxing
int a = aObj;
```

Wrapper Class—example of unboxing

```
import java.util.ArrayList;
class Main {
  public static void main(String[] args) {

    ArrayList<Integer> list = new ArrayList<>();

    //autoboxing
    list.add(5);
    list.add(6);

    System.out.println("ArrayList: " + list);

    // unboxing
    int a = list.get(0);
    System.out.println("Value at index 0: " + a);
  }
}
```

Wrapper Class…

From previous example, notice the line,

int a = list.get(0);

Here, the get() method returns the object at index 0. However, due to unboxing, the object is automatically converted into the primitive type int and assigned to the variable a.

THANK YOU

Ms. Archana A
Department of Computer Applications

archana@pes.edu

+91 80 6666 3333 Extn 392