LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL

CS-306

JAVA

Vision of the Department

To be recognized for keeping innovation, research and excellence abreast of learning in the field of computer science & engineering to cater the global society.

Mission of the Department

M1: To provide an exceptional learning environment with academic excellence in the field of computer science and engineering.

M2: To facilitate the students for research and innovation in the field of software, hardware and computer applications and nurturing to cater the global society.

M3: To establish professional relationships with industrial and research organisations to enable the students to be updated of the recent technological advancements.

M4: To groom the learners for being the software professionals catering the needs of modern society with ethics, moral values and full of patriotism.

Program Educational Objectives (PEO's)

PEO1: The graduate will have the knowledge and skills of major domains of computer science and engineering in providing solution to real world problems most efficiently.

PEO2: The graduate will be able to create and use the modern tools and procedures followed in the software industry in the relevant domain.

PEO3: The graduate will be following the ethical practices of the software industry and contributing to the society as a responsible citizen.

PEO4: The graduate will have the innovative mindset of learning and implementing the latest developments and research outcomes in the computer hardware and software to keep pace with the fast changing socio economic world.

LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL

CS-306

JAVA

LAB OUTCOMES

Student will be able to

CO1: Elaborate basic architecture of JAVA and capabilities of Java Language.

CO2: Illustrate basic concepts of object oriented programming and apply these concepts with the help of Java Language.

CO3: Update and retrieve the data from the database using JDBC connectivity.

CO4: Develop the graphical user interaction programs.

CO5: Demonstrate development of web based applications with the help of servlets and JSP.

LIST OF EXPERIMENTS

1. Installation of J2SDK
2. Write a program to show Scope of Variables.
3. Write a program to show Concept of CLASS in JAVA.
4. Write a program to show Type Casting in JAVA
5. Write a program to show How Exception Handling is in JAVA.
6. Write a Program to show Inheritance.
7. Write a program to show Polymorphism.
8. Write a program to show Access Specifier (Public, Private, Protected) in JAVA
9. Write a program to add a Class to a Package.
10. Write a program to hide a Class.

LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL

CS-306

JAVA

EXPERIMENT -1

JAVA:

Java is a multi-platform, object-oriented, and network-centric language that can be used as a platform in itself. It is a fast, secure, reliable programming language for coding everything from mobile apps and enterprise software to big data applications and server-side technologies.

INSTALLATION OF J2SDK:

Introduction:

The Java Development Kit (**JDK**) is software used for Java programming, along with the Java Virtual Machine (**JVM**) and the Java Runtime Environment (**JRE**). The JDK includes the compiler and class libraries, allowing developers to create Java programs executable by the JVM and JRE.

Download Java for Windows 10

Download the latest Java Development Kit installation file for Windows 10 to have the latest features and bug fixes.

1. Using your preferred web browser, navigate to the Oracle Java Downloads page.
2. On the *Downloads* page, click the **x64 Installer** download link under the **Windows** category. At the time of writing this article, Java version 17 is the latest long-term support Java version.

Linux macOS Windows

Product/file description	File size	Download
x64 Compressed Archive	170.66 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.zip (sha256)
x64 Installer	152 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.exe (sha256)
x64 MSI Installer	150.89 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.msi (sha256)

Wait for the download to complete.

Install Java on Windows 10

After downloading the installation file, proceed with installing Java on your Windows system.

Follow the steps below:

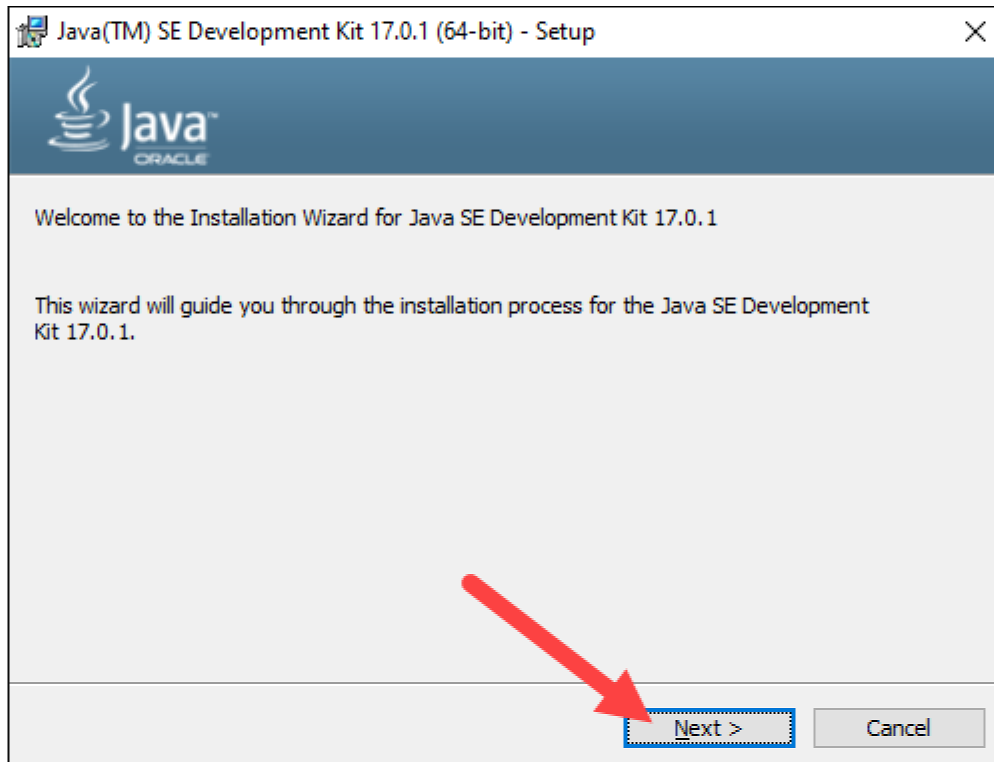
Step 1: Run the Downloaded File

Double-click the **downloaded file** to start the installation.

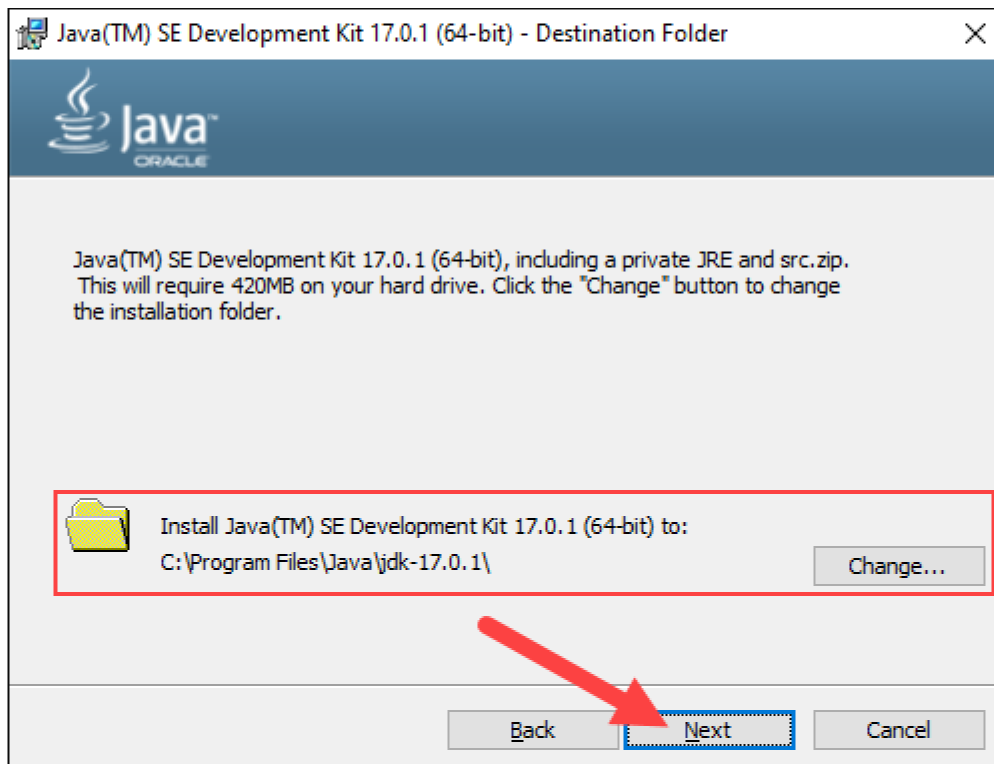
Step 2: Configure the Installation Wizard

After running the installation file, the installation wizard welcome screen appears.

1. Click **Next** to proceed to the next step.



2. Choose the destination folder for the Java installation files or stick to the default path. Click **Next** to proceed.



3. Wait for the wizard to finish the installation process until the *Successfully Installed* message appears. Click **Close** to exit the wizard.

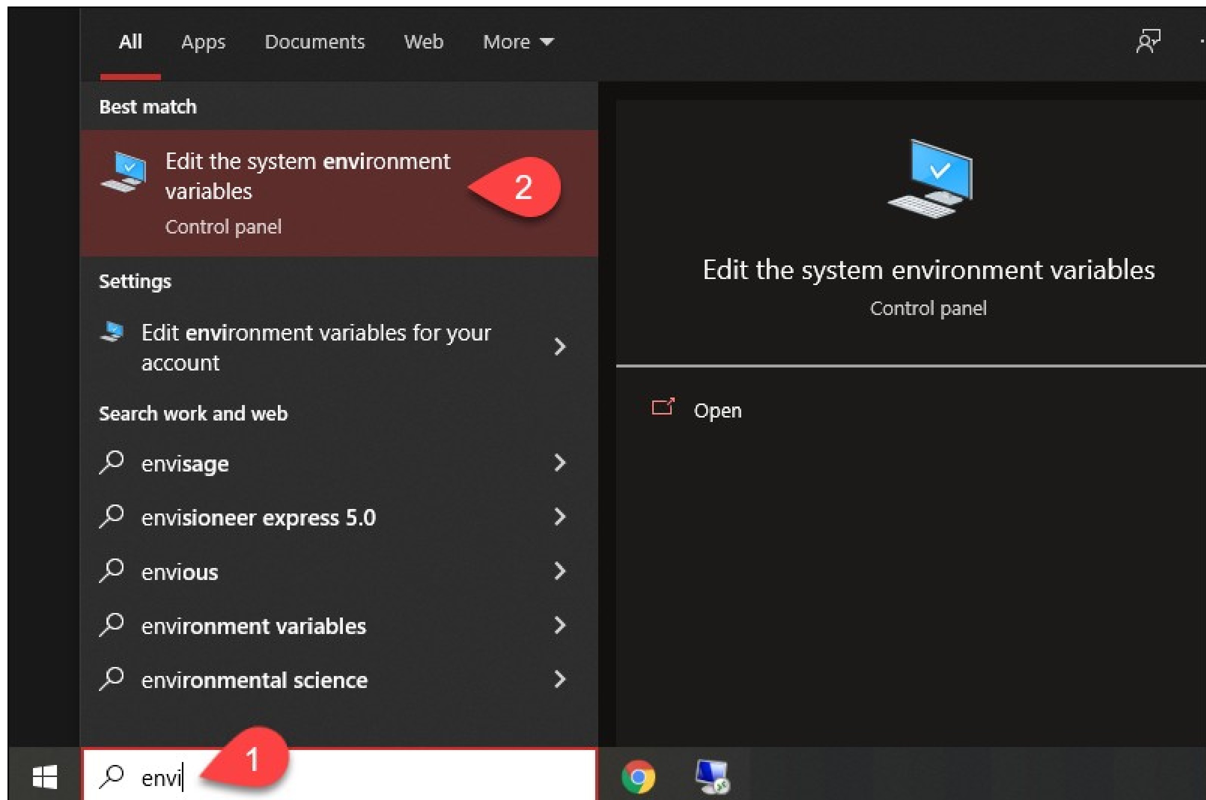
Set Environmental Variables in Java

Set Java environment variables to enable program compiling from any directory. To do so, follow the steps below:

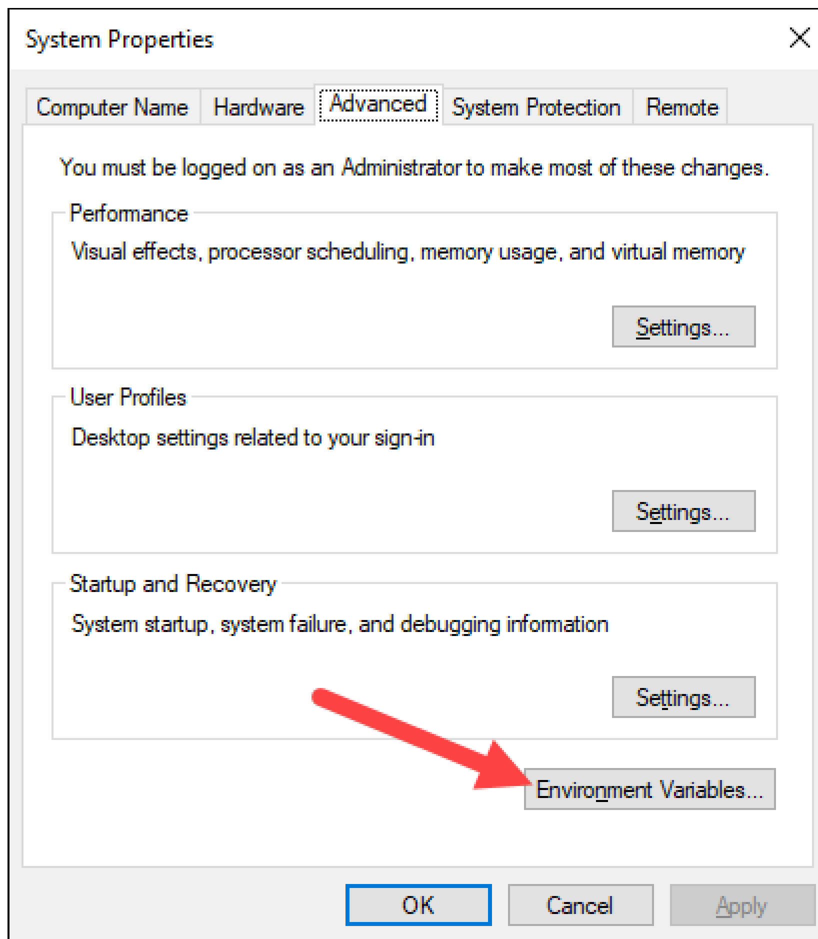
Step 1: Add Java to System Variables

1. Open the **Start** menu and search for *environment variables*.

2. Select the **Edit the system environment variables** result.



3. In the *System Properties* window, under the *Advanced* tab, click **Environment Variables…**



4. Under the *System variables* category, select the **Path** variable and click **Edit**:

Environment Variables

User variables for Boško

Variable	Value
OneDrive	C:\Users\Sneško\Desktop\OneDrive
OneDriveConsumer	C:\Users\Sneško\Desktop\OneDrive
Path	C:\Program Files\MySQL\MySQL Shell 8.0\bin\;C:\Users\Sneško\Ap...
TEMP	C:\Users\Sneško\AppData\Local\Temp
TMP	C:\Users\Sneško\AppData\Local\Temp

New... Edit... Delete

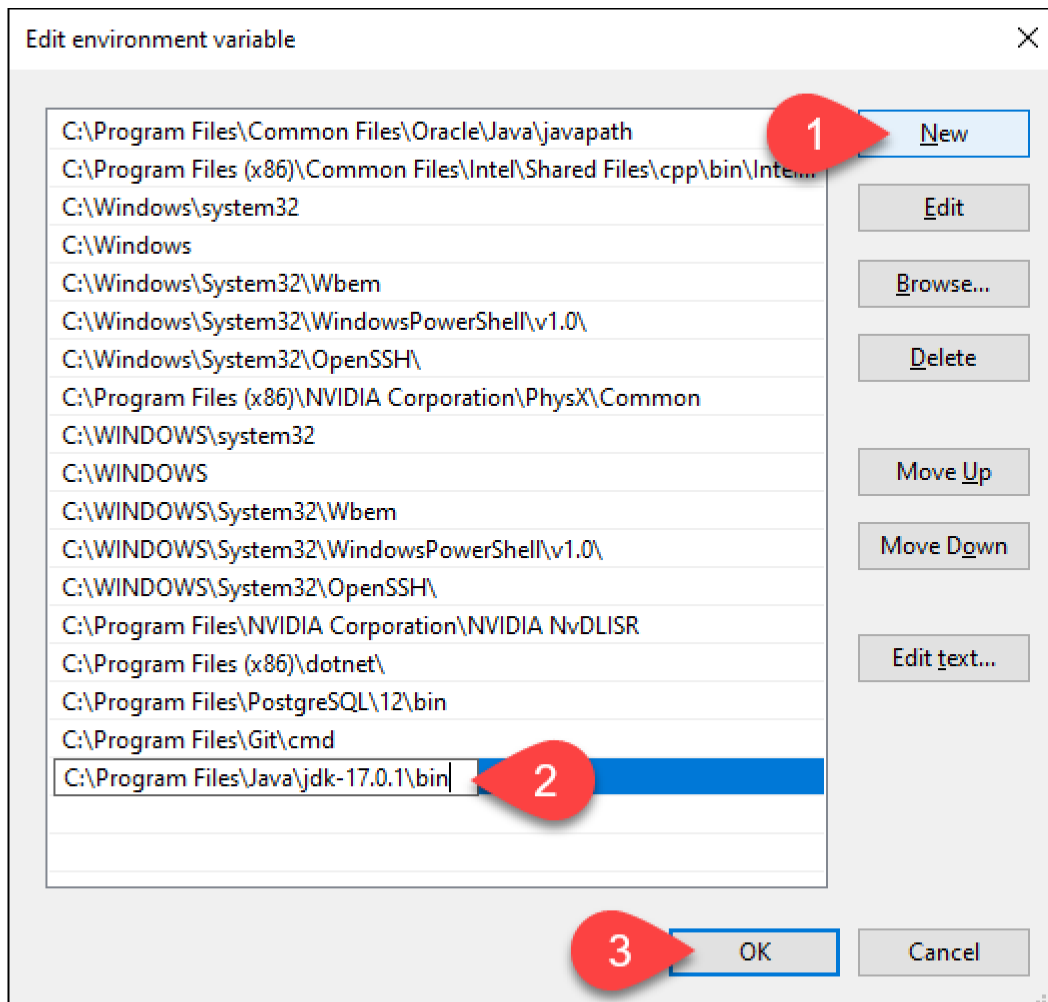
System variables

Variable	Value
NUMBER_OF_PROCESSORS	8
OS	Windows_NT
Path	C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program ...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64
PROCESSOR_IDENTIFIER	AMD64 Family 23 Model 8 Stepping 2, AuthenticAMD
PROCESSOR_LEVEL	23

New... Edit... Delete

OK Cancel

5. Click the **New** button and enter the path to the Java bin directory:



Note: The default path is usually *C:\Program Files\Java\jdk-17.0.1\bin*.

6. Click **OK** to save the changes and exit the variable editing window.

Step 2: Add JAVA_HOME Variable

Some applications require the **JAVA_HOME** variable. Follow the steps below to create the variable:

1. In the *Environment Variables* window, under the *System variables* category, click the **New…** button to create a new variable.

Environment Variables

User variables for Boško

Variable	Value
OneDrive	C:\Users\Snesko\Desktop\OneDrive
OneDriveConsumer	C:\Users\Snesko\Desktop\OneDrive
Path	C:\Program Files\MySQL\MySQL Shell 8.0\bin\;C:\Users\Sneško\Ap...
TEMP	C:\Users\Snesko\AppData\Local\Temp
TMP	C:\Users\Snesko\AppData\Local\Temp

New... Edit... Delete

System variables

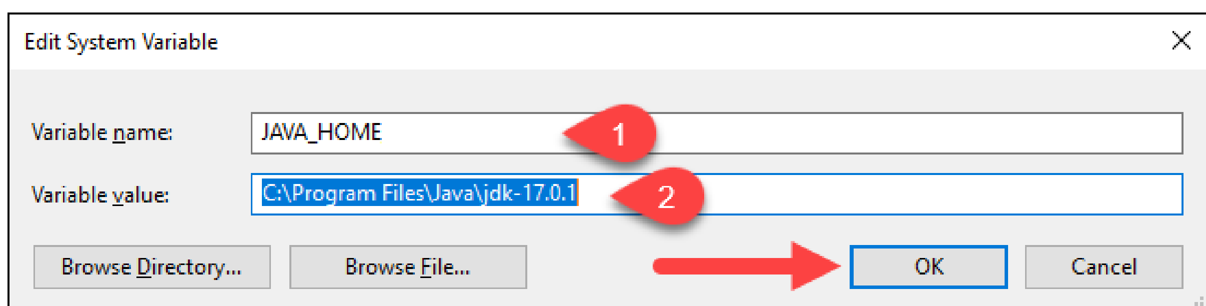
Variable	Value
C_EM64T_REDIST11	C:\Program Files (x86)\Common Files\Intel\Shared Files\cpp\
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	8
OS	Windows_NT
Path	C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program ...

New... Edit... Delete

OK Cancel

2. Name the variable as ***JAVA_HOME***.

3. In the variable value field, paste the path to your Java jdk directory and click **OK**.



4. Confirm the changes by clicking **OK** in the *Environment Variables* and *System properties* windows.

Test the Java Installation

Run the **java -version** command in the command prompt to make sure Java installed correctly:

C:\Users\boskom>java -version

Java version "17.0.1" 2021-10-19 LTS

Java(TM) SE Runtime Environmental (build 17.0.1+12-LTS-39)

Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)

If installed correctly, the command outputs the Java version. Make sure everything works by writing a simple program and compiling it.

EXPERIMENT -2

VARIABLE AND SCOPE:

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e. Scope of a variable can determined at compile time and independent of function call stack. Java programs are organized in the form of classes. Every class is part of some package. Java scope rules can be covered under following categories.

Member Variables (Class Level Scope):

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
public class Test
{
   // All variables defined directly inside a class
   // are member variables
   int a;
   private String b;
   void method1() {....}
   int method2() {....}
   char c;
}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't affect scope of them within a class.
- Member variables can be accessed outside a class with following rules

Modifier Package Subclass World

public	Yes	Yes	Yes
protected	Yes	Yes	No
Default (no modifier)	Yes	No	No
private	No	No	No

Variables declared inside a method have method level scope and can't be accessed outside the method.

```
public class Test
{
   void method1()
   {
     // Local variable (Method level scope)
     int x;
   }
}
```

Note : Local variables don't exist after method's execution is over.
Here's another example of method scope, except this time the variable got passed in as a parameter to the method:

```
class Test
{
   private int x;
   public void setX(int x)
   {
this.x = x;
   }
}
```

The above code uses this keyword to differentiate between the local and class variables.
As an exercise, predict the output of following Java program.

- Java

```
publicclassTest
{
   staticintx = 11;
   privateinty = 33;
   publicvoidmethod1(intx)
   {
      Test t = newTest();
      this.x = 22;
      y = 44;

      System.out.println("Test.x: "+ Test.x);
      System.out.println("t.x: "+ t.x);
      System.out.println("t.y: "+ t.y);
      System.out.println("y: "+ y);
   }

   publicstaticvoidmain(String args[])
   {
      Test t = newTest();
      t.method1(5);
   }
}
```

Output:

Test.x: 22

t.x: 22

t.y: 33

y: 44

Loop Variables (Block Scope)

A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

- Java

```
publicclassTest
{
   publicstaticvoidmain(String args[])
   {
      {
         // The variable x has scope within
         // brackets
         intx = 10;
         System.out.println(x);
```

```
        }

        // Uncommenting below line would produce
        // error since variable x is out of scope.

        // System.out.println(x);
    }
}
```

Output:
10

STUDENT ASSIGNMENT PROGRAMS:

1. Tell whether the below code will run or not.

```
classTest {

publicstaticvoidmain(String args[])
{
for(inti = 1; i<= 10; i++) {
System.out.println(i);
}
inti = 20;
System.out.println(i);
}
}
```

EXPERIMENT- 3

CLASS IN JAVA:

1. Class is a set of object which shares common characteristics/ behavior and common properties/ attributes.
2. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
3. Class does not occupy memory.
4. Class is a group of variables of different data types and a group of methods.

A class in java can contain:

- data member
- method
- constructor
- nested class and
- interface

Syntax to declare a class:

```
access_modifier class<class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

Example:

- Animal
- Student
- Bird
- Vehicle
- Company

```
classStudent {
    intid; // data member (also instance variable)
    String name; // data member (also instance variable)

    publicstaticvoidmain(String args[])
    {
        Student s1 = newStudent(); // creating an object of
                                   // Student
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}
```

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. Modifiers: A class can be public or has default access (Refer this for details).

2. Class keyword: class keyword is used to create a class.

3. Class name: The name should begin with an initial letter (capitalized by convention).

4. Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

5. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

6. Body: The class body is surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.
There are various types of classes that are used in real-time applications such as nested classes, anonymous classes, lambda expressions.

EXPERIMENT – 4

JAVA TYPE CASTING:

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss type casting and its types with proper examples.

Type casting

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.

- The target type must be larger than the source type.

1. **byte -> short -> char -> int -> long -> float -> double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

WideningTypeCastingExample.java

```
1. public class WideningTypeCastingExample
2. {
3. public static void main(String[] args)
4. {
5. int x = 7;
6. //automatically converts the integer type into long type
7. long y = x;
8. //automatically converts the long type into float type
9. float z = y;
10. System.out.println("Before conversion, int value "+x);
11. System.out.println("After conversion, long value "+y);
12. System.out.println("After conversion, float value "+z);
13. }
14. }
```

Output

Before conversion,the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

1. **double -> float -> long -> int -> char -> short -> byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
1.  public class NarrowingTypeCastingExample
2.  {
3.  public static void main(String args[])
4.  {
5.  double d = 166.66;
6.  //converting double data type into long data type
7.  long l = (long)d;
8.  //converting long data type into int data type
9.  int i = (int)l;
10. System.out.println("Before conversion: "+d);
11. //fractional part lost
12. System.out.println("After conversion into long type: "+l);
13. //fractional part lost
14. System.out.println("After conversion into int type: "+i);
15. }
16. }
```

Output

Before conversion: 166.66

After conversion into long type: 166

After conversion into int type: 166

EXPERIMENT – 5

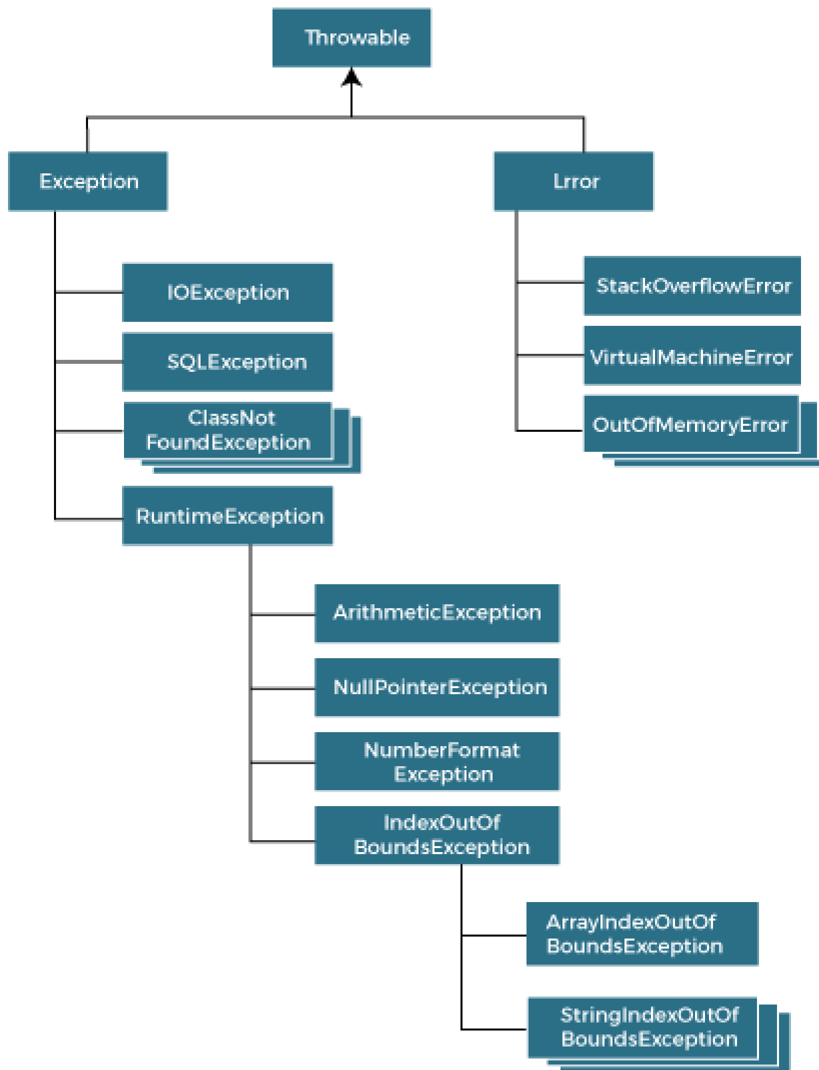
Exception Handling is in JAVA:

The ExceptionHandlinginJava is one of the powerful mechanismto handle the runtimeerrors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions:

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception:

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception:

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error:

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code . . .

EXPERIMENT – 6

INHERITANCE:

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, "extends" refers to increased functionality.

Syntax :

```
class derived-class extends base-class
{
 //methods and fields
}
```

Inheritance in Java Example

In the below example of inheritance, class Employee is a base class, class Engineer is a derived class that extends the Employee class and class Test is a driver class to run the program.

```
importjava.io.*;


// Base or Super Class
classEmployee {
   intsalary = 60000;
}

// Inherited or Sub Class
classEngineer extendsEmployee {
   intbenefits = 10000;
}

// Driver Class
classGfg {
   publicstaticvoidmain(String args[])
   {
      Engineer E1 = newEngineer();
      System.out.println("Salary : "+ E1.salary
                   + "\nBenefits : "+ E1.benefits);
   }
}
```

Output:

Salary : 60000

Benefits : 10000

In practice, inheritance, and polymorphism are used together in Java to achieve fast performance and readability of code.

Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

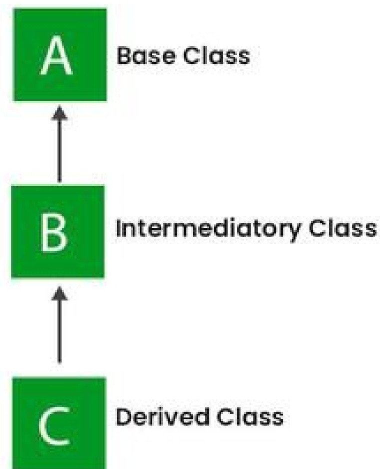
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

1. Single Inheritance

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.

2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

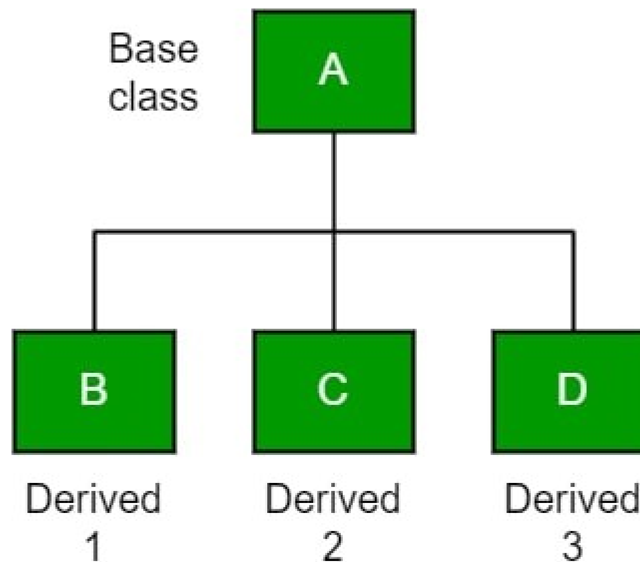


3.

Multilevel Inheritance

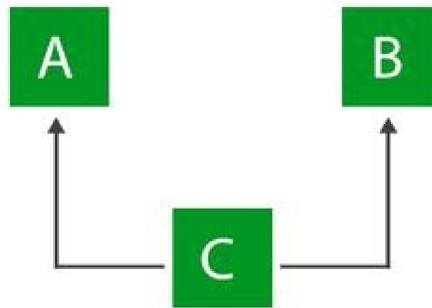
Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



4. Multiple Inheritance (Through Interfaces)

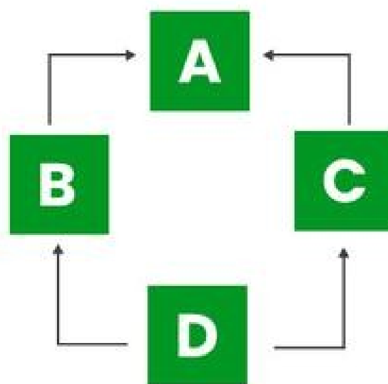
In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

5. Hybrid Inheritance(Through Interfaces)

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

1. Single Inheritance

```
importjava.io.*;
importjava.lang.*;
importjava.util.*;

// Parent class
classone {
   publicvoidprint_geek()
   {
      System.out.println("Geeks");
   }
}

classtwo extendsone {
   publicvoidprint_for() { System.out.println("for"); }
}

// Driver class
publicclassMain {
     // Main function
   publicstaticvoidmain(String[] args)
   {
      two g = newtwo();
      g.print_hello ();
      g.print_my();
      g.print_friends();
   }
}
```

Output:

```
hello
my
friends
```

LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL

CS-306

JAVA

EXPERIMENT – 7

POLYMORPHISM:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real-life Illustration: Polymorphism
A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.

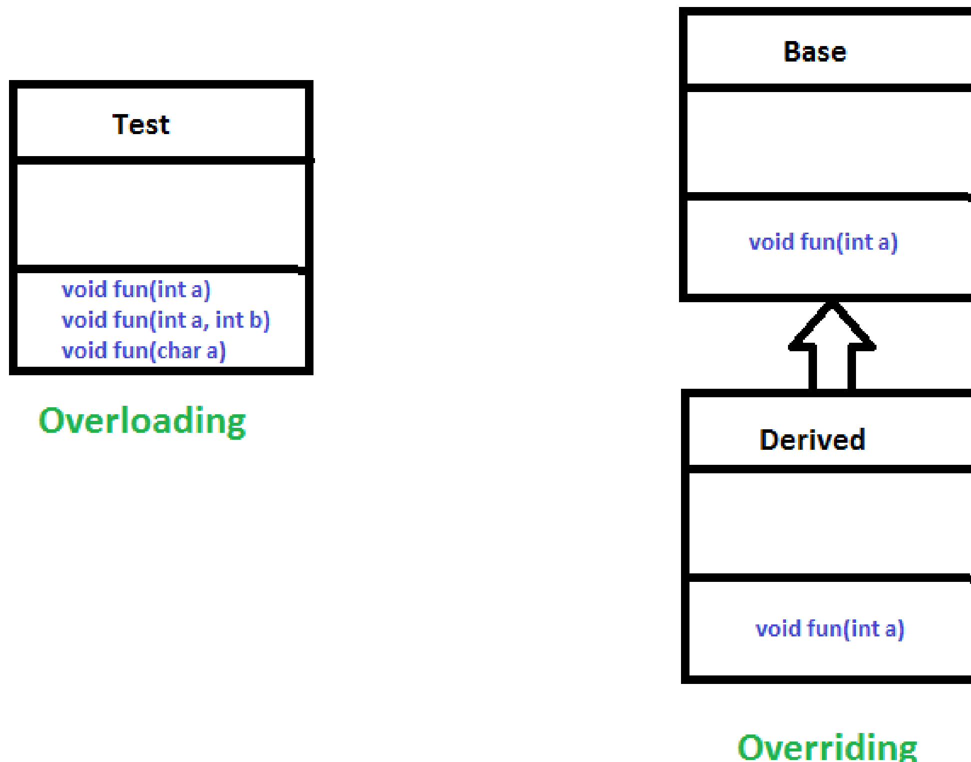
Types of polymorphism

In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

Type 1: Compile-time polymorphism
It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

***Note:** But Java doesn't support the Operator Overloading.*



Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

```
classHelper {

    // Method with 2 integer parameters
    staticintMultiply(inta, intb)
    {

        // Returns product of integer numbers
        returna * b;
    }

    // Method 2
    // With same name but with 2 double parameters
    staticdoubleMultiply(doublea, doubleb)
    {

        // Returns product of double numbers
        returna * b;
    }
}

// Class 2
// Main class
classGFG {

    // Main driver method
    publicstaticvoidmain(String[] args)
    {

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

Output:

8

34.65

Subtypes of Compile-time Polymorphism:

1. **Function Overloading**: It is a feature in C++ where multiple functions can have the same name but with different parameter lists. The compiler will decide which function to call based on the number and types of arguments passed to the function.
2. **Operator Overloading:** It is a feature in C++ where the operators such as +, -, * etc. can be given additional meanings when applied to user-defined data types.
3. **template:** it is a powerful feature in C++ that allows us to write generic functions and classes. A template is a blueprint for creating a family of functions or classes.

Type 2: Runtime polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. **Method overriding**, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Example:

```
classParent {

   // Method of parent class
   voidPrint()
   {

      // Print statement
      System.out.println("parent class");
   }
}

// Class 2
// Helper class
classsubclass1 extendsParent {

   // Method
   voidPrint() { System.out.println("subclass1"); }
}

// Class 3
// Helper class
classsubclass2 extendsParent {

   // Method
   voidPrint()
   {
```

```
        // Print statement
        System.out.println("subclass2");
    }
}

// Class 4
// Main class
classGFG {

    // Main driver method
    publicstaticvoidmain(String[] args)
    {

        // Creating object of class 1
        Parent a;

        // Now we will be calling print methods
        // inside main() method

        a = newsubclass1();
        a.Print();

        a = newsubclass2();
        a.Print();
    }
}
```

Output:

subclass1

subclass2

Output explanation:

Here in this program, When an object of child class is created, then the method inside the child class is called. This is because The method in the parent class is overridden by the child class. Since The method is overridden, This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

Subtype of Run-time Polymorphism:

1. **Virtual functions:** It allows an object of a derived class to behave as if it were an object of the base class. The derived class can override the virtual function of the base class to provide its own implementation. The function call is resolved at runtime, depending on the actual type of the object.

Diagram –

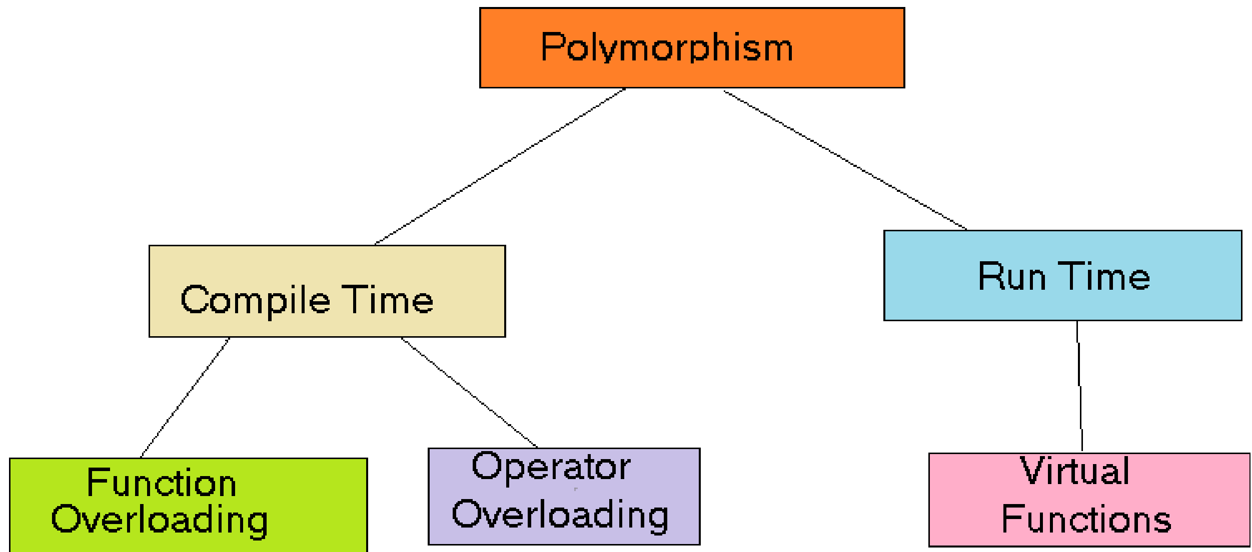


Fig – Types of polymorphism

Polymorphism in Java is a concept that allows objects of different classes to be treated as objects of a common class. It enables objects to behave differently based on their specific class type.

Advantages of Polymorphism in Java:

1. Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
2. Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.
3. Supports dynamic binding, enabling the correct method to be called at runtime, based on the actual class of the object.
4. Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

Disadvantages of Polymorphism in Java:

1. Can make it more difficult to understand the behavior of an object, especially if the code is complex.
2. May lead to performance issues, as polymorphic behavior may require additional computations at runtime.

EXPERIMENT – 8

Access Specifier (Public, Private, Protected) in JAVA:

in Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element. Let us learn about Java Access Modifiers, their types, and the uses of access modifiers in this article.

Types of Access Modifiers in Java:

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

Program 1:

```
// Java program to illustrate default modifier
packagep1;

// Class Geek is having Default access modifier
classGeek
{
   voiddisplay()
   {
      System.out.println("Hello World!");
   }
}
```

Program 2:

```
// Java program to illustrate error while

// using class from different package with
// default modifier
packagep2;
```

```
importp1.*;

// This class is having default access modifier
classGeekNew
{
   publicstaticvoidmain(String args[])
   {
      // Accessing class Geek from package p1
      Geek obj = newGeek();

      obj.display();
   }
}
```

Output:

```
Compile time error
```

2. Private Access Modifier

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because
- private means “only visible within the enclosing class”.
- protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
packagep1;

classA
{
privatevoiddisplay()
   {
      System.out.println("GeeksforGeeks");
   }
}
```

```
classB
{
publicstaticvoidmain(String args[])
    {
        A obj = newA();
        // Trying to access private method
        // of another class
        obj.display();
    }
}
```

Output:

error: display() has private access in A

obj.display();

3. Protected Access Modifier

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

Program 1:

```
// Java program to illustrate
// protected modifier
packagep1;

// Class A
publicclassA
{
protectedvoiddisplay()
    {
        System.out.println("hello friends");
    }
}
```

Program 2:

```
// Java program to illustrate
// protected modifier
packagep2;
```

```
import p1.*; // importing all classes in package p1

// Class B is subclass of A
class B extends A
{
public static void main(String args[])
{
    B obj = new B();
    obj.display();
}

}
```

Output:

hello friends

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package N0on-Subclass	No	No	No	Yes

Expirement-9

Class:

A class in Java is a logical template to create objects that share common properties and methods. Hence, all objects in a given class will have the same methods or properties.

Class to a Package:

The packages are used for categorization of the same type of classes and interface in a single unit. There is no core or in-built classes that belong to unnamed default package. To use any classes or interface in other class, we need to use it with their fully qualified type name. **But** some time, we've the need to use all or not all the classes or interface of a package then it's a tedious job to use in such a way discussed. Java supports *imports* statement to bring entire package, or certain classes into visibility. **It** provides flexibility to the programmer to save a lot of time just by importing the classes in his/her program, instead of rewriting them.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and must be before of any class definitions. This is the general form is as follows:

Built-in Packages:

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website:

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

Syntax:

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

program:

```
package p1;
class C1()
{
public void m1()
{
System.out.println("m1 of C1");
}
public static void main(string args[])
{
C1 obj = new C1();
obj.m1();
}
}
```

Experiment- 10

Hiding Class

Hidden classes are classes that cannot be used directly by the bytecode or other classes. Even though it's mentioned as a class, it should be understood to mean either a hidden class or interface. It can also be defined as a member of the access control nest and can be unloaded independently of other classes

How to Write Hidden Classes

The hidden class is created by invoking

Lookup::defineHiddenClass

. It causes the JVM to derive a hidden class from the supplied bytes, links the hidden class, and returns a Lookup object that provides reflective access to the hidden class.

The following are 4 steps for creating and using hidden classes.

1.Create Lookup object. Get a lookup object, which will be used to create hidden class in the next steps.

MethodHandles.Lookup lookup = MethodHandles.lookup();

2**.Create class bytes using ASM**. We are using the byte code manipulation library ASM. We create ClassWriter object using helper class GenerateClass. If you look at the details in GenerateClass, this class implements interface Test, which we will use in further steps.

ClassWritercw = GenerateClass.getClassWriter(HiddenClassDemo.class);

byte[] bytes = cw.toByteArray();

3.Define hidden class. In this step, we are creating a hidden class. It is important to note the invoking program should store the lookup object carefully since it is the only way to obtain the Class object of the hidden class.

Class<?> c = lookup.defineHiddenClass(bytes, true, NESTMATE).lookupClass();

4.Use hidden class. In this step, we are using reflection to access the hidden class. First, create the constructor, then create an object using this, typecast this object to interface Test and call method test. This method ignores the argument passed and prints "Hello test" to the console.

```
Constructor<?> constructor = c.getConstructor(null);

Object object = constructor.newInstance(null);

Test test = (Test) object;

test.test(new String[]{""});
```

How to Hide Class in a Package

When we import a package using astric(*),all public classes are imported.however ,we may preffer to “not import”certainclasses.i.e,we may like to hide these classes from accessing from outside of the package.such classes should be declared”not public”.

EX:

```
package p1;
public class X
{
Body of X
}


class Y
{
Body of Y
}
```

Here, the class y which is not declared public is hidden from out side of the package p1. this class can be seen and used only by other classes in the same package note that a java source file should contain only one public class and may include any number of non public classes.

Now ,consider the following code ,which imports the package p1 that contains classes X and Y:

```
import p1.*;

X objectX;

Y objectY;
```

Java compiler would generate an error message for this code because the class Y,which has not been declared public,is not imported and therefore not available for creating its objects.