Java solution

Anonymous Class in Java

Introduction

The anonymous class in Java is a way to create an unnamed class that extends a superclass or implements an interface on the fly. They are often used for one-time implementations or event handlers. Anonymous classes are defined and instantiated in a single step, providing a concise and inline approach for certain scenarios. Learn more about anonymous classes in this tutorial.

Overview

In this tutorial, we'll explore how to use anonymous class in Java with examples. You will also learn h`ow to extend a superclass or implement an interface, providing a concise and inline approach for certain programming scenarios.

Syntax of Anonymous Class in Java

Here is the syntax for using the anonymous class in Java:

```
SuperClass obj = new SuperClass() {
    // Anonymous inner class body
    // Implementations of methods or additional members
};
```

Let us break down the syntax:

- We start by declaring a variable of the superclass type and assigning it to a new superclass instance.
- We then use the open curly braces {} to define the body of the anonymous inner class.
- Inside the anonymous inner class body, we can provide implementations for methods or add additional members as needed. These implementations and members are specific to the anonymous inner class and do not affect the superclass or other classes.

Java Anonymous Inner Class Example Using Class

```
public class upGradTutorials {
    public static void main(String[] args) {
        // Creating an instance of the class using anonymous inner class
        MyClass myClass = new MyClass() {
            @Override
            public void displayMessage() {
                System.out.println("Hello, I'm an anonymous inner class!");
            }
        };
        // Calling the method on the anonymous inner class object
        myClass.displayMessage();
    }
}
// Class to be extended by anonymous inner class
class MyClass {
    public void displayMessage() {
        System.out.println("Hello, I'm the base class!");
    }
}
```

This example has a base class **MyClass** with a method **displayMessage()**. We create an instance of this class using an anonymous inner class. The anonymous inner class is defined and instantiated inline without explicitly declaring a separate class.

We override the **displayMessage()** method inside the anonymous inner class and provide our custom implementation. When we call the **displayMessage()** method on the anonymous inner class object, it executes the overridden method, printing "Hello, I'm an anonymous inner class!" to the console.

Internal Class Generated By The Compiler

When we compile the Java code containing an anonymous inner class, the compiler generates a separate class file for the anonymous inner class. The generated class file has a name that combines the outer class's name, a dollar symbol (**$**), and a number. The number represents the order of appearance of the anonymous inner class in the code.

For example, if we compile the above code, the compiler will generate a class file named **upGradTutorial$1.class** for the anonymous inner class.

Java Anonymous Inner Class Example Using Interface

```
public class upGradTutorials {
    public static void main(String[] args) {
        // Creating an instance of the interface using an anonymous inner class
        MyInterface obj = new MyInterface() {
            @Override
            public void doSomething() {
                System.out.println("Anonymous inner class implementation of doSomething method");
            }
        };
            // Calling the method using the interface reference
        obj.doSomething();
    }}
// Interface definition
interface MyInterface {
    void doSomething(); }
```

Inside the **main** method, we create an instance of an anonymous inner class that implements the **MyInterface** interface. The anonymous inner class is defined using the **new MyInterface() { ... }** syntax.

We implement the **doSomething()** method within the anonymous inner class, which simply prints the message "Anonymous inner class implementation of doSomething method".

After creating the anonymous inner class instance, we assign it to a variable **obj** of type **MyInterface**. This allows us to reference the object and call methods defined in the interface.

Finally, we invoke the **doSomething()** method on the **obj** object, which calls the overridden **doSomething()** method within the anonymous inner class and prints the desired message to the console.

Internal Class Generated By The Compiler

The Java compiler generates a class for the anonymous inner class behind the scenes, with a name similar to **upGradTutorials$1**. This internal class contains the implementation of the interface method. However, as a developer, we do not need to explicitly create this class, as the compiler automatically generates it.

The generated class is a subtype of the interface or superclass that the anonymous inner class implements or extends. In this case, the compiler generates a class file that implements the **MyInterface** interface.

The generated class has a synthetic name, meaning it is not explicitly named in the source code. It allows us to create an interface instance without explicitly defining a separate class that implements the interface. The anonymous inner class simplifies the code by providing a concise way to implement interfaces or extend classes on-the-fly without the need for a separate named class implementation.

Types of Anonymous Inner Class

Anonymous inner classes in Java can be classified into two types:

- **Anonymous inner classes that extend a class:** These classes are created and instantiated simultaneously without explicitly defining a separate class. They can override methods and access variables of the superclass. Such classes are useful for event handling and providing custom implementations for abstract classes.
- **Anonymous inner classes that implement an interface:** Similar to the previous type, these classes are defined and instantiated inline without explicitly creating a separate class.

They provide implementations for methods defined in the interface. These classes are commonly used for situations requiring one-time implementations, such as listeners or callbacks.

Example: Anonymous Class Extending a Class

```
public class upGradTutorials {
    public static void main(String[] args) {
        // Creating an instance of the abstract class using an anonymous inner class
        AbstractClass obj = new AbstractClass() {
            @Override
            public void display() {
                System.out.println("Anonymous inner class implementation of display method");
            }
            @Override
            public void additionalMethod() {
                System.out.println("Additional method implementation in the anonymous inner class");
            }
        };
        // Calling methods using the abstract class reference
        obj.display();
        obj.additionalMethod();
    }
}
// Abstract class definition
abstract class AbstractClass {
    public abstract void display();
```

```
    public void additionalMethod() {
        System.out.println("Default implementation of additionalMethod");
    }
}
```

In this example, we define an anonymous inner class that extends the **AbstractClass** abstract class. The anonymous inner class provides implementations for the abstract **display()** method and another method, **additionalMethod()**.

Inside the main method, we create an instance of the **AbstractClass** using the syntax **new AbstractClass() { ... }**. Within the anonymous inner class, we override the **display()** method and provide a custom implementation that prints a specific message. We also override the **additionalMethod()** and provide a custom implementation.

After creating the anonymous inner class instance, we assign it to a variable **obj** of type **AbstractClass**. This allows us to reference the object and call the overridden methods.

Finally, we invoke the **display()** and **additionalMethod()** methods on the **obj** object. Since the anonymous inner class extends the **AbstractClass**, the overridden implementations within the anonymous inner class are executed, printing the custom messages to the console.

Example: Anonymous Class Implementing an Interface

```
public class upGradTutorials {
    public static void main(String[] args) {
        // Creating an instance of the interface using an anonymous inner class
        MyInterface obj = new MyInterface() {
            @Override
            public void printMessage(String message) {
                System.out.println("Anonymous inner class implementation: " + message);
            }
```

```
        @Override
        public void additionalMethod() {
            System.out.println("Additional method implementation in the anonymous inner class");
        }
    };

    // Calling methods using the interface reference
    obj.printMessage("Hello, World!");
    obj.additionalMethod();
  }
}
// Interface definition
interface MyInterface {
  void printMessage(String message);
  void additionalMethod();
}
```

This example defines an anonymous inner class that implements the **MyInterface** interface. The anonymous inner class provides implementations for the **printMessage()** method and another method, **additionalMethod()**.

Inside the main method, we create an instance of the **MyInterface** using the syntax **new MyInterface() { ... }**. Within the anonymous inner class, we override the **printMessage()** method and provide a custom implementation that prints the message with additional text. We also override the **additionalMethod()** and provide a custom implementation.

After creating the anonymous inner class instance, we assign it to a variable **obj** of type **MyInterface**. This allows us to reference the object and call the overridden methods.

Finally, we invoke the **printMessage()** and **additionalMethod()** methods on the **obj** object. Since the anonymous inner class implements the **MyInterface**, the overridden implementations within the anonymous inner class are executed, printing the custom messages to the console.

Advantages of Anonymous Classes

Anonymous classes in Java provide several advantages:

- **Conciseness and Inline Definition:** Anonymous classes allow you to define and instantiate a class in a single step without needing a separate class declaration. This makes the code shorter and eliminates the need for additional class files.
- **Localized Implementation:** Anonymous classes are commonly used for small, specific tasks or one-time implementations. They encapsulate the implementation within the relevant context, making the code more focused and easier to understand.
- **Improved Readability:** By defining a class inline where it is used, anonymous classes make the code more straightforward and comprehensible. The implementation is directly associated with its usage, enhancing code clarity.
- **Access to Local Variables:** Anonymous classes can access local variables from the enclosing method or block in which they are defined. This allows for convenient usage of a local state within the anonymous class.
- **Customization and Flexibility:** Anonymous classes can override methods and provide custom implementations for abstract classes or interfaces. This enables on-the-fly behavior customization, providing flexibility to adapt to specific needs.
- **Event Handling:** Anonymous classes are frequently used for event handling in graphical user interfaces. They offer a convenient way to define and handle events concisely without needing separate event listener classes.

Conclusion

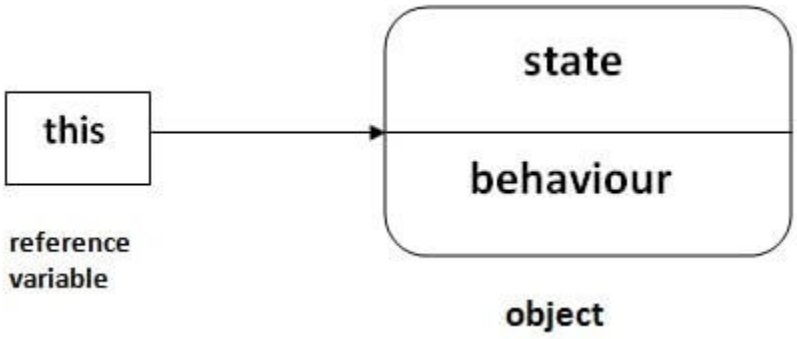
Anonymous class in Java provides a powerful and convenient way to define and instantiate classes inline without separating class declarations. They offer advantages such as conciseness, localized implementations, improved code readability, access to local variables, customization, and flexibility.

Anonymous classes are beneficial for one-time implementations, event handling, and providing custom implementations for abstract classes or interfaces. You can write more concise, focused,

and readable code in Java by leveraging anonymous classes. Sign up for a professional course at upGrad to hone your skills in various Java concepts.

This keyword in Java

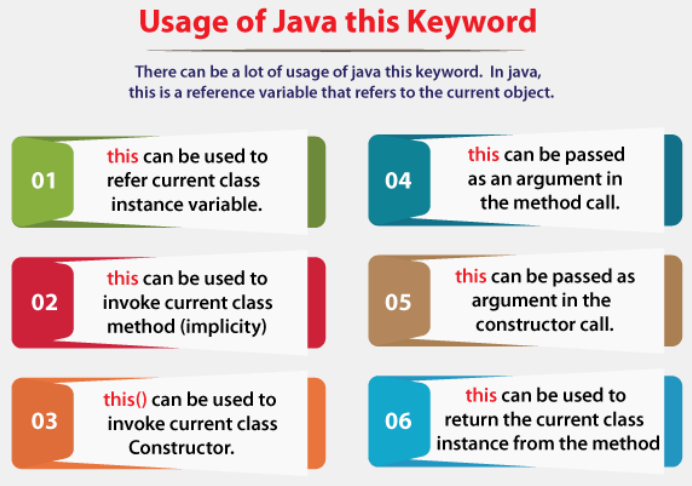
There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.



1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. class TestThis1{
13. public static void main(String args[]){
14. Student s1=new Student(111,"ankit",5000f);
15. Student s2=new Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}
```

Output:

```
0 null 0.0
0 null 0.0
```

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. this.rollno=rollno;
7. this.name=name;
8. this.fee=fee;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis2{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

What is Polymorphism in Java?

The word polymorphism means having many forms. In simple words, we can define Java Polymorphism as the ability of a message to be displayed in more than one form. In this article, we will learn what is polymorphism and it's type.

Real-life Illustration of Polymorphism in Java: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behaviors in different situations. This is called polymorphism.

What is Polymorphism in Java?

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 Java Polymorphism

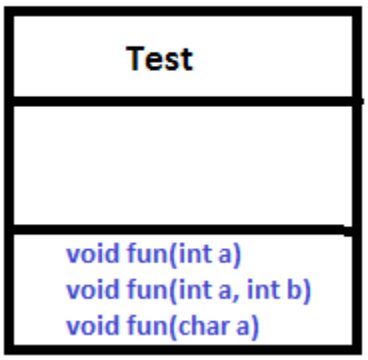
In Java Polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

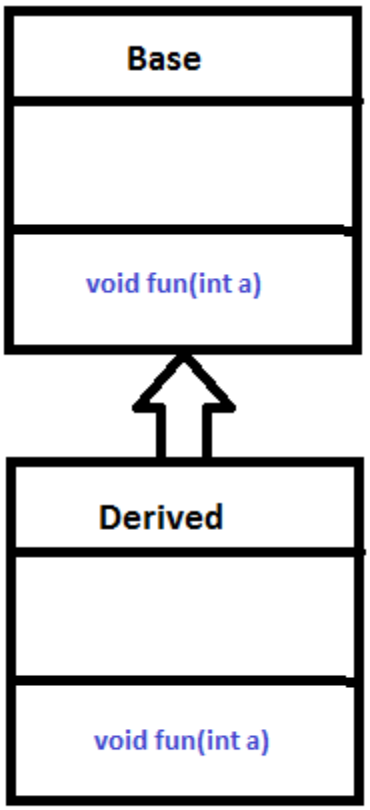
Compile-Time Polymorphism in Java

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



Overloading



Overriding

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.

Example 1:

```
// Java Program for Method overloading

// By using Different Types of Arguments
```

```
// Class 1

// Helper class

class Helper {

    // Method with 2 integer parameters

    static int Multiply(int a, int b)

    {

        // Returns product of integer numbers

        return a * b;

    }

    // Method 2

    // With same name but with 2 double parameters

    static double Multiply(double a, double b)

    {

        // Returns product of double numbers

        return a * b;

    }
}

// Class 2

// Main class
```

```
class GFG {

    // Main driver method

    public static void main(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
```

Example 2:

```
// Java program for Method Overloading

// by Using Different Numbers of Arguments

 // Class 1

// Helper class

class Helper {

     // Method 1
```

```
    // Multiplication of 2 numbers
    static int Multiply(int a, int b)
    {
        // Return product
        return a * b;
    }
    // Method 2
    // // Multiplication of 3 numbers
    static int Multiply(int a, int b, int c)
    {
        // Return product
        return a * b * c;
    }
}
// Class 2
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
```

```
    {

        // Calling method by passing

        // input as in arguments

        System.out.println(Helper.Multiply(2, 4));

        System.out.println(Helper.Multiply(2, 7, 3));

    }

}
```

Output

```
8

42
```

- 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.
- Runtime Polymorphism in Java
- 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.
- ExampleJava

```
// Java Program for Method Overriding



// Class 1

// Helper class

class Parent {



    // Method of parent class

    void Print()

    {

         // Print statement

        System.out.println("parent class");

    }

}



// Class 2

// Helper class

class subclass1 extends Parent {
```

```
    // Method
    void Print() { System.out.println("subclass1"); }
}


// Class 3
// Helper class
class subclass2 extends Parent {


    // Method
    void Print()
    {

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


// Class 4
// Main class
```

```
class GFG {



    // Main driver method

    public static void main(String[] args)

    {

         // Creating object of class 1

        Parent a;

         // Now we will be calling print methods

        // inside main() method



        a = new subclass1();

        a.Print();



        a = new subclass2();

        a.Print();

    }

}
```

Output

```
subclass1

subclass2
```

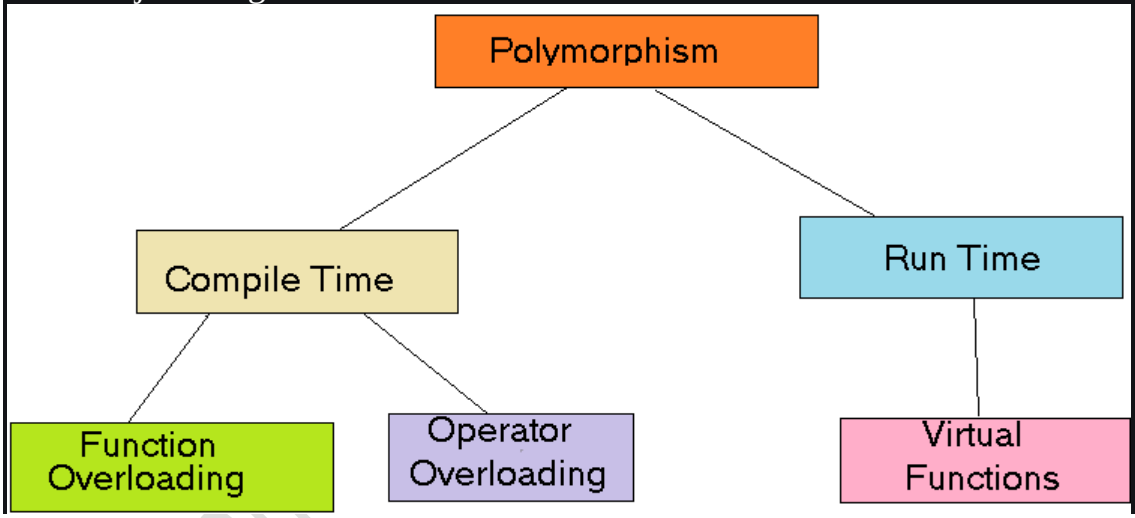
Explanation of the above code:

Here in this program, When an object of a 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

i. 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 –



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. This may lead to performance issues, as polymorphic behavior may require additional computations at runtime.

Naming Convections for Packages?

A programmer is always said to write clean codes, where naming has to be appropriate so that for any other programmer it acts as an easy way out to read the code. At a smaller level, this seems meaningless but think of the industrial level where it becomes necessary to write clean codes in order to save time for which there are certain rules been laid of which one of the factors is to name the keyword right which is termed as a naming convention in Java.

For example when you are using a variable name depicting displacement then it should be named as “displace” or similar likewise not likely x, d which becomes complex as the code widens up and decreases the readability aperture. Consider the below illustrations to get a better understanding which later on we will be discussing in detail.

Illustrations:

Class: If you are naming any class then it should be a noun and so should be named as per the goal to be achieved in the program such as Add2Numbers, ReverseString, and so on not likely A1, Programming, etc. It should be specific pointing what exactly is there inside without glancing at the body of the class.

Interface: If you are naming an interface, it should look like an adjective such as consider the existing ones: Runnable, Serializable, etc. Try to use ‘able’ at the end, yes it is said to try as there are no hard and fast bound rules as if we do consider an

inbuilt interface such as ‘Remote’, it is not having ble at the end. Consider if you are supposed to create an interface to make read operation then it is suggested as per naming conventions in java to name a similar likely ‘Readable’ interface.

Methods: Now if we do look closer a method is supposed to do something that it does contains in its body henceforth it should be a verb.

Constants: As the name suggests it should look like as we read it looks like it is fixed for examples PI, MAX_INT, MIN_INT, etc as follows.

Naming Conventions in Java

In java, it is good practice to name class, variables, and methods name as what they are actually supposed to do instead of naming them randomly. Below are some naming conventions of the java programming language. They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages, and constants.

Camel’s case in java programming consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital. Here in simpler terms, it means if there are two

Note: Do look out for these exceptions cases to camel casing in java as follows:

In package, everything is small even while we are combining two or more words in java

In constants, we do use everything as uppercase and only ‘_’ character is used even if we are combining two or more words in java.

Type 1: Classes and Interfaces

Class names should be nouns, in mixed cases with the first letter of each internal word capitalized. Interfaces names should also be capitalized just like class names.

Use whole words and must avoid acronyms and abbreviations.

Classes: class Student { }

class Integer {}

class Scanner {}

Interfaces : Runnable

Remote

Serializable

Type 2: Methods

Methods should be verbs, in mixed case with the first letter lowercase and with the first letter of each internal word capitalized.

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

As the name suggests the method is supposed to be primarily method which indeed it is as main() method in java is the method from where the program begins its execution.

Type 3: Variables

Variable names should be short yet meaningful.

Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.

Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.

One-character variable names should be avoided except for temporary variables.

Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

int[] marks;

double double answer,

As the name suggests one stands for marks while the other for an answer be it of any e do not mind.

Type 4: Constant variables

Should be all uppercase with words separated by underscores ("_").

There are various constants used in predefined classes like Float, Long, String etc.

num = PI;

Type 5: Packages

The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, like com, edu, gov, mil, net, org.

Subsequent components of the package name vary according to an organization's own internal naming conventions.

java.util.Scanner ;

java.io.*;

As the name suggests in the first case we are trying to access the Scanner class from the java.util package and in other all classes(* standing for all) input-output classes making it so easy for another programmer to identify.

What is an Unchecked Exception in Java ?

It occurs at the time of execution and is known as a run time exception. It includes **bugs**, **improper usage of API**, and **syntax** or **logical error** in programming.

In Java, exceptions that are under **Error** and **, Runtime exception** classes are unchecked exceptions.

Types of Unchecked Exceptions

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException
- InputMismatchException
- IllegalStateException

Arithmetic exception :

It is a type of unchecked error in code that is thrown whenever there is a **wrong mathematical** or **arithmetic calculation** in the code, especially during run time. For instance, when the denominator in a fraction is zero, the arithmetic exception is thrown.

NullPointerException :

It is a **run time exception**. It is thrown when a null value is assigned to a reference object and the program tries to use that null object.

ArrayIndexOutOfBoundsException :

It occurs when we access an array with an invalid index. This means that either the index value is less than zero or greater than that of the array's length.

NumberFormatException :

It is a type of unchecked exception that occurs when we are trying to convert a string to an int or other numeric value. This exception is thrown in cases when it is not possible to convert a string to other numeric types.

InputMismatchException :

It occurs when an input provided by the user is incorrect. The type of incorrect input can be out of range or incorrect data type.

IllegalStateException :

It is a run time exception that occurs when a method of a code is triggered or invoked at the wrong time. This exception is used to give a signal that the method is invoked at the wrong time.

Examples

Example of Arithmetic Exception :

```
public class EgArithemetic {

  void divideInt(int n1, int n2) {
    int res = n1 / n2;
    System.out.println("Output: " + res);
  }

  public static void main(String argvs[]) {
    // creating an object of the class ArithmeticException to call the
function
    ArithmeticException ae = new ArithmeticException();
    ae.divideInt(1, 0);
  }
}
```

Output :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ArithmeticException.divide(ArithmeticException.java:6)
        at ArithmeticException.main(ArithmeticException.java:16)
```

In this, when the number 1 is divided by 0, an exception is thrown at a run time.

Example of NullPointerException

```
public class Tester {

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

```
    Object ref = null;
    ref.toString(); // this will throw a NullPointerException
  }
}
```

Output :

```
java -cp /tmp/n0Xl19gobM Tester
Exception in thread "main" java.lang.NullPointerException
at Tester.main(Tester.java:4)
```

In this, the null pointer exception is thrown at a run time when the reference of the object is null.

Example of ArrayIndexOutOfBoundsException

```
public class EgAOOB {

  public static void main(String[] args) {
    String[] arr = { "A", "B", "C", "D" };

    for (int i = 0; i <= arr.length; i++) {
      System.out.println(arr[i]);
    }
  }
}
```

Output :

```
A
B
C
D
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4
out of bounds for length 4
        at EgAOOB.main(EgAOOB.java:7)
```

In this, the for loop starts from 0 and ends at the length of the array which is 4 in this case. Because the index pointer 'i' is equal to the length of an array which is 5, it iterates 5 times in an array and prints the exception because several elements in an array are less than the length.

Example of NumberFormatException

```
public class Example {

  public static void main(String[] args) {
    int a = Integer.parseInt(null); //throws Exception as     //the input
string is of illegal format for parsing as it is null.
  }
}
```

Output :

```
Exception in thread "main" java.lang.NumberFormatException: Cannot parse null
string
        at java.base/java.lang.Integer.parseInt(Integer.java:630)
        at java.base/java.lang.Integer.parseInt(Integer.java:786)
        at Example.main(Example.java:4)
```

In this, instead of a string, a null value is passed an exception occurs.

Example of InputMismatchException

```
import java.util.Scanner;

public class InputEg {

  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter name: ");
    String name = sc.next();
    System.out.println("Enter age: ");
    int age = sc.nextInt();
    System.out.println(name);
    System.out.println(age);
  }
}
```

Output :

```
java -cp /tmp/V4mOs0Qj2l InputEg
Enter name:
titu
Enter age:
ten
Exception in thread "main" java.util.InputMismatchExceptionat
java.base/java.util.Scanner.throwFor(Scanner.java:939)
at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)at
java.base/java.util.Scanner.nextInt(Scanner.java:2212)
at InputEg.main(InputEg.java:8)
```

In this example, instead of passing an int value for the age, the string value is passed. This caused an exception.

Example of IllegalStateException

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Example {

  public static void main(String args[]) {
    ArrayList<String> list = new ArrayList<String>();

    list.add("apples");
```

```
    list.add("mangoes");

    ListIterator<String> it = list.listIterator();

    it.remove();
  }
}
```

Output :

```
Exception in thread "main" java.lang.IllegalStateException
   at java.util.ArrayList$Itr.remove(Unknown Source)
   at MyPackage.NextElementExample.main(NextElementExample.java:17)
```

Array Basics

An array is a container object that holds a fixed number of values of a single type. We do not need to create different variables to store many values, instead we store them in different indices of the same objects and refer them by these indices whenever we need to call them.
Syntax:

```
Datatype[] arrayName = {val1, val2, val3,……. valN}
```

Copy

Note: array indexing in java starts from [0].

There are two types of array in java:

- Single Dimensional Array
- Multi-Dimensionalal Array

We will see how to perform different operations on both the type of arrays,

A. Length of an array:

Finding out the length of an array is very crucial when performing major operations. This is done using the .length property:

Example:

```
public class ArrayExample {
    public static void main(String[] args) {
        //creating array objects
        String[] cities = {"Delhi", "Mumbai", "Lucknow",
"Pune", "Chennai"};
        int[] numbers = {25,93,48,95,74,63,87,11,36};

        System.out.println("Number of Cities: " +
cities.length);
        System.out.println("Length of Num List: " +
numbers.length);
    }
}
```

Copy

Output:

```
Number of Cities: 5
Length of Num List: 9
```

Copy

B. Accessing array elements:

Array elements can be accessed using indexing. In java, indexing starts from 0 rather than 1.
Example:

```
public class ArrayExample {
    public static void main(String[] args) {
        //creating array objects
```

```
        String[] cities = {"Delhi", "Mumbai", "Lucknow",
"Pune", "Chennai"};
        int[] numbers = {25,93,48,95,74,63,87,11,36};

        //accessing array elements using indexing
        System.out.println(cities[3]);
        System.out.println(numbers[2]);
    }
}
```

Copy

Output:

```
Pune
48
```

Copy

C. Change array elements:

The value of any element within the array can be changed by referring to its index number.
Example:

```
public class ArrayExample {
    public static void main(String[] args) {
        //creating array objects
        String[] cities = {"Delhi", "Mumbai", "Lucknow",
"Pune", "Chennai"};

        cities[2] = "Bangalore";
        System.out.println(cities[2]);
```

```
    }
}
```

Copy

Output:

```
Bangalore
```

Array list

```
import java.util.*;
  public class Exercise7 {
  public static void main(String[] args) {
  // Creae a list and add some colors to the list
  List<String> list_Strings = new ArrayList<String>();
  list_Strings.add("Red");
  list_Strings.add("Green");
  list_Strings.add("Orange");
  list_Strings.add("White");
  list_Strings.add("Black");
  // Search the value Red
    if (list_Strings.contains("Red")) {
    System.out.println("Found the element");
    } else {
    System.out.println("There is no such element");
    }
 }

}
```

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

FileName: ThreadPriorityExample.java

```
1.  // Importing the required classes
2.  import java.lang.*;
3.
4.  public class ThreadPriorityExample extends Thread
5.  {
6.
7.  // Method 1
8.  // Whenever the start() method is called by a thread
9.  // the run() method is invoked
10. public void run()
11. {
12. // the print statement
13. System.out.println("Inside the run() method");
14. }
15.
16. // the main method
17. public static void main(String argvs[])
18. {
19. // Creating threads with the help of ThreadPriorityExample class
20. ThreadPriorityExample th1 = new ThreadPriorityExample();
21. ThreadPriorityExample th2 = new ThreadPriorityExample();
22. ThreadPriorityExample th3 = new ThreadPriorityExample();
23.
24. // We did not mention the priority of the thread.
25. // Therefore, the priorities of the thread is 5, the default value
26.
27. // 1st Thread
28. // Displaying the priority of the thread
29. // using the getPriority() method
30. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
31.
32. // 2nd Thread
```

```
33. // Display the priority of the thread
34. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
35.
36. // 3rd Thread
37. // // Display the priority of the thread
38. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
39.
40. // Setting priorities of above threads by
41. // passing integer arguments
42. th1.setPriority(6);
43. th2.setPriority(3);
44. th3.setPriority(9);
45.
46. // 6
47. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
48.
49. // 3
50. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
51.
52. // 9
53. System.out.println("Priority of the thread th3 is : " + th3.getPriority());
54.
55. // Main thread
56.
57. // Displaying name of the currently executing thread
58. System.out.println("Currently Executing The Thread : " + Thread.currentThread().getNam
    e());
59.
60. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority(
    ));
61.
62. // Priority of the main thread is 10 now
63. Thread.currentThread().setPriority(10);
64.
```

```
65. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority(
    ));
66. }
67. }
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

FileName: ThreadPriorityExample1.java

```
1.  // importing the java.lang package
2.  import java.lang.*;
3.
4.  public class ThreadPriorityExample1 extends Thread
5.  {
6.
7.  // Method 1
8.  // Whenever the start() method is called by a thread
9.  // the run() method is invoked
10. public void run()
11. {
12. // the print statement
```

```
13. System.out.println("Inside the run() method");
14. }
15.
16.
17. // the main method
18. public static void main(String argvs[])
19. {
20.
21. // Now, priority of the main thread is set to 7
22. Thread.currentThread().setPriority(7);
23.
24. // the current thread is retrieved
25. // using the currentThread() method
26.
27. // displaying the main thread priority
28. // using the getPriority() method of the Thread class
29. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority(
    ));
30.
31. // creating a thread by creating an object of the class ThreadPriorityExample1
32. ThreadPriorityExample1 th1 = new ThreadPriorityExample1();
33.
34. // th1 thread is the child of the main thread
35. // therefore, the th1 thread also gets the priority 7
36.
37. // Displaying the priority of the current thread
38. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
39. }
40. }
```

Output:


```
Priority of the main thread is : 7
Priority of the thread th1 is : 7
```

Explanation: If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

Example of IllegalArgumentException

We know that if the value of the parameter *newPriority* of the method getPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException. Let's observe the same with the help of an example.

FileName: IllegalArgumentException.java

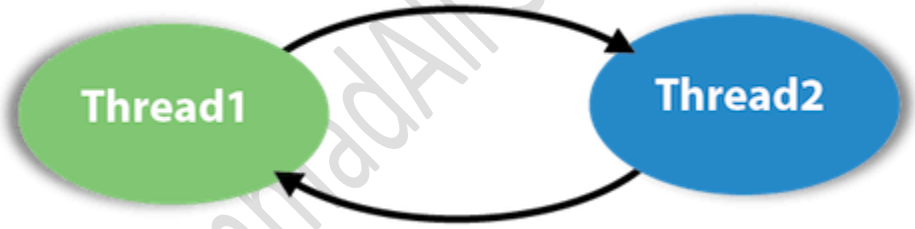
```
1. // importing the java.lang package
2. import java.lang.*;
3.
4. public class IllegalArgumentException extends Thread
5. {
6.
7. // the main method
8. public static void main(String argvs[])
9. {
10.
11. // Now, priority of the main thread is set to 17, which is greater than 10
12. Thread.currentThread().setPriority(17);
13.
14. // The current thread is retrieved
15. // using the currentThread() method
16.
17. // displaying the main thread priority
18. // using the getPriority() method of the Thread class
19. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority(
    ));
20.
21. }
22. }
```

When we execute the above program, we get the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentException
        at java.base/java.lang.Thread.setPriority(Thread.java:1141)
        at IllegalArgumentException.main(IllegalArgumentException.java:12)
```

Deadlock in Java

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in Java

TestDeadlockExample1.java

```
1. public class TestDeadlockExample1 {
2.   public static void main(String[] args) {
3.     final String resource1 = "ratan jaiswal";
4.     final String resource2 = "vimal jaiswal";
5.     // t1 tries to lock resource1 then resource2
6.     Thread t1 = new Thread() {
```

```
7.        public void run() {
8.           synchronized (resource1) {
9.            System.out.println("Thread 1: locked resource 1");
10.
11.           try { Thread.sleep(100);} catch (Exception e) {}
12.
13.           synchronized (resource2) {
14.            System.out.println("Thread 1: locked resource 2");
15.           }
16.         }
17.      }
18.    };
19.
20.    // t2 tries to lock resource2 then resource1
21.    Thread t2 = new Thread() {
22.      public void run() {
23.        synchronized (resource2) {
24.          System.out.println("Thread 2: locked resource 2");
25.
26.          try { Thread.sleep(100);} catch (Exception e) {}
27.
28.          synchronized (resource1) {
29.            System.out.println("Thread 2: locked resource 1");
30.          }
31.        }
32.      }
33.    };
34.
35.
36.    t1.start();
37.    t2.start();
38.  }
39.}
```

Output:

```
Thread 1: locked resource 1
        Thread 2: locked resource 2
```

More Complicated Deadlocks

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

DeadlockSolved.java

```
1.  public class DeadlockSolved {
2.
3.     public static void main(String ar[]) {
4.        DeadlockSolved test = new DeadlockSolved();
5.
6.        final resource1 a = test.new resource1();
7.        final resource2 b = test.new resource2();
8.
9.     // Thread-1
10. Runnable b1 = new Runnable() {
11.    public void run() {
```

```
12.         synchronized (b) {
13.             try {
14.                 /* Adding delay so that both threads can start trying to lock resources */
15.                 Thread.sleep(100);
16.             } catch (InterruptedException e) {
17.                 e.printStackTrace();
18.             }
19.             // Thread-1 have resource1 but need resource2 also
20.             synchronized (a) {
21.                 System.out.println("In block 1");
22.             }
23.         }
24.     }
25. };
26.
27. // Thread-2
28. Runnable b2 = new Runnable() {
29.     public void run() {
30.         synchronized (b) {
31.             // Thread-2 have resource2 but need resource1 also
32.             synchronized (a) {
33.                 System.out.println("In block 2");
34.             }
35.         }
36.     }
37. };
38.
39.
40.         new Thread(b1).start();
41.         new Thread(b2).start();
42.     }
43.
44.     // resource1
45.     private class resource1 {
```

```
46.     private int i = 10;
47.
48.     public int getI() {
49.         return i;
50.     }
51.
52.     public void setI(int i) {
53.         this.i = i;
54.     }
55. }
56.
57. // resource2
58. private class resource2 {
59.     private int i = 20;
60.
61.     public int getI() {
62.         return i;
63.     }
64.
65.     public void setI(int i) {
66.         this.i = i;
67.     }
68. }
69.}
```

Output:

```
In block 1
In block 2
```

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

Java HashMap contains values based on the key.

- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

1. **public class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, S erializable

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Constructors of Java HashMap class

Methods of Java HashMap class

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

```
1. import java.util.*;
2. public class HashMapExample1{
3.  public static void main(String args[]){
4.    HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.    map.put(1,"Mango");  //Put elements in Map
6.    map.put(2,"Apple");
7.    map.put(3,"Banana");
8.    map.put(4,"Grapes");
9.
10.   System.out.println("Iterating Hashmap...");
11.   for(Map.Entry m : map.entrySet()){
12.    System.out.println(m.getKey()+" "+m.getValue());
13.   }
14. }
15. }
```

```
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
```

In this example, we are storing Integer as the key and String as the value, so we are using `HashMap<Integer,String>` as the type. The `put()` method inserts the elements in the map.

To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the *getKey()* and *getValue()* methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry.

No Duplicate Key on HashMap

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

```
1. import java.util.*;
2. public class HashMapExample2{
3.  public static void main(String args[]){
4.    HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.    map.put(1,"Mango");  //Put elements in Map
6.    map.put(2,"Apple");
7.    map.put(3,"Banana");
8.    map.put(1,"Grapes"); //trying duplicate key
9.
10.   System.out.println("Iterating Hashmap...");
11.   for(Map.Entry m : map.entrySet()){
12.    System.out.println(m.getKey()+" "+m.getValue());
13.   }
14. }
15. }
```

```
Iterating Hashmap...
1 Grapes
2 Apple
3 Banana
```

Java HashMap example to add() elements

Here, we see different ways to insert elements.

```
1. import java.util.*;
2. class HashMap1{
3.  public static void main(String args[]){
4.    HashMap<Integer,String> hm=new HashMap<Integer,String>();
```

```
5.      System.out.println("Initial list of elements: "+hm);
6.       hm.put(100,"Amit");
7.       hm.put(101,"Vijay");
8.       hm.put(102,"Rahul");
9.
10.      System.out.println("After invoking put() method ");
11.      for(Map.Entry m:hm.entrySet()){
12.       System.out.println(m.getKey()+" "+m.getValue());
13.      }
14.
15.      hm.putIfAbsent(103, "Gaurav");
16.      System.out.println("After invoking putIfAbsent() method ");
17.      for(Map.Entry m:hm.entrySet()){
18.          System.out.println(m.getKey()+" "+m.getValue());
19.         }
20.      HashMap<Integer,String> map=new HashMap<Integer,String>();
21.      map.put(104,"Ravi");
22.      map.putAll(hm);
23.      System.out.println("After invoking putAll() method ");
24.      for(Map.Entry m:map.entrySet()){
25.          System.out.println(m.getKey()+" "+m.getValue());
26.         }
27.  }
28. }
```

```
Initial list of elements: {}
After invoking put() method
100 Amit
101 Vijay
102 Rahul
After invoking putIfAbsent() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
After invoking putAll() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
104 Ravi
```

Java HashMap example to remove() elements

Here, we see different ways to remove elements.

```
1. import java.util.*;
2. public class HashMap2 {
3.   public static void main(String args[]) {
4.    HashMap<Integer,String> map=new HashMap<Integer,String>();
5.      map.put(100,"Amit");
6.      map.put(101,"Vijay");
7.      map.put(102,"Rahul");
8.      map.put(103, "Gaurav");
9.    System.out.println("Initial list of elements: "+map);
10.   //key-based removal
11.   map.remove(100);
12.   System.out.println("Updated list of elements: "+map);
13.   //value-based removal
14.   map.remove(101);
15.   System.out.println("Updated list of elements: "+map);
16.   //key-value pair based removal
17.   map.remove(102, "Rahul");
18.   System.out.println("Updated list of elements: "+map);
19.  }
20. }
```

Output:

```
Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}
Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}
Updated list of elements: {102=Rahul, 103=Gaurav}
Updated list of elements: {103=Gaurav}
```

Java HashMap example to replace() elements

Here, we see different ways to replace elements.

```
1. import java.util.*;
2. class HashMap3{
3.  public static void main(String args[]){
4.    HashMap<Integer,String> hm=new HashMap<Integer,String>();
5.       hm.put(100,"Amit");
6.       hm.put(101,"Vijay");
7.       hm.put(102,"Rahul");
8.       System.out.println("Initial list of elements:");
9.      for(Map.Entry m:hm.entrySet())
10.     {
11.       System.out.println(m.getKey()+" "+m.getValue());
12.     }
13.     System.out.println("Updated list of elements:");
14.     hm.replace(102, "Gaurav");
15.     for(Map.Entry m:hm.entrySet())
16.     {
17.       System.out.println(m.getKey()+" "+m.getValue());
18.     }
19.     System.out.println("Updated list of elements:");
20.     hm.replace(101, "Vijay", "Ravi");
21.     for(Map.Entry m:hm.entrySet())
22.     {
23.       System.out.println(m.getKey()+" "+m.getValue());
24.     }
25.     System.out.println("Updated list of elements:");
26.     hm.replaceAll((k,v) -> "Ajay");
27.     for(Map.Entry m:hm.entrySet())
28.     {
29.       System.out.println(m.getKey()+" "+m.getValue());
30.     }
31.  }
32. }
```

```
Initial list of elements:
100 Amit
101 Vijay
```

```
102 Rahul
Updated list of elements:
100 Amit
101 Vijay
102 Gaurav
Updated list of elements:
100 Amit
101 Ravi
102 Gaurav
Updated list of elements:
100 Ajay
101 Ajay
102 Ajay
```

Difference between HashSet and HashMap

HashSet contains only values whereas HashMap contains an entry(key and value).

Java HashMap Example: Book

```
1. import java.util.*;
2. class Book {
3. int id;
4. String name,author,publisher;
5. int quantity;
6. public Book(int id, String name, String author, String publisher, int quantity) {
7.     this.id = id;
8.     this.name = name;
9.     this.author = author;
10.    this.publisher = publisher;
11.    this.quantity = quantity;
12. }
13. }
14. public class MapExample {
15. public static void main(String[] args) {
16.    //Creating map of Books
17.    Map<Integer,Book> map=new HashMap<Integer,Book>();
18.    //Creating Books
```

```
19.    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

21.    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.    //Adding Books to map
23.    map.put(1,b1);
24.    map.put(2,b2);
25.    map.put(3,b3);
26.
27.    //Traversing map
28.    for(Map.Entry<Integer, Book> entry:map.entrySet()){
29.       int key=entry.getKey();
30.       Book b=entry.getValue();
31.       System.out.println(key+" Details:");
32.       System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33.    }
34. }
35. }
```

Output:

```
1 Details:
101 Let us C Yashwant Kanetkar BPB 8
2 Details:
102 Data Communications and Networking Forouzan Mc Graw Hill 4
3 Details:
103 Operating System Galvin Wiley 6
```

TreeMap and TreeSet in Java

TreeSet is mainly an implementation of SortedSet in java where duplication is not allowed and objects are stored in sorted and ascending order.

TreeMap is an implementation of Map Interface . TreeMap is also Implementation of NavigableMap along with AbstractMap class.

Similarities between TreeSet and TreeMap in java.

- Both TreeMap and TreeSet belong to java.util package.
- Both are the part of the Java Collection Framework.
- They do not allow null values.
- Both are sorted. Sorted order can be natural sorted order defined by Comparable interface or custom sorted order defined by Comparator interface.
- They are not synchronized by which they are not used in concurrent applications.
- Both Provide O(log(n)) time complexity for any operation like put, get, containsKey, remove.
- Both TreeSet and TreeMap Internally uses Red-Black Tree.

TreeSet and TreeMap are both classes in Java that implement the Set and Map interfaces, respectively. They are both sorted collections, which means the elements in the collections are stored in a sorted order.

The main difference between the two is that a TreeSet is a set of unique elements, while a TreeMap is a map that associates a key with a value.

TreeSet:

- Stores unique elements
- Elements are stored in a sorted order based on their natural ordering or based on a custom Comparator provided at the time of creation
- Does not allow null elements
- Implements the Set interface

TreeMap:

- Stores key-value pairs, where each key is unique
- Elements are stored in a sorted order based on the keys' natural ordering or based on a custom Comparator provided at the time of creation
- Allows null values, but not null keys
- Implements the Map interface

In summary, if you want to store a set of unique elements in a sorted order, use a TreeSet. If you want to store key-value pairs in a sorted order, use a TreeMap.

TreeSet Example

```
import java.util.TreeSet;

public class TreeSetExample {
  public static void main(String[] args) {
    TreeSet<String> treeSet = new TreeSet<>();

    // Add elements to the TreeSet
    treeSet.add("apple");
    treeSet.add("banana");
    treeSet.add("orange");

    // Print the TreeSet
    System.out.println(treeSet);

    // Remove an element from the TreeSet
    treeSet.remove("banana");

    // Print the TreeSet again
    System.out.println(treeSet);
  }
}
```

Output below, TreeSet automatically sorts its elements in natural order, so the output may vary depending on the order in which the elements are added.

```
[apple, banana, orange] [apple, orange]
```

TreeMap Example

```
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        // create a new TreeMap to store integers as keys and strings as values
        TreeMap<Integer, String> treeMap = new TreeMap<>();

        // add some key-value pairs to the TreeMap
        treeMap.put(3, "three");
        treeMap.put(1, "one");
        treeMap.put(2, "two");
```

```
        // print out the TreeMap
        System.out.println(treeMap);

        // get the value associated with a key
        String value = treeMap.get(1);
        System.out.println("The value associated with key 1 is " + value);

        // remove a key-value pair
        treeMap.remove(2);

        // print out the TreeMap again
        System.out.println(treeMap);
    }
}
```

List Interface in Java with Examples

Java List

List in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

List Interface declaration

1. **public interface** List<E> **extends** Collection<E>

```
/Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();
```

```
//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();

//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();

//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

Java List Example

```
import java.util.*;
public class ListExample1{
public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

}
}
```

```
Mango
Apple
Banana
```

```
Grapes
```

Java KeyListener in AWT

The Java KeyListener in the Abstract Window Toolkit (AWT) is a fundamental tool for achieving this. The KeyListener Interface is found in “java.awt.event” package. In this article, we’ll explore what the KeyListener is, and its declaration methods, and supply examples with explanatory comments.

Java KeyListener in AWT

The KeyListener port in Java AWT is quite used to listen for keyboard events, such as key presses and key releases. It allows your program to respond to user input from the keyboard, which is crucial for building interactive applications.

Declaring KeyListener

public interface KeyListener extends EventListener

Methods of Java KeyListener in AWT

The KeyListener port defines three method

Method	Description
keyPressed(KeyEvent e)	Invoked when a key is pressed down.
keyReleased(KeyEvent e)	Called when a key is released.
keyTyped(KeyEvent e)	Fired when a key press/release results in a character.


```
// Java program to demonstrate textfield and
// display typed text using KeyListener
import java.awt.*;
import java.awt.event.*;

public class KeyListenerExample extends Frame implements KeyListener {

    private TextField textField;
    private Label displayLabel;
```

```
// Constructor
public KeyListenerExample() {
        // Set frame properties
        setTitle("Typed Text Display");
        setSize(400, 200);
        setLayout(new FlowLayout());

        // Create and add a TextField for text input
        textField = new TextField(20);
        textField.addKeyListener(this);
        add(textField);
        // Create and add a Label to display typed text
        displayLabel = new Label("Typed Text: ");
        add(displayLabel);
        // Ensure the frame can receive key events
        setFocusable(true);
        setFocusTraversalKeysEnabled(false);
        // Make the frame visible
        setVisible(true);
}
```

```
        // Implement the keyPressed method
        @Override
        public void keyPressed(KeyEvent e) {
                // You can add custom logic here if needed
        }
        // Implement the keyReleased method
        @Override
        public void keyReleased(KeyEvent e) {
                // You can add custom logic here if needed
        }
        // Implement the keyTyped method
        @Override
        public void keyTyped(KeyEvent e) {
                char keyChar = e.getKeyChar();
                displayLabel.setText("Typed Text: " + textField.getText() + keyChar);
        }
        public static void main(String[] args) {
                new KeyListenerExample();
        }
}
```


```
import java.awt.*;
import java.awt.event.*;

class Student extends Frame implements ActionListener
{
        Label lsname, lsrollno, lsclass, lgander, lsbg, lsmob, lsadrs;
        CheckboxGroup gander;
        Checkbox male, female, trainpass;
        Choice csclass;
        TextField tfsname, tfsrollno, tfsmob;
        TextArea tasadrs;
        Button submit;

        TextArea display_details;
```

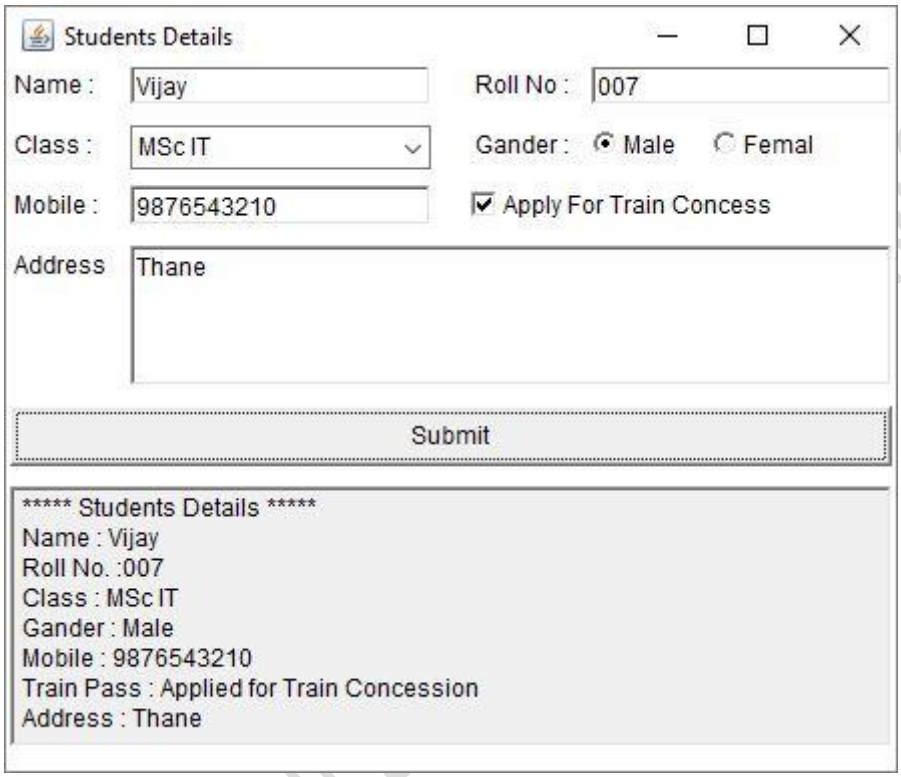
```
        Student()
        {
                lsname   = new Label("Name : ");
                lsrollno = new Label("Roll No : ");
                lsclass  = new Label("Class : ");
                lgander  = new Label("Gander : ");
                lsbg     = new Label("Blood Group : ");
                lsmob    = new Label("Mobile : ");
                lsadrs   = new Label("Address : ");

                gander = new CheckboxGroup();
male   = new Checkbox("Male", gander, false);
female = new Checkbox("Female", gander, false);

trainpass = new Checkbox("Apply For Train Concession");

csclass = new Choice();
csclass.add("BSc IT");
csclass.add("BSc CS");
csclass.add("BCA");
csclass.add("MSc IT");
csclass.add("MSc CS");
csclass.add("MCA");

                tfsname   = new TextField();
                tfsrollno = new TextField();
                tfsmob    = new TextField();

                tasadrs = new TextArea("", 2 , 100 , TextArea.SCROLLBARS_NONE);

                submit  = new Button("Submit");

                display_details = new TextArea("", 2 , 100 , TextArea.SCROLLBARS_NONE);
                display_details.setEditable(false);

                lsname.setBounds(10, 30, 50, 20);
                tfsname.setBounds(70, 30, 150, 20);

                lsrollno.setBounds(240, 30, 50, 20);
```

```
tfsrollno.setBounds(300, 30, 150, 20);

lsclass.setBounds(10, 60, 50, 20);
csclass.setBounds(70, 60, 150, 20);

lgander.setBounds(240, 60, 50, 20);
male.setBounds(300, 60, 50, 20);
female.setBounds(360, 60, 50, 20);

lsmob.setBounds(10, 90, 50, 20);
tfsmob.setBounds(70, 90, 150, 20);

trainpass.setBounds(240, 90, 150, 20);

lsadrs.setBounds(10, 120, 50, 20);
tasadrs.setBounds(70, 120, 380, 70);

submit.setBounds(10, 200, 440, 30);

display_details.setBounds(10, 240, 440, 130);

add(lsname);
add(lsrollno);
add(lsclass);
add(lgander);
add(lsbg);
add(lsadrs);
add(lsmob);

add(male);
add(female);

add(csclass);

add(tfsname);
add(tfsrollno);
add(tasadrs);
add(tfsmob);
```

```
                add(trainpass);

                add(submit);

                add(display_details);

                submit.addActionListener(this);

                setTitle("Students Details");
                setSize(460,390);
                setLayout(null);
                setVisible(true);

                addWindowListener(new WindowAdapter()
                {
    public void windowClosing(WindowEvent e)
    {
       dispose();
    }
});
        }

        public void actionPerformed(ActionEvent e)
        {
                if(e.getSource()==submit)
                {
                        String tp = trainpass.getState() ? "Applied for Train Concession" : "Not Applied for Train Concess

                        String sdetails = " ***** Students Details *****n Name : " + tfsname.getText() + "n Roll No. :" +
Mobile : " + tfsmob.getText() + "n Train Pass : " + tp + "n Address : " + tasadrs.getText();

                        display_details.setText(sdetails);
                }
        }

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

```
}
```

Output



Java Java Program to Retrieve Contents of a Table Using JDBC connection

It can be of two types namely structural and non-structural database. The structural database is the one which can be stored in row and columns. A nonstructural database can not be stored in form of rows and columns for which new concepts are introduced which would not be discussing here. Most of the real-world data is non-structural like photos, videos, social media. As they are not having pre-defined data-types, so they are not present in the database. Hence, the structural database is stored in data warehouses and unstructured in data lakes. Java Database Connectivity is basically a standard API(application interface) between the java programming language and various

databases like Oracle, SQL, PostgreSQL, MongoDB, etc. It connects the front end(for interacting with the users) with the backend for storing data
JDBC consists of 7 elements that are known as connection steps. They are listed below:

JDBC connection steps	Sep/Connection Number
1	Import the package
2	Load and Register the drivers
3	Establish the connection
4	Create the statement
5	Execute the statement
6	Process Result
7	Close/terminate

The syntax for importing package to deal with JDBC operations:

```
import java.sql.* ;
```

The syntax for registering drivers after loading the driver class:

```
forName(com.mysql.jdbc.xyz) ;
```

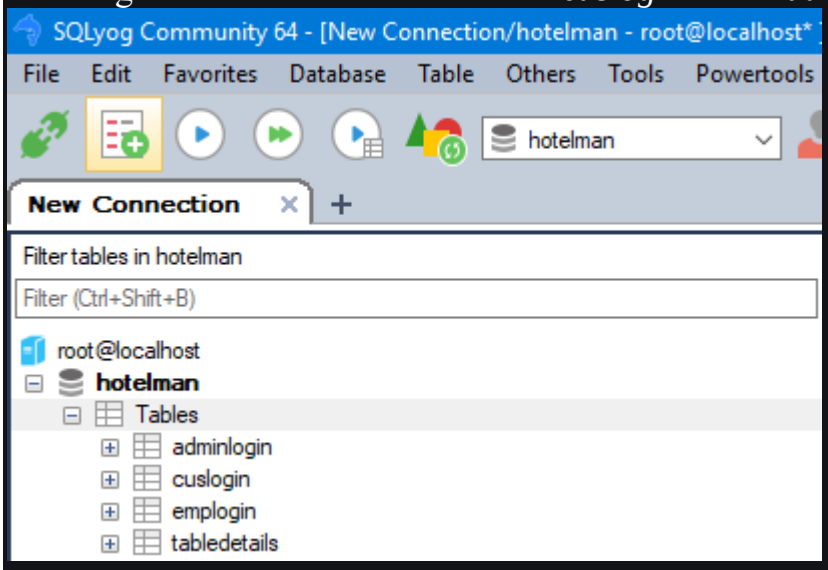
Steps: Below are 3 steps listed covering all 7 components and applying the same:

1. The first database will be created from which the data is supposed to be fetched. If the data is structured SQL can be used to fetch the data. If the data is unstructured MongoDB to fetch data from da lakes. Here SQL database is created and so do all further executions.
2. This step is optional as here need is simply too old and register drivers. In some IDE drivers are inbuilt for example NetBeans but they are missing in eclipse. Here the

installation of drivers is needed. Jar files are needed to be downloaded hardly taking any space in memory.

3. Retrieve Contents of a Table Using JDBC connection

Step 1: Creating a database using SQLyog: The user can create a database using '*SQLyog'* and create some tables in it and fill data inside it. Considering a random example of hotel database management systems. Now applying SQL commands over it naming this database as 'hotelman'. Now suppose the user starts inserting tables inside it be named as "*cuslogin*" and "*adminlogin*".



Step 2: Create Connection: Considering IDE as Netbeans for clear understanding because now work let in this package else if using any other IDEs install drivers first. Once cleared with

1. Create a new package
2. Open a new java file
3. Write the below code for JDBC
4. Save the filename with ***connection.java.***

Here considering IDE as Netbeans for clear' sake of understanding. Below is the illustration for the same:

- Java

```
// Java Program  to Retrieve Contents of a Table Using JDBC

// connection
```

```
// Showing linking of created database


// Importing SQL libraries to create database
import java.sql.*;


public class connection {


    Connection con = null;


    public static Connection connectDB()


    {


        try {
                // Importing and registering drivers
                Class.forName("com.mysql.jdbc.Driver");
```

```
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/hotelman",
                "root", "1234");
            // here,root is the username and 1234 is the
            // password,you can set your own username and
            // password.
            return con;
        }
        catch (SQLException e) {

            System.out.println(e);
        }
    }
}
```

Step 3: Retrieve Contents of a Table Using JDBC connection: Suppose "cuslogin" table has columns namely "id","name","email" and the user wants to see the contents of "cuslogin" table. It involves a series of steps given below with declaration and syntax for interpretation

3.1: Initialize a string with the SQL query as follows

```
String sql="select * from cuslogin";
```

3.2: Initialize the below objects of Connection class, PreparedStatement class, and ResultSet class(needed for JDBC) and connect with the database as follows:

```
Connection con=null;
PreparedStatement p=null;
```

```
ResultSet rs=null;
con=connection.connectDB();
```

3.3: Now, add the SQL query of 3.1 inside prepareStatement and execute it as follows

```
p =con.prepareStatement(sql);
rs =p.executeQuery();
```

3.4 Run a loop till rs.next() is not equal to NULL , fetch values from the table based on the data types, for example we use *getInt()* for integer datatype values and getString() for string datatype values.
3.5 Open a new java file (here, its result.java) inside the same package and type the full code (shown below) for the retrieval of contents of table "cuslogin".
3.6 Both the files viz '*result.java'* and '*connection.java'* should be inside the same package, else the program won't give the desired output!
Implementation: Below is the java example illustrating the same:

- Java

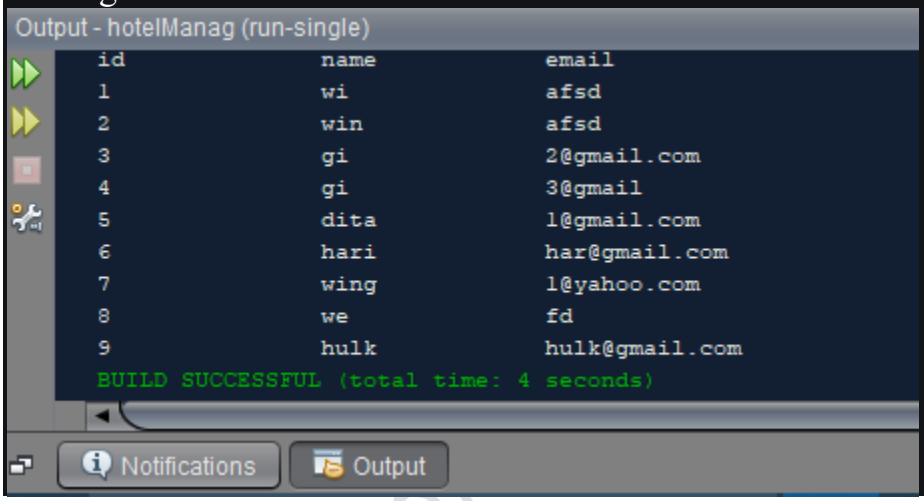
```
// Java Program retrieving contents of
// Table Using JDBC connection


// Java code producing output which is based
// on values stored inside the "cuslogin" table in DB


// Importing SQL libraries to create database
import java.sql.*;


public class GFG {
```

```
// Step1: Main driver method
public static void main(String[] args)
{
    // Step 2: Making connection using
    // Connection type and inbuilt function on
    Connection con = null;
    PreparedStatement p = null;
    ResultSet rs = null;

    con = connection.connectDB();

    // Try block to catch exception/s
    try {

        // SQL command data stored in String datatype
        String sql = "select * from cuslogin";
        p = con.prepareStatement(sql);
        rs = p.executeQuery();
```

```
        // Printing ID, name, email of customers
        // of the SQL command above
        System.out.println("id\t\tname\t\temail");

        // Condition check
        while (rs.next()) {

            int id = rs.getInt("id");
            String name = rs.getString("name");
            String email = rs.getString("email");
            System.out.println(id + "\t\t" + name
                               + "\t\t" + email);
        }
    }

    // Catch block to handle exception
    catch (SQLException e) {
```

```
            // Print exception pop-up on screen

            System.out.println(e);

        }

    }

}
```

Output: The above output is completely based on the values stored inside the "cuslogin" table as shown in the above code



Servlet to Display Factorial of Number

i)HTML form to input a given number

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JSP Page</title>
   </head>
   <body>
      <h1>Factorial of Number</h1>
      <form action="MyServlet" method="Get">
         Enter the Number:<input type="text" name="n1"/><br/>
```

```
        <input type="submit" value="Find"/>
      </form>
   </body>
</html>
```

ii) Servlet Program to accept the given number convert to integer and find the factorial of a number and result is sent back to the Client.

```
package fact;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      String x=request.getParameter("n1");
      int n=Integer.parseInt(x);
      int i=1,fact=1;
      if(n==0)
      {
      out.println("<h1>factorial of 0"+"is"+n+"</h1>");
      }
      else
      {
      while(i<=n)
      {
      fact=fact*i;
      i=i+1;
      }
      out.println("<h1>factorial of " + n +"="+fact+ "</h1>");
      }
   }
}
```