Introduction and Editions of Java {Coding Ninjas Content}

Introduction

Java is an object-oriented, class-based, secured, platform-independent, and general-purpose programming language. Java was originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystem's Java platform. Java programming language is based on the write once, run anywhere (WORA) principle, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of the underlying operating system.

Different Editions of Java

There are four editions of the Java programming language.

- Java Platform, Standard Edition (Java SE)
- Java Platform, Enterprise Edition (Java EE)
- Java Platform, Micro Edition (Java ME)
- JavaFX

1. **Java SE:** Java SE, also known as Core Java, is the most basic and standard version of java. It consists of a wide variety of general-purpose APIs (like java.lang, java.util) as well as many special-purpose APIs. Java SE is used to create Desktop applications. It defines everything from the basic types and objects of java programming language to high-level classes that are used for networking, security, database access, graphical user interface development (GUI), and XML parsing.

2. **Java EE:** The Java EE platform is built on top of the Java SE platform. The Enterprise Edition version of java has a much larger usage of Java, like the development of web services, networking, server-side scripting, and other various web-based applications. Java EE uses HTML, CSS, JavaScript, etc., so as to create web pages and web services. It is also one of the most widely used web development standards.

3. **Java ME:** The Java ME platform is widely used for developing embedded systems, mobiles, and small devices. Java ME uses many libraries and APIs of Java SE, as well as many of its own. The basic aim of this edition is to work on mobiles, wireless devices, set up boxes, etc. Most of the apps developed for the phones were built on Java ME only.

4. **JavaFX:** JavaFX is another edition of java technology, which is now merged with Java SE 8. It is mainly used to create rich GUI (Graphical User Interface) in java apps. It is supported by both desktop environments as well as web browsers.

Previous
Next
Features and Uses of Java

Features and Uses of Java

Features of Java

The features of the Java programming language are given below.

1. **Simple:** Java programming language is easy to learn and its syntax is simple and easy to understand. According to Sun Microsystem, the java programming language is simple because the java syntax is similar to C++. So, it is easier for the programmer to learn java after C++. Java has removed many rarely-used features like explicit pointers, operator overloading, etc. In java, there is no need to remove unused objects because there is an automatic garbage collection in java.

2. **Object-Oriented:** Java is an object-oriented programming language. Everything in java is an object. We can use the functionality of our program by using objects. Object-oriented programming language simplifies software development and maintenance by providing some rules. The basic concepts of OOPs are classes and objects, packages, polymorphism, inheritance, abstraction, encapsulation, and etc.

3. **Platform independent:** Java is a platform-independent programming language. We can write java code once and run it anywhere. Java code can be run on multiple platforms, for example, Windows, Linux, Mac, etc. Java code is compiled by the compiler and converted into byte code. This byte code is platform-independent code because it can be run on multiple platforms.

4. **Robust:** Java programming language is robust because it uses strong memory management. There is an automatic garbage collection in java to destroy the objects which are unused. Exception handling is another feature that makes java programming robust.

5. **Portable:** Java programming language is portable because it enables you to carry byte code to any platform.

6. **Multithreaded:** Java also supports multithreaded programming. We can create multiple threads in java. The main advantage of multithreading is that it doesn't occupy the memory for each thread. It shares a common memory area. Threads are important for web applications, multimedia applications, etc.

7. **Distributed:** Java programming language is distributed because it facilitates the user to create distributed applications. We can create distributed applications in java by using RMI and EJB.

8. **Secure:** When a java program is compiled, it generates a byte code that is in the non-readable form. The java byte code cannot be read by humans. That makes the java language secure.

Uses of Java

Java is the most popular, widely used object-oriented programming language. By using java, we can develop a variety of applications such as web applications, mobile applications, desktop applications, network applications, and many more. Java is used for developing different applications. Some of them are listed below:

1. **Banking:** In banking, the Java programming language is used to develop transaction management.

2. **Mobile App Development:** Java can be used to develop mobile applications. Most of the android applications are built using java. So, if you are familiar with java, it will become much easier to develop mobile applications.

3. **Desktop Applications:** We can also create GUI applications in java. Java provides AWT, Swings, and JavaFX to develop desktop applications.

4. **Big Data:** In Big Data, the Hadoop MapReduce framework is written using java.

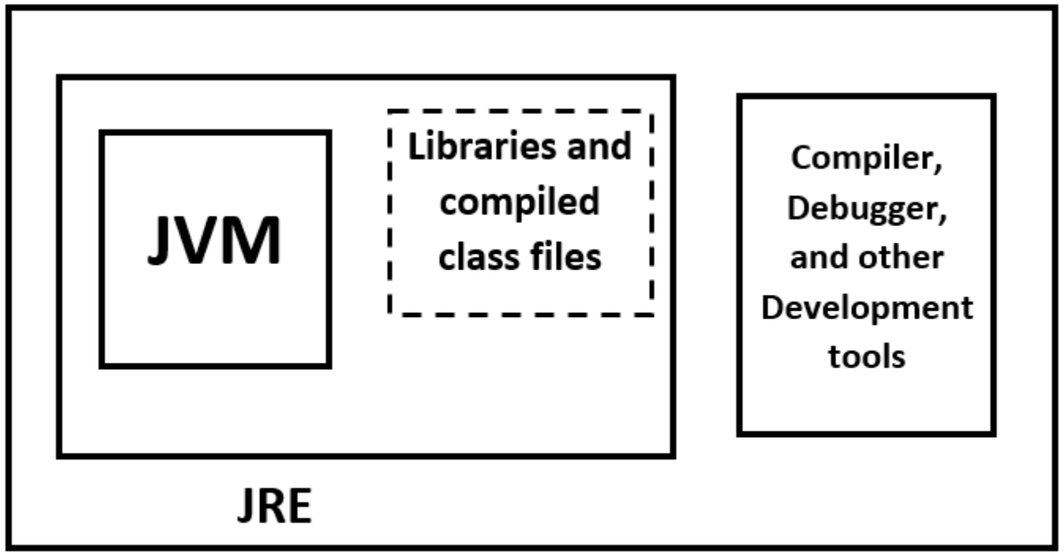
5. **Web applications:** We can also create web applications using java. The most popular frameworks like Spring, Spring Boot, Hibernate used for developing web applications are based on java.

Previous

Next
JDK, JRE and JVM

JDK, JRE, and JVM

JDK, JRE, and JVM are the most important parts of the Java programming language. Without these you can not develop and run java programs on your machine.



1. **JDK:** JDK stands for Java Development Kit. JDK provides an environment to develop and execute the java program. JDK is a kit that includes two things - Development Tools to provide an environment to develop your java programs and JRE to execute your Java programs.
2. **JRE:** JRE stands for Java Runtime Environment. JRE provides an environment to only run (not develop) the java programs onto your machine. JRE is only used by the end-users of the system. JRE consists of libraries and other files that JVM uses at runtime.
3. **JVM:** JVM stands for Java Virtual Machine, is a very important part of both JDK and JRE because it is inbuilt in both. Whatever java program you run using JDK and JRE goes into the JVM and JVM is responsible for executing the java program line by line.

Previous
Next
main() Method

main() Method

Before explaining the java main() method, let's first create a simple program to print Hello World. After that, we will explain why the main() method in java is public static void main(String args[]).

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

- **public:** the public is an access modifier that can be used to specify who can access this main() method. It simply defines the visibility of the method. The JVM calls the main() method outside the class. Therefore it is necessary to make the java main() method as public.
- **static:** static is a keyword in java. We can make static variables and static methods in java with the help of the static keyword. The main advantage of a static method is that we can call this method without creating an instance of the class. JVM calls the main() method without creating an instance of the class, therefore it is necessary to make the java main() method static.
- **void:** void is a return type of method. The java main() method doesn't return any value. Therefore, it is necessary to have a void return type.
- **main:** main is the name of the method. It is a method where program execution starts.
- **String args[]:** String in java is a class that is used to work on Strings and args is a reference variable that refers to an array of type String. If you want to pass the argument through the command line then it is necessary to make the argument of the main() method as String args[].

Previous
Next
Compile and Run

Compile and run Java programs

Below are the steps to compile and run your first java program

1. Open a text editor like notepad or notepad++.
2. Write a java program in your text editor and save it with (.java extension), remember that the file name and the class name must be the same. For example, if you are creating a class HelloWorld, then you need to save it with HelloWorld.java.
3. Open your command prompt.
4. Set the directory where your java program is saved.
5. Once you enter the directory where your java program is saved, now it's time to compile your java program, to compile the java program use the command **javac HelloWorld.java** (In this case my java file is HelloWorld.java).
6. Once your Java program compiles fine, now it's time to run your java program. You can run your java program by using **java HelloWorld** (java file_Name) command.

Comments in Java

Comments are statements that are not executed by the compiler. Comments make the program more human-readable by including the details of the code involved. The proper use of comments makes maintenance and debugging of the code easier.

Types of Comments

There are three types of comments in java

- Single line comments
- Multi-line comments
- Documentation comments

- **Single line comment:** The single line comment is used to comment only one line. A beginner-level programmer uses mostly single-line comments for describing the code functionality.

Syntax:

```
// write your comment here
```

Example:

```
// Java program to show single line comment
public class HelloWorld {
    public static void main(String args[]) {

        // Print "Hello World" on console
        System.out.println("Hello World");
    }
}
```

- **Multi-line comments:** The multi-line comments are used to comment multiple lines of code.

Syntax:

```
/*

This
is
multiline
comment

*/
```

Example:

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

        /*

        Here, we have
        declared a
        variable and we are
        printing its value

        */
        int a = 10;
        System.out.println(a);
    }
}
```

- **Documentation Comments:** This type of comment is used generally when we write the code for projects. It helps to generate a documentation page for reference, which can be used for getting information about methods present, their parameters, etc.

Syntax:

```
/**Comment start
*
*tags are used in order to specify a parameter
*or method or heading.
*HTML tags can also be used
*such as <h1>
*
```

```
*comment ends*/
```

Example:

```
// Java program to illustrate documentation comments
/**
 * Find product of four numbers! The FindPro program implements an application
 * that simply calculates product of four integers and Prints the output on the
 * screen.
 *
 * @author Prashant Srivastava
 * @version 1.0
 * @since 2021-02-26
 */
public class FindPro {
    /**
     * This method is used to find the product of four integers.
     *
     * @param num1 This is the first parameter to FindPro method
     * @param num2 This is the second parameter to FindPro method
     * @param num3 This is the third parameter to FindPro method
     * @param num4 This is the fourth parameter to FindPro method
     * @return int This returns average of numA, numB and numC.
     */
    public int FindPro(int num1, int num2, int num3, int num4) {
        return (num1 * num2 * num3 * num4);
    }

    /**
     * This is the main method which makes use of the FindPro method.
     *
     * @param args Unused.
     * @return Nothing.
     */

    public static void main(String args[]) {
        FindPro obj = new FindPro();
        int pro = obj.FindPro(10, 20, 30, 40);

        System.out.println("Product of 10, 20, 30 and 40 is : " + pro);
    }
```

```
}
```

Output:

```
Product of 10, 20, 30 and 40 is : 240000
```

Previous

Next

Introduction and Declaration of Variables

Introduction and Declaration of Variables

Introduction to Variables

A **Variable** is a name given to a memory location. It is used to store a value that may vary. Java is a statically typed language, and hence, all the variables are declared before use.

Variable Declaration

In Java, we can declare variables as follows:

- **type:** Type of the data that can be stored in this variable. It can be int, float, double, etc.
- **name:** Name given to the variable.

```
data_type variable_name;
Example: int x;
```

In this way, we can only create a variable in the memory location. Currently, it doesn't have any value. We can assign the value in this variable by using two ways:

- By using variable initialization.
- By taking input

Here, we have discussed only the first way, i.e. variable initialization. We will discuss the second way later.

```
data_type variable_name = value;
Example: int x = 10;
```

Previous
Next
Variables Naming Convention

Variables naming Convention in Java

- A variable name should be short and meaningful.
- It should begin with a lowercase letter.
- It can begin with special characters such as _ (underscore) and $ (dollar) sign.
- If the variable name contains multiple words, then use the camel case, i.e. variable name should start with a lowercase letter followed by an uppercase letter. For eg: codingNinja, camelCase.
- Always try to avoid single character variable names such as i, j, and k except for the temporary variables.
- A variable name can not contain whitespaces.
- We can’t use keywords(pre-defined literals) as the variable names.

Previous
Next
Java Keywords

Java Keywords

What is a Keyword?

Keywords in Java are also known as reserved words. These are the predefined words/literals. Hence, they can’t be used as a variable name. If we try to use a keyword as a variable name, the result will be a compile-time error. The list of all the Java Keywords is given below.

abstract	final	protected
assert	finally	public
boolean	for	return
break	float	short
byte	if	static
catch	implements	super
char	import	switch
case	instanceOf	synchronized
class	int	this
continue	interface	throw
default	long	throws
do	native	transient
double	new	try
else	null	void
enum	package	volatile
extends	private	while

Examples:

```
// Correct
int x = 10;
int _x = 10;
int $x = 10;
Int x1 = 10;
// Incorrect
int 1x = 10;
int num ber = 10;
```

Previous

Data types in Java

The data type defines the type of value that can be stored in a variable. For example, if a variable has an ***int*** data type, it can only store an integer value. In java, there are two categories of data types.

1. **Primitive Data Type:** A primitive data type is predefined by the language and is named by a keyword or reserved keyword. There are eight types of primitive data types in java such as boolean, char, int, short, byte, long, float, and double.

- **boolean:** boolean data type specifies only one bit of information and it is used to store only two possible values either true or false.
- **byte:** byte data type is 8 bit signed two's complement integer. Its value lies between -128 to 127. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type is most commonly used to save memory in large arrays.
- **short:** short data type is a 16-bit signed two's complement integer. It can hold any number between -32768 to 32767 (inclusive). Like byte data type, it is commonly used to save memory in large arrays.
- **int:** int data type is 32-bit signed two's complement integer. It can hold the number between -2,147,483,648 to 2,147,483,648. The default value of the int data type is 0.
- **long:** long data type is 64-bit two's complement integer. It can hold the number between -2^63 to 2^63-1. The default value of long data type is 0.
- **float: float data type is used to store** floating-point **numbers. The float data type is a single-precision 32-bit IEEE 754** floating-point**. It can hold 6 to 7 decimal digits. It is recommended to use float instead of double if you need to save memory in large arrays of floating-point numbers. The default value of float is 0.0f.**
- **double:** double data type is generally used to store decimal values. The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The default value of double is 0.0d.
- **char:** The char data type is used to store characters. The char data type is a single 16-bit Unicode character.

Data Type	Default Size	Range
byte	1 byte or 8 bits	-128 to 127
short	2 byte or 16 bits	-32768 to 32767
int	4 bytes or 32 bits	-2^31 to 2^31-1
long	8 bytes or 64 bits	-2^63 to 2^63-1
float	4 bytes or 32 bits	Sufficient to store 6 to 7 digits
double	8 bytes or 64 bits	Sufficient to store 15 to 16 digits
boolean	1 bit	Stores true or false
char	2 bytes	Stores a single character.

2. Non-Primitive Data Type: Non-Primitive data type refers to the objects. ArrayList and String are some of the examples of Non-Primitive data type. We will discuss the Non-Primitive data type later.

Example

```
// Primitive Data Types
int price = 5000;                           // Integer Value
float rateOfInterest = 5.99f;               // Floating point number
char ch = 'a';                              // Character


// Non-Primitive Data Types
String str = "Coding Ninjas";               // String
```

Previous
Next
Scope of Variables in Java

Scope of Variables in Java

The variable scope is the part of the program where the variable is accessible. The scope of the variable can be determined at compile time. There are mainly two types of variable scope.

1) Local Variables Scope: A variable that is defined inside a block, method body, or constructor is called a local variable. These variables can’t be accessed outside the method.

Example:

```
public class VariableScope {
        void method() {
                // local variable (Method Level Scope)
                // This method can’t be accessed outside
                // method body.
                int x;
        }
}
```

2) Member/Class Level Variable Scope: A variable that is declared inside the class but outside the method body, block, or constructor is known as member/class level variable. These variables can be directly accessed anywhere in the class.

Example:

```java
class VariableScope {

        // variable defined inside the class
        int x;
}

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

                // Creating VariableScope class object
                VariableScope obj = new VariableScope();

                // Assigning values in the variable
                obj.x = 10;

                // Printing the value
                System.out.println(obj.x);
        }
}
```

Output:

```
10
```

Previous

Next

Types of Variable

Types of Variable

A variable is a name given to memory location. There are three types of variables in java.

- Local Variable.
- Instance Variable.

- Static variable

1. Local Variables: A variable that is defined inside a block, method body, or constructor is called a local variable. These variables are created when the methods are called and they get destroyed when the methods are executed and return to the caller.
The initialization of the local variable is mandatory. If you don't initialize the variable before use, the compiler will give a compile-time error.

Example:

```
public class Addition {


    // Function to add two numbers
    public void add() {


        // Local variables
        int a = 10;
        int b = 20;
        int c = a + b;

        // Printing the sum
        System.out.println(c);
    }


    // Driver Code
    public static void main(String args[]) {
        // Creating an object of Addition class
        Addition obj = new Addition();
        // Function Call
        obj.add();
    }
}
```

Output:

```
30
```

2. Instance Variables: A variable that is declared inside the class but outside the method body, block, or constructor is known as an instance variable. It is a non-static variable. These variables are created when an instance (object) of the class is created and are destroyed when the object is destroyed. Initialization of the instance variable is not mandatory. If you don't initialize the instance variable, its default value is 0. Instance variables can be accessed only by creating the object of the class.

Example:

```
class Student {

        // These are instance variables
        // these are declared inside the
        // class but outside the method body
        String name;
        int rollno;
}


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

        // Creating Student class object
        Student obj = new Student();

       // Assigning values in the variables
        obj.name = "Ram";
        obj.rollno = 10;

        // Printing name and rollno
        System.out.println(obj.name);
        System.out.println(obj.rollno);
        }
}
```

Output:

```
Ram
10
```

3. Static Variables: A variable that is declared as static is known as a static variable. It is also known as a class variable. These variables are created at the beginning of the program execution and destroyed automatically when the program execution ends. We can create only a single copy of a static variable. To access the static variables, we don't need to create the object of the class. We can simply access the static variable as

```
class_Name.variable_Name;
```

Example:

```
class Student {


        // static variables
        public static int rollno;
        public static String name = "Ram";
}


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

                // accessing static variable without creating object
                Student.rollno = 10;
                System.out.println(Student.name + " 's rollno is :" + Student.rollno);
        }
}
```

Output:

```
Ram's rollno is 10
```

Previous
Next
TypeCasting in Java

TypeCasting in Java

TypeCasting in Java is the process of converting one primitive data type into another. TypeCasting can be done automatically and explicitly.
When we assign the value of one data type to another data type, then there is a chance that two data types might not be compatible with each other. If the data types are compatible, then the Java compiler will perform the conversion automatically. This type of conversion is known as Automatic Type Conversion. If the java compiler is unable to perform the conversion automatically, then they need to be cast explicitly.
There are two types of TypeCasting in Java.

- Widening or Automatic Type Conversion.
- Narrowing or Explicit Type Conversion.

1. Widening or Automatic Type Conversion: When we assign a value of a smaller data type into a large data type, this process is known as Widening Type Casting. It is also known as Automatic Type Conversion because the Java compiler will perform the conversion automatically. This can happen only when the two data types are compatible with each other.

```
byte -> short -> int -> long -> float -> double ( Widening or Automatic Type Conversion)
```

For example, In Java, int data types are compatible with each other, but it isn't compatible with char and boolean data types. Also, char and boolean data types are not compatible with each other.

Example:

```
public class WideningConversation {
    public static void main(String args[]) {
        // Automatic Type Conversion.
        int i = 2147483647; // Int max value in java.
        long l = i; // Automatically converted to long, now we can extend l's value.
        l = l + 1;
        double d = l; // Automatically converted to double.
        System.out.println("Int value : " + i);
        System.out.println("Long value : " + l);
        System.out.println("Double value : " + d);
    }
}
```

Output

```
Int value : 2147483647
Long value : 2147483648
Double value : 2.147483648E9
```

2. Narrowing or Explicit Type Conversion: When we assign a value of a large data type into a small data type, the process is known as Narrowing Type Casting. This can't be done automatically. We need to explicitly convert the type. If we don't perform casting, then the java compiler will give a compile-time error.

```
double -> float -> long -> int -> short -> byte   ( Narrowing or Explicit Type Conversion)
```

Example:

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

                // Explicit Type Conversion
                double d = 25.123;
                int i = (int) d;
                byte b = (byte) i;
                System.out.println("Double value : " + d);
                System.out.println("Int value : " + i);
                System.out.println("Byte value : " + b);
        }
}
```

Output

```
Double value : 25.123
Int value : 25
Byte value : 25
```

Previous
Next
Overflow and Underflow in Java

Overflow and Underflow in Java

Overflow in java happens when we assign a value to a variable that is more than its range and Underflow is opposite to overflow. In case of overflow and underflow, the Java compiler doesn't throw any error. It simply changes the value.

For example, in the case of an int variable, its size is 4 bytes or 32 bits. The maximum value of int data type is 2,147,483,647 (Integer.MAX_VALUE) and after incrementing 1 on this value, it will return -2,147,483,648 (Integer.MIN_VALUE). This is known as overflow. The minimum value of int data type is -2,147,483,648 (Integer.MIN_VALUE) and after decrementing 1 on this value, it will return 2,147,483,647 (Integer.MAX_VALUE). This is known as underflow in Java.

Example:

```
public class OverflowExample {

        public static void main(String args[]) {

                // Overflow
                int overFlow = 2147483647;
                System.out.println("Overflow : " + (overFlow + 1));

                // Underflow
                int underFlow = -2147483648;
                System.out.println("Underflow : " + (underFlow - 1));

        }
}
```

Output

```
Overflow : -2147483648
Underflow : 2147483647
```

Previous

Next

Introduction to Basic I/O

Input / Output In Java

Before discussing how to take input in java, let's first understand how to print a statement in java.

1. Using println() method: In java, we usually use println() method to print the text on the console. The text is passed as the parameter to this method in the form of string. This method prints the text on the console, and after printing the text, the cursor remains at the start of the next line at the console. The next printing takes place from the next line.

Example:

```
System.out.println(“Coding”);
System.out.println(“Ninjas”);
```

Output:

```
Coding
Ninjas
```

2. Using print() method: In java, we usually use print() method to print the text on the console. The text is passed as the parameter to this method in the form of string. This method prints the text on the console, and after printing the text, the cursor remains at the end of the text at the console. The next printing takes place from just here.
Example:

```
System.out.print(“Coding”);
System.out.print(“Ninjas”);
```

Output:

```
CodingNinjas
```

3. Using printf() method: The printf() method in java is used to print formatted data on the console. The print() and println() method take single arguments, but printf() method may take multiple arguments.

Example:

```
// this will print upto 2 decimal places
System.out.printf(“"Formatted with precision: PI = %.2f\n", Math.PI”);


// Automatically appends Zeros to the
// rightmost part of the decimal
float f = 5.2f;
System.out.printf(“Formatted to specific width: n = %.4f\n", n”);
```

Output:

```
Formatted with precision: PI = 3.14
Formatted to specific width: n = 5.2000
```

Example:

```
public class Test {

        public static void main(String args[]) {

                int age = 21;
                String firstName = "King";
                String lastName = "Kong";

                System.out.println("My name is " + firstName + " " + lastName);
                System.out.println("My age is " + age);
        }

}
```

Output:

```
My name is King Kong
My age is 21
```

In the above program, we have declared and initialized three variables: one int variable (age) and two String variables (firstName and lastName). Then, we are printing messages along with the values of these variables by using concatenation. We use String concatenation operator + for concatenating strings in java.

Previous

Next

Taking Input

Taking input in java

In Java, there are mainly two ways to get input from the user:

- Using Scanner class
- Using BufferedReader class

1. Using Scanner class: Scanner is a class in java that is used to take input from the user. It is present in the java.util package. Scanner class is one of the most preferable ways to take input from the user. This class is used to read the input of primitive types such as int, double, long, etc., and some non-primitive(wrapper class objects) types such as String, Boolean, etc. You need to import the java.util package before using the Scanner class.

Methods of Scanner class in Java:

Java Scanner class provides various methods to read different primitive data types from the user.

Method	Description
nextInt()	reads an int value from the user.
nextFloat()	reads a float value from the user.
nextDouble()	reads a double value from the user.
nextLong()	reads a long value from the user.
nextShort()	reads a short value from the user.
nextByte()	reads a byte value from the user.
nextBoolean()	reads a boolean value from the user.
nextLine()	reads a line of text from the user.
next()	reads a word from the user.

Example 1: Taking int value from the user

```
import java.util.Scanner;
public class TakingInputFromUser {

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

```
                // Creating an object of Scanner class
                Scanner sc = new Scanner(System.in);

                // Read integer value from the user
                System.out.println("Enter first number :");
                int a = sc.nextInt();

                System.out.println("Enter second number :");
                int b = sc.nextInt();

                // Adding two values
                int c = a + b;

                // Printing the sum
                System.out.println("Sum is : " +c);

        }
}
```

Output:

```
Enter first number : 10
Enter second number : 20
Sum is : 30
```

Example 2: Taking String from the user

```
import.java.util.Scanner;
public class TakingInputFromUser {
      public static void main(String args[]) {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter a String : ");
                // Read a string from the user
                String str = sc.nextLine();
                System.out.println("Your entered string is : " + str);
    }
}
```

Output:

```
Enter a String : Coding Ninjas
Your entered string is : Coding Ninjas
```

Example 3: Taking char from the user

To read a character from the user, we use next().charAt(0). The next() function of the Scanner class returns the next token/words in the input as a String and charAt(0) function returns the first character in that String.

```
import.java.util.Scanner;
public class TakingInputFromUser {
      public static void main(String args[]) {
              Scanner sc = new Scanner(System.in);
              System.out.println("Enter a character : ");
              // Read a char from the user
              char ch = sc.next().charAt(0);
              System.out.println("Your entered character is : " + ch);
    }
}
```

Output:

```
Enter a character : C
Your entered character is : C
```

Example 4: Taking double from the user

```
import.java.util.Scanner;
public class TakingInputFromUser {
      public static void main(String args[]) {
              Scanner sc = new Scanner(System.in);
              System.out.println("Enter a double value : ");
              // Read a double value from the user
              double d = sc.nextDouble();
              System.out.println("Your entered value is : " + d);
    }
}
```

Output:

```
Enter a double value : 10.0
```

```
Your entered value is : 10.0
```

If you want to take long, short, and float values from the user, then you can use nextLong(), nextShort(), and nextFloat() methods of Scanner class respectively.

2. Using BufferedReader class:

Java BufferedReader class is used to read the stream of characters from the character input stream. The **read()** and **readLine()** method of BufferedReader class is mainly used to read and return the characters.

Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader


public class TakingInputFromUser {
      public static void main(String arg[]) throws IOException {

            // Taking input using BufferedReader
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));


            // Reading input using readLine()
            String name = reader.readLine();


            // Printing the input
            System.out.println(name);
    }
}
```

Input:

```
Coding Ninjas
```

Output:

```
Coding Ninjas
```

Previous

Next

Operators in Java

Operators in Java

Operators in Java are the special symbol specific operations on one, two, or three operands and then return a result. There are different types of operators available in Java which are given below:

- Arithmetic Operators
- Unary Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators
- Instance of Operators

Previous
Next
Arithmetic Operators

Arithmetic Operators

Arithmetic Operators in Java are used to perform mathematical operations, i.e. Addition, Subtraction, Multiplication, and Division etc. The basic arithmetic operators in Java are given below:

- **Addition Operator (+) :** It is used to add two numbers.
- **Subtraction Operator (-) :** It is used to subtract two numbers.
- **Multiplication Operator (*) :** It is used to multiply two numbers.
- **Division Operator (/) :** It is used to divide two numbers.
- **Modulus Operator (%) :** It is used to return the remainder of the division operation.

Example:

In this example, we are going to perform basic mathematical operations on two numbers. Let’s see the example:

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

        // Taking two numbers
        int a = 50;
        int b = 20;

        // Performing addition operation
        System.out.println("Addition of " +a+ " and" +b+ " is : " +(a+b));


        // Performing subtraction operation
        System.out.println("Subtraction of " +a+ " and" +b+ " is : " +(a-b));


        // Performing multiplication operation
        System.out.println("Multiplication of " +a+ " and" +b+ " is : " +(a*b));


        // Performing division operation
        System.out.println("Division of " +a+ " and" +b+ " is : " +(a/b));


        // Performing modulus operation
        System.out.println("Modulus of " +a+ " and" +b+ " is : " +(a%b));
    }
}
```

Output:

```
Addition of 50 and 20 is : 70
Subtraction of 50 and 20 is : 30
Multiplication of 50 and 20 is : 1000
Division of 50 and 20 is : 2
Modulus of 50 and 20 is : 10
```

Previous
Next

Unary Operators

Unary Operators

Unary Operators in Java are the types of operators that require only one operand. They form various operations on single operands such as incrementing or decrementing the value by one, negation of an expression, or inverting the value of a boolean. Let's understand the various unary operators with an example.

(i) Unary minus operator (-): This operator can be used to convert a negative value into a positive value or positive value into a negative value.

Example:

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

       // Convert a positive value
       // into a negative value
       int num1 = 10;
       num1 = -num1;
       System.out.println("Negative Value : " +num1);

       // Convert a negative value
       // into a positive value
       int num2 = -20;
       num2 = -num2;
       System.out.println("Positive Value : " +num2);
   }
}
```

Output:

```
Negative Value : -10
Positive Value : 20
```

(ii) Unary NOT Operator (!): This operator is used to convert the true to false and vice versa.

Example:

```
public class UnaryOperator {
   public static void main(String args[]) {
        int a = 10, b = 20;

         // Without using NOT unary operator.
        System.out.println("(a < b) = " + (a < b));

        // Using unary NOT operator.
        System.out.println("!(a < b) = " + !(a < b));
   }
}
```

Output:

```
(a < b) = true
!(a < b) = false
```

(iii) Increment Operator (++): This operator is used to increment the value by 1. There are two types of increment operator

1. **Post-increment operator: Post increment operator is used to increment the value of the variable after it has been evaluated for use in the expression.**
2. **Pre-increment operator: Pre increment operator is used to increment the value of the variable before it's evaluated in the expression.**

Example:

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


      // Initialize the variable
      int num = 10;


      // first print 10, then number is
      // increment to 11
      System.out.println("Post increment = " + num++);


      // num was 11, incremented to 12 and print
      System.out.println("Pre increment = " + ++num);
   }
}
```

Output:

```
Post increment = 10
Pre increment = 12
```

(iv) Decrement Operator (--): This operator is used to decrement the value by 1. There are two types of decrement operators.

1. **Post-decrement operator: Post decrement operator is used to decrement the value of the variable after it has been evaluated for use in the expression.**
2. **Pre-decrement operator: Pre decrement operator is used to decrement the value of the variable before it's evaluated in the expression.**

Example:

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

      // Initialize the variable
      int num = 10;

      // first print 10, then number is
      // decrement to 9
      System.out.println("Post decrement = " + num--);

      // num was 9, decremented to 8 and print
      System.out.println("Pre increment = " + --num);
   }
}
```

Output:

```
Post decrement = 10
Pre decrement = 8
```

v) Bitwise Complement (~): This operator is used to return the one's complement representations of the input value.

Example:

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

```
        int num1 = 7;
        int num2 = -8;
        // Performing bitwise complement
        System.out.println(num1 +" 's bitwise complement = " + ~num1);
        System.out.println(num2 +" 's bitwise complement = " + ~num2);
    }
}
```

Output:

```
7's bitwise complement = -8
-8's bitwise complement = 7
```

Previous
Next
Assignment Operators

Assignment Operators

The Assignment operators are used to assign a value to the variable. In Java, we can use many assignment operators. These are explained below:

i) +=: This assignment operator is used to add the left operand with the right operand and then assigning it to a variable on the left.

Example:

```
public class AssignmentOperator {
    public static void main(String args[]) {
        int num = 10;
        num += 20;    // num = num + 20;
        System.out.println(num);
    }
}
```

Output:

```
30
```

ii) -=: This assignment operator is used to subtract the left operand with the right operand and then assign it to a variable on the left.

Example:

```
public class AssignmentOperator {
   public static void main(String args[]){
       int num = 20;
       num -= 10;    // num = num - 10;
       System.out.println(num);
   }
}
```

Output:

```
10
```

iii) *=: This assignment operator is used to multiply the left operand with the right operand and then assign it to a variable on the left.

Example:

```
public class AssignmentOperator {
   public static void main(String args[]) {
       int num = 10;
       num *= 5;     // num = num * 5;
       System.out.println(num);
   }
}
```

Output:

```
50
```

iv) /=: This assignment operator is used to divide the left operand with the right operand and then assign it to a variable on the left.

Example:

```
public class AssignmentOperator {
   public static void main(String args[]) {
      int num = 10;
      num /= 2;   // num = num / 2;
      System.out.println(num);
   }
}
```

Output:

```
5
```

v) %=: This assignment operator is used to modulo the left operand with the right operand and then assign it to a variable on the left.

Example:

```
public class AssignmentOperator {
   public static void main(String args[]) {
      int num = 19;
      num %= 2;      // num = num % 2;
      System.out.println(num);
   }
}
```

Output:

```
1
```

Previous

Next

Relational Operators

Relational Operators

The Relational operators are used to check the relationship between two operands. This operator is also called a comparison operator because it is used to make a comparison between two operands. The result of these operators is always boolean value. These operators are used in ***if*** statements and loops. There are many types of relational operators, which are given below:

i) Equal to operator (==): This operator is used to check whether the two operands are equal or not. If they are equal, it returns ***true***; otherwise, it returns ***false.***

Example:

```
public class EqualToOperator {
   public static void main(String args[]) {
      int a = 10;
      int b = 20;
      // Check two operands are equal or not
      // if they equal return true, else return false
      System.out.println(a == b);
   }
}
```

Output:

```
false
```

ii) Not Equal to operator (!=): This operator is used to check whether the two operands are equal or not. It returns true(1) if the left operand is not equal to the right operand; otherwise, it returns false(0).

Example:

```
public class NotEqualToOperator {
   public static void main(String args[]) {
      int a = 10;
      int b = 20;

      // Returns true if left operand
      // is not equal to right operand
      System.out.println(a != b);
   }
}
```

Output:

```
true
```

iii) Greater than operator (>): This operator is used to check whether the first operand is greater than the second operand or not. It returns true(1) if the first operand is greater than the second operand and false(0) if not.

Example:

```
public class GreaterThanOperator {
   public static void main(String args[]) {
      int a = 10;
      int b = 20;

      // Returns true if left operand
      // greater than the right operand
      // else return false
      System.out.println(a > b);
   }
}
```

Output:

```
false
```

iv) Greater than equal to the operator (>=): This operator is used to check whether the first operand is greater than or equal to the second operand or not. It returns true(1) if the first operand is greater than or equal to the second operand; otherwise, it returns false(0).

Example:

```
public class GreaterThanEqualTo {
   public static void main(String args[]) {
      int a = 10;
      int b = 8;

      // It returns true because the
      // first operand is greater than
      // second operand
      System.out.println(a >= b);
   }
}
```

Output:

```
true
```

v) Less than operator (<): This operator is used to check whether the first operand is less than the second operand or not. It returns true(1) if the first operand is less than the second operand else returns false(0).

Example:

```
public class LessThanOperator {
   public static void main(String args[]) {
      int a = 10;
      int b = 15;
      // It returns true because
      // first operand is smaller
      // than the second operand
      System.out.println(a < b);
   }
}
```

Output:

```
true
```

vi) Less than or equal to operator (<=): This operator is used to check whether the first operand is less than or equal to the second operand or not. It returns true(1) if the first operand is less than or equal to the second operand; else, return false(0).

Example:

```
public class LessThanEqualTo {
   public static void main(String args[]) {
      int a = 10;
      int b = 5;

      // Returns false because first operand
      // is not less than or equal to
      // the second operand
      System.out.println(a <= b);
   }
}
```

Output:

```
false
```

Previous

Next

Logical Operators

Logical Operators

These operators are used to perform logical operations such as OR, AND, and NOT operation. It operates on two boolean values, which return true or false as a result. There are three types of Logical Operators in Java:

i) Logical AND operator (&&): This operator returns true(1), if both the conditions are true else returns false(0).

Example:

```
public class Solution {
    public static void main(String args[]) {
        int a = 10;
        int b = 20;
        int c = 30;

        System.out.println((b > a) && (c > b));  // true
        System.out.println((b > a) && (c < b));; // false
    }
}
```

Output:

```
true
false
```

ii) Logical OR operator (||): This operator returns true(1) if any one of the conditions is true.

Example:

```
public class LogicalOrOperator {
   public static void main(String args[]) {
      int a = 10;
      int b = 20;
      int c = 30;

      System.out.println((b > a) || (c < b));  // true
      System.out.println((b < a) || (c < b));  // false
   }
}
```

Output:

```
true
false
```

iii) Logical NOT operator (!): This operator is used to reverse the operand's value. If the operand's value is true, it returns false(0), and if the value of the operand is false, it returns true(1).

Example:

```
public class LogicalOrOperator {
   public static void main(String args[]) {
      int a = 10;
      int b = 20;

      System.out.println( a != b );  // true
      System.out.println( a == b );  // false
   }
}
```

Output:

```
true
false
```

Previous

Next

Bitwise Operators

Bitwise Operators

The Bitwise operators are used to perform bit manipulation on numbers. There are various types of Bit operators that are used in Java.

i) Bitwise AND operator (&): It takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. Mind that the commutative property holds true here.
That is,

1 & 1 = 1
1 & 0 = 0

Example:

```
public class BitwiseAndOperator {
   public static void main(String args[]) {
      int a = 6;
      int b = 7;
      // Binary representation of 6 is 0110
      // Binary representation of 7 is 0111
      // Result is                     0110 = 6
      System.out.println("a & b = " + (a & b));
   }
}
```

Output

```
a & b = 6
```

ii) Bitwise OR operator (|): It takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1. Mind that the commutative property holds true here.

That is,

1 | 1 = 1
1 | 0 = 1
0 | 0 = 0

Example:

```
public class BitwiseOrOperator {
  public static void main(String args[]) {
      int a = 6;
      int b = 7;

      // Binary representation of 6 is 0110
      // Binary representation of 7 is 0111
      // Result is                    0111 = 7
      System.out.println("a | b = " + (a | b));
  }
}
```

Output

```
a | b = 7
```

iii) Bitwise NOT operator (~): It takes one number and inverts all bits of it.

That is,

~1 = 0

~0 = 1

Example:

```
public class ComplementOperator {
   public static void main(String args[]) {
       int a = 6;

       // Binary representation of 6 is 00000000000000000000000000000110                                  // 32 bit representation
       // complement is invert the bits 11111111111111111111111111111001, which is binary representation of -7
       // Result is -7
       System.out.println(" ~a = " + ~a);
   }
}
```

Output

```
~a = -7
```

iv) Bitwise XOR operator (^): It takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different. Mind that the commutative property holds true here.

That is,

1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 0 = 0

Example:

```
public class BitwiseExclusiveOr {
  public static void main(String arsg[]) {
      int a = 6;
      int b = 7;

      // Binary representation of 6 is 0110
      // Binary representation of 7 is 0111
      // Result is                     0001 = 1
      System.out.println("a ^ b = " + (a ^ b));
  }
}
```

Output

```
a ^ b = 1
```

v) Left shift operator (<<): It takes two numbers, the left shift operator shifts the bits of the first operand, the second operand decides the number of places to shift.

Example:

```
public class LeftShiftOperator {
  public static void main(String args[]) {
      int a = 8;

      // Binary representation of 8 is 1 0 0 0
      // Left shift means append number of
      // 0's to the right. 1 0 0 0 0 0 = 32
      System.out.println("a << 2 = " + (a << 2));
  }
}
```

Output

```
a << 2 = 32
```

vi) Right shift operator (>>): It takes two numbers; the right shift operator shifts the bits of the first operand, the second operand decides the number of places to shift.

Example:

```
public class SignedRightShift {
  public static void main(String args[]) {
      int a = 8;

      // Binary representation of 8 is 1 0 0 0
      // signed right shift means remove
      // the number of 0's to the right. 1 0  = 2
      System.out.println("a >> 2 = " + (a >> 2));
  }
}
```

Output

```
a >> 2 = 2
```

NOTE: The left shift and right shift operators should not be used for negative numbers. If any of the operands is a negative number, it results in undefined behaviour. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the integer's size, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits.

vii) Unsigned Right shift operator (>>>): This operator is used to shift the bits of the number to the right and fills 0 in the void spaces that are left as a result. The leftmost bit is set to be 0.

Example:

```
public class UnsignedRightShift {
   public static void main(String args[]) {
       int a = 240;


       // Binary representation of 240 is 1 1 1 1 0 0 0 0
       // Unsigned right shift means remove
       // number of bits to the right and
       // append into the left 0 0 1 1 1 1 0 0 = 60.
```

```
        System.out.println("a >>> 2 = " + (a >>> 2));
    }
}
```

Output

```
a >>> 2 = 60
```

Previous

Next

Ternary Operator

Ternary Operator

Java Ternary operator is also known as the conditional operator: This operator is the only conditional operator in java that takes three operands. We can use ternary operator in place of if-else statement or even switch statement. The syntax of the ternary operator is shown below:

```
variable = expression1 ? expression2 : expression3
```

Here, if expression1 is true then expression2 is evaluated, else expression3 is evaluated.

Example:

```
public class TernaryOperator {
    public static void main(String args[]) {
        int a = 50;
        int b = 100;
        int minimum;


        // Printing the value of a and b
        System.out.println("First Number = " + a);
        System.out.println("Second Number = " + b);

        // Find the minimum number
        minimum = (a < b) ? a : b;
```

```
        // Printing the minimum number
        System.out.println("Minimum Number = " + minimum);
    }
}
```

Output:

```
First Number = 50
Second Number = 100
Minimum Number = 50
```

Previous

Next

Ternary Operator

Ternary Operator

Java Ternary operator is also known as the conditional operator: This operator is the only conditional operator in java that takes three operands. We can use ternary operator in place of if-else statement or even switch statement. The syntax of the ternary operator is shown below:

```
variable = expression1 ? expression2 : expression3
```

Here, if expression1 is true then expression2 is evaluated, else expression3 is evaluated.

Example:

```
public class TernaryOperator {
    public static void main(String args[]) {
        int a = 50;
        int b = 100;
        int minimum;


        // Printing the value of a and b
```

```
        System.out.println("First Number = " + a);
        System.out.println("Second Number = " + b);

        // Find the minimum number
        minimum = (a < b) ? a : b;


      // Printing the minimum number
      System.out.println("Minimum Number = " + minimum);
    }
}
```

Output:

```
First Number = 50
Second Number = 100
Minimum Number = 50
```

Previous

Next

Instance of operator

Instance of operator

The ***instanceof*** operator in java is used to compare an object to a specified type. We can use it to check if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface. The ***instanceof*** operator is either returned **true** or **false**.

Example:

```
public class InstanceOfExample {

    public static void main(String args[]) {

        // Creating the objects of parent
        // and the child class
```

```
        ParentClass obj1 = new ParentClass();
        ChildClass obj2 = new ChildClass();

        // Comparing the object to a specified type
        System.out.println("obj1 is instance of ParentClass = " + (obj1 instanceof ParentClass));
        System.out.println("obj1 is instance of ChildClass = " +  (obj1 instanceof ChildClass));
        System.out.println("obj1 is instance of MyInterface = " + (obj1 instanceof MyInterface));
        System.out.println("obj2 is instance of ParentClass = " + (obj2 instanceof ParentClass));
        System.out.println("obj2 is instance of ChildClass = " +  (obj2 instanceof ChildClass));
        System.out.println("obj2 is instance of MyInterface = " + (obj2 instanceof MyInterface));
    }
}

// Creating parent class
class ParentClass {


}

// Creating an interface
interface MyInterface {


}

// Creating child class
class ChildClass extends ParentClass implements MyInterface {


}
```

Output:

```
obj1 is instance of ParentClass = true
obj1 is instance of ChildClass = false
obj1 is instance of MyInterface = false
obj2 is instance of ParentClass = true
obj2 is instance of ChildClass = true
obj2 is instance of MyInterface = true
```

Note: In the case of ***null*** value, it returns false because null is not an instance of anything. Let's look at an example:

Example:

```
public class InstanceOfOperator {

   public static void main(String args[]) {


      // Creating an object of class
      // and assigning it with null.
      InstanceOfOperator obj = null;


      // Returns false
      System.out.println(obj instanceof InstanceOfOperator);
   }
}
```

Output:

```
false
```

Previous

Next

Introduction and Selection Statements

Control Statements in Java

A Control Statement is used to control the flow of the execution of a program. In the Java programming language, we can control the flow of execution of a program based on some conditions. In Java, we can put control statements in the following three categories. These are selection statements, iteration statements, and jump statements.

Selection Statements:

The selection statements in java allow your program to choose a different path of execution based on certain conditions. Java selection statements provide different types of selection statements such as:

i) if statement: The if statement in Java is a decision-making statement that determines whether or not a certain statement or block of statements will be executed. The block of statements is executed if a certain condition evaluates to true; otherwise, it is not.

Syntax:

```
if(condition) {
   // Code to be executed if the
   // Condition is true
}
```

Example:

```
public class JavaIfStatement {
   public static void main(String arg[]) {
      int a = 10;


      // If the condition is true, then only
      // this block of statement is executed.
      if(a < 20) {
         System.out.println("If block is executed");
      }
   }
}
```

Output:

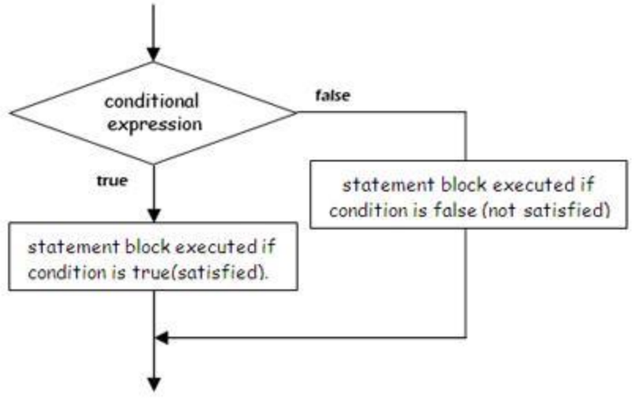
```
If block is executed
```

ii) if-else statement: Java if statement is used to decide whether a certain statement or block of statements will be executed or not. If a certain condition is true, then the block of statements is executed, otherwise not. But if you want to do something else if your condition is false, then you should use else statements. One can use an else statement with if statement to execute the block of statements when the condition is false.

Syntax:

```
if(condition) {
        // execute this block if the
        // condition is true
```

```
}
else {
        // execute this block if the
        // condition is false
}
```



Example:

```
public class JavaIfElseExample {
   public static void main(String args[]) {
      int a = 49;


      // Check the condition,if the condition
      // is true then if block is executed
      // otherwise else block is executed
      if(a % 2 == 0) {
         System.out.println("Even Number");
      }

      else {
           System.out.println("Number is odd");
      }
   }
}
```

Output:

```
Number is odd
```

iii) nested if-statement: nested if-statement means if statement inside the another if statement. In nested-if statements, the inner if block statements execute only if the outer if block statement is true. Let's look at the example to understand better.

Syntax:

```
if(condition1) {
        // execute this block if condition1 is true
        if(condition2) {
                // execute this block if condition2 is true
        }
}
```

Example:

```
public class JavaNestedIfStatement {
   public static void main(String args[]) {
      int a = 20;

      if(a == 20) {


         // First if statement
         if(a < 25)
            System.out.println("a is smaller than 25");


         // Nested if statement, it will execute
         // only when the above if-statement is true.
         if(a < 22)
            System.out.println("a is smaller than 22 too");
         else
```

```
            System.out.println("a is greater than 25");
        }
    }
}
```

Output:

```
a is smaller than 25
a is smaller than 22 too
```

iv) if-else-if ladder: When we need to deal with different conditions in Java, we use the if-else-if ladder. From the top-down, the if sentences are executed. The assumption connected with that it is executed as soon as one of the conditions governing the if is true, and the remainder of the ladder is bypassed. The final else sentence will be executed if none of the conditions are valid. The final else sentence serves as a default condition, meaning that it is executed if all other conditional checks fail. There will be no steps taken if there is no final else and all other circumstances are incorrect.

Syntax:

```
if(condition1) {
    // If condition1 is true, this part of the code will be executed
}
else if (condition2) {
    // If condition2 is true, this part of the code will be executed
}
else if (condition3) {
    // If condition3 is true, this part of the code will be executed
}
...
else {
    // If all the above conditions are false, this part of the code will be executed
}
```

Example:

```
public class JavaIfElseIfLadder {
   public static void main(String arg[]) {
      int a = 50;
```

```
        // if-else-if ladder, checking the
        // multiple conditions
        if(a == 30)
            System.out.println("a is 30");
        else if(a == 45)
             System.out.println("a is 45");
        else if(a == 50)
             System.out.println("a is 50");
        else
            System.out.println("a is not present");
    }
}
```

Output:

```
a is 50
```

v) Switch Statement: A multiway branch statement in Java is the switch statement. It can be used to run a single statement based on a set of conditions. It will deal with data types such as byte, short, char, and int. The value of the expression is compared to one of the literal values in the case statement in the Java switch statement. If there is a match, the code sequence that follows the case statement is executed. The default statement is used if none of the constants fit the value of the expression. It's not necessary to use the default sentence. The declaration sequence is terminated with the split statement within the turn event.

Syntax:

```
switch (expression)  {
    case constant1:
        // Code to be executed
        // If expression is equal to constant1
        break;
    case constant2:
        // Code to be executed
        // If expression is equal to constant2
        break;
        ...
    default:
        // Code to be executed
```

```
        // If the expression doesn't match any constant
}
```

Example:

```
public class JavaSwitchStatement {
        public static void main(String arg[]) {

                int i = 5;

                // switch expression
                switch(i) {

                        // case statements
                        case 0:
                                System.out.println("i is zero");
                                break;

                        case 1:
                                System.out.println("i is one");
                                break;

                        case 2:
                                System.out.println("i is two");
                                break;

                        // Default case statement
                        default:
                                System.out.println("i is greater than two");
                }
        }
}
```

Output:

```
i is greater than two
```

Iteration Statements

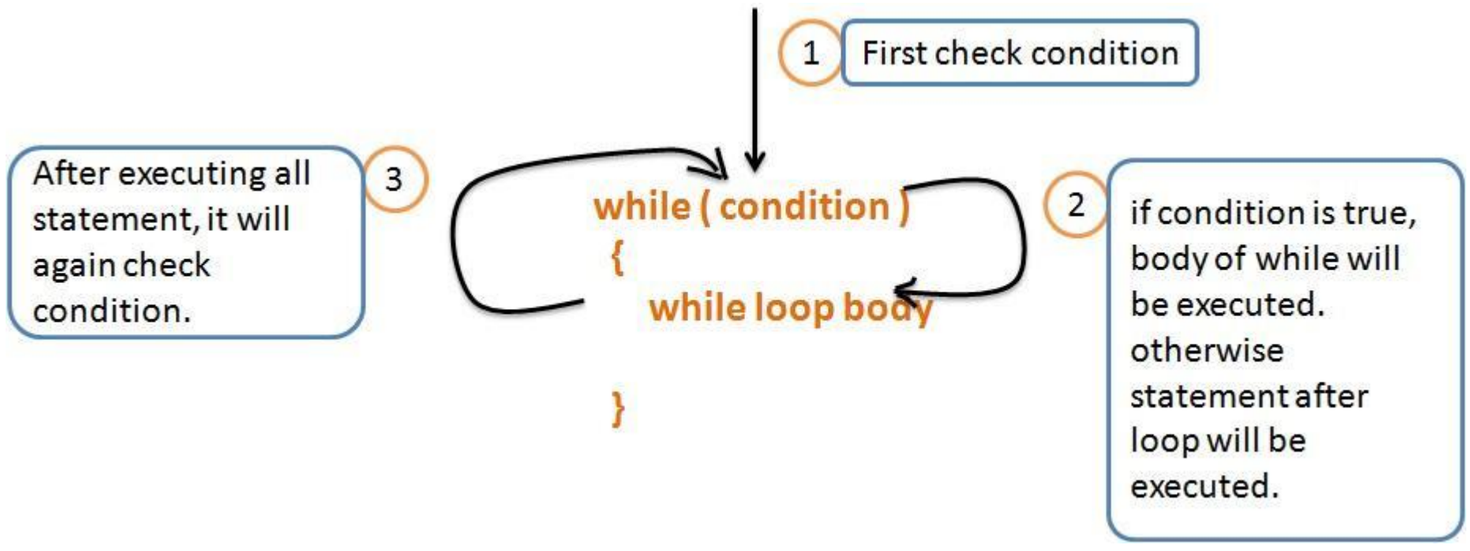
Java iteration statements are used to repeat the set of statements until the condition of the termination is true. There are three types of iteration statements in java.

i) while loop: Java while loop is used to repeat a statement or block of the statement until a condition is true. We can use a while loop if the number of iterations is not fixed. The condition of a while loop is any boolean expression. The loop will run till the condition is true, if the condition becomes false the control goes to the next statement immediately following the loop.

Syntax:

```
initialization;
while (condition) {
    // statements


    update_expression;
}
```

To better understand, look at the diagram given below:



Example:

```
public class JavaWhileLoop {
   public static void main(String arg[]) {

      // Initialization
      int i = 1;


       // While loop run till the
       // condition is true
       while(i < = 10) {
            System.out.print(i + " ");


            // Increment the value by 1.
            i++;
       }
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

ii) do-while loop: Java do-while loop is also used to repeat a statement until a condition is true. Sometimes in our program we want to execute the body of the loop at least once, even if the conditional expression is false, in other words, there are times when you would like to test the conditional expression at the end of the loop rather than the beginning. Then we should go for a do-while loop in java. It executes the body loop at least once because the conditional expression is checking at the end. Let's look at the example to understand this.

Syntax:

```
initialization;
do {
    // statements
    // update_expression;
}
while (condition);
```

Example:

```
public class JavaDoWhileLoop {
   public static void main(String arg[]) {

        // Initialization
        int i = 1;
        do {
            System.out.print(i + " ");
            i++;
         }

         // Checking condition at the end
         while(i <= 10);
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

iii) for loop: The for loop in java is used to iterate a part of the program several times. It consumes the initialization, condition checking, and increment/decrement a value in one line. If the number of iterations is fixed then it is recommended you use Java for loop.

Syntax:

```
for(initialization condition; testing condition; increment/decrement) {
    // statements
}
```

Example:

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

      // for loop to print the value
      // from 1 to 10
      for(int i = 1; i <= 10; i++)
          System.out.print(i + " ");
  }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Java Enhanced for loop: Java Enhanced for loop provides a simpler way to iterate through the elements of a collection or array. It is used when we need to sequentially iterate through elements without knowing the index of the currently processing element.

Syntax:

```
for(T element: collection obj / array) {
    // statements
}
```

Example:

```
public class JavaEnhancedForLoop {
   public static void main(String arg[]) {
```

```
        // array of string
        String array[] = {“Coding”, “Ninjas”, “Welcomes”, “You”};


        // Enhanced for loop
        for(String x : array) {
            System.out.println(x);
        }
    }
}
```

Output:

```
Coding
Ninjas
Welcomes
You
```

Previous

Next

Jump Statements

Jump Statements

In Java, we use jump statements to transfer the control of the program to other parts of the program. There are various types of jump statements in java.

i) break statement: The loops are terminated using the Java break expression. It can be used within a loop. When a break statement is found within a loop, the loop is ended and control is passed to the next statement after the loop. There are three uses for the Java break statement.

- It can be used to terminate a statement sequence in the switch statement.
- It can be used to terminate a loop.
- It can be used as a civilized form of goto statement for performing a labeled break.

Syntax:

```
break;
```

Example:

```
public class JavaBreakStatement {
        public static void main(String args[]) {
        for(int i = 1; i <= 10; i++) {
                // terminate the loop if i is 5.
                if(i == 5)
                break;
                System.out.print(i + " ");
        }
        }
}
```

Output:

```
1 2 3 4
```

ii) labeled break statement: Java labeled break statement can be used as a civilized form of a goto statement. Java doesn't provide a goto statement. It utilizes a label, which is a code block that must enclose the break statement but does not have to do so immediately. To exit a series of nested blocks, we can use labeled break statements.

Example:

```
public class JavaLabeledBreak {
        public static void main(String args[]) {
                boolean b = true;
                // first label
                first: {
                        // second label
                        second: {
                                // third label
                                third: {
                                        System.out.println("Before the break statement");
                                        // break will take the control out
                                        // of the second label
                                        if(b)
```

```
                                        break second;
                                    System.out.println("This would not be execute");
                                }
                                System.out.println("This would not be execute");
                            }
                            System.out.println("This is after the second block");
                        }
                    }
}
```

Output:

```
Before the break statement
This is after the second block
```

iii) continue statement: Java continue statement is used to skip the current iteration of the loop. We can use continue statements in all types of loops. In the case of for loop, the continue statement forces the control to jump immediately to the update statement. In case of while and do-while loop the control immediately jumps to the boolean expression.

Syntax:

```
continue;
```

Example:

```
public class JavaContinueStatement {
    public static void main(String args[]) {
        for(int i = 1; i <= 5; i++) {
            // using continue statement
            // skip the value when i is 3
            if(i == 3)
                continue;

            // Printing the value
            System.out.print(i + " ");
        }
    }
}
```

Output:

```
1 2 4 5
```

iv) Continue Statement with nested loop: It continues the inner loop only when you use the continue statement inside the inner loop.

Example:

```
public class JavaContinueNested {
    public static void main(String args[]) {
        // outer for loop
        for(int i = 1; i <= 3; i++) {
            // inner for loop
            for(int j = 1; j <= 3; j++) {
                // continue statement inside the inner loop
                // to skip the current iteration when
                // i == 2 and j == 2

                if(i == 2 && j == 2) {
                    continue;
                }
                System.out.println(i + " " + j);
            }
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

v) continue statement with labeled for loop: Like the java break statement, the continue statement may also specify a label to describe which enclosing loop to continue. Let's look at the example to understand better.

Example:

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

```
        aa:
        for(int i = 1; i <= 3; i++) {
            bb:
            for(int j = 1; j <= 3; j++) {
                if(i == 2 && j == 2) {
                    // using continue statement with label
                        continue aa;
                }
                System.out.println(i + " " + j);
            }
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

vi) return statement: Java return statement is used to explicitly return from the method. It transfers the program control back to the caller method. The return statement immediately terminates the method in which it is executed. We need to declare a method return type in its method declaration. Any method declared void doesn't return any value. A compile-time error can occur if you attempt to return a value from a method that is declared void.

Syntax:

```
return;
```

Example:

```
public class JavaReturnStatement {
        // function to find sum
        public static int findSum(int num1, int num2) {
        int sum = num1 + num2;
        // return sum
        return sum;
        }
```

```
        public static void main(String args[]) {
        // Creating object of class
        JavaReturnStatement obj = new JavaReturnStatement();
        // Function call
        System.out.println(obj.findSum(10, 20);
        }
}
```

Output:

```
30
```

Previous
Next

Introduction { Methods in Java }

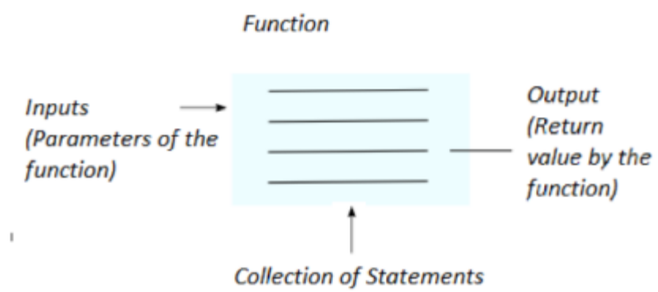
A function is a collection of statements or a block of code that is used to perform some specific task. It is used to reuse the code without retyping the code. We can write the function once and use it many times. We don't need to write the code again and again. The function is executed only when we call it. In Java, a function is also called a method.

Method Declaration in Java: The method declaration in java provides information about the method attributes such as method name, method return type, method visibility, method parameters, and the method body. Let's look at the example of the method declaration.

```
public int max(int x, int y) {
    // Method body
}
```

Here,

- **public:** the public is a modifier.
- **int:** int is a return type of method that means it returns an int value.
- **max:** max is the name of the method.
- **int a, int b:** int a, and int b is the list of parameters



Example of Method Declaration: In this example, we declare a method that accepts two parameters and returns the maximum of two numbers. Let's look at the example:

```
public int max(int x, int y) {

    if(x > y)
        return x;
    else
         return y;
}
```

Method Calling in Java: Once we declare a method, we need to call this method to perform some specific task. When we call a method, the program controls transfer to the called method. Let's look at the example.

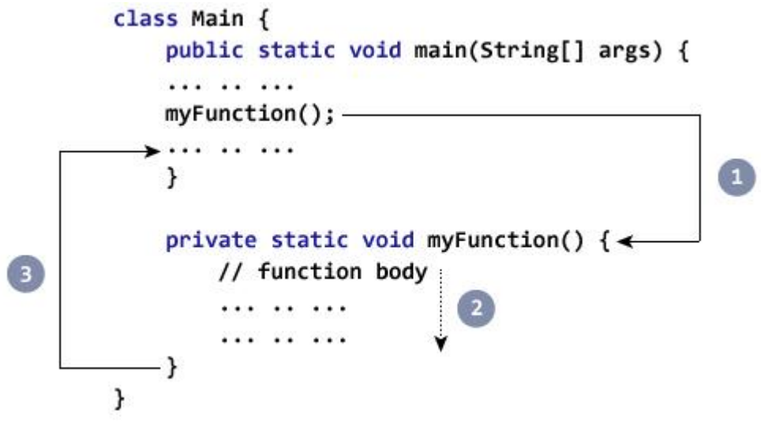
```
public class Example {

    // Function to return a maximum of two numbers
    public static int max(int x, int y) {

         if(x > y)
                return x;
         else
                return y;
    }

    // Driver Method
    public static void main(String arg[]) {
        int a = 10;
        int b = 20;

        // Method Calling
        int maximum = max(a, b);
        System.out.println(maximum);
```

```
    }
}
```

Output:

```
20
```

How does function calling work?



Consider the following code where there is a function called findSum which calculates and returns the sum of two numbers.

```
public class Solution {

    public static int findSum(int a, int b) {
        int sum = a + b;                                    // Line number A
        return sum;
    }

    public static void main(String args[]) {

        int a = 10;
        int b = 20;
        int c = findSum(a, b);          // Line number B
        System.out.println(c);
    }
}
```

The function being called is called **callee**(here it is findSum function) and the function which calls the callee is called the **caller** (here the main function is the caller). When a function is called, program control goes to the entry point of the function. The entry point is where the function is defined. So focus now shifts to the callee and the caller function goes in paused state.

For Example: In the above code entry point of the function **findSum()** is at line number A. So when at line number B the function call occurs the control goes to line number A, then after the statements in the function **findSum()** are executed the program control comes back to line number B.

Why do we need functions?

- **Reusability: A function can be reused once it has been defined. The function can be called as many times as necessary, which saves time. Consider the situation where you must calculate the circle's area. Now you can either use the formula to find the circle area any time or create a function to find the circle area and call it whenever you need it.**

- **Neat code:** A code created with a function is easy to read and dry run. You don't need to repeatedly type the same statements; instead, you can invoke the function whenever needed.

- **Modularisation:** Functions help in modularizing code. Modularisation means dividing the code into small modules, each performing a specific task. Functions allow in doing so as they are the program's tiny fragments designed to perform the specified task.

- **Easy Debugging:** It is easy to find and correct the error in functions compared to raw code without functions where you must correct the error everywhere the specific task of the function is performed.

Previous
Next
Predefined Method

Types of Methods in Java

In Java, there are two types of methods:

- Predefined Method
- User-defined Method

Predefined Method

Predefined methods are the methods that are already defined in the java class libraries. It's also known as the built-in method or the standard library method. At any point in our program, we can use predefined methods explicitly.Some predefined methods in java are: sqrt(), max(), min(), round(), etc. These are defined inside the Math class. Some predefined methods of String class are: length(), toUpperCase(), toLowerCase(), equals(), etc. Let's look at some examples using predefined methods.

Example 1: Find the maximum of two numbers using the built-in method.

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

                // Maximum of two numbers using Math.max()
```

```
        int maximum = Math.max(100, 30);
        System.out.println(maximum);
    }
}
```

Output:

```
100
```

Example 2: Find the minimum of two numbers using the built-in method.

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

        // Minimum of two numbers using Math.min()
        int minimum = Math.min(100, 30);
        System.out.println(minimum);
    }
}
```

Output:

```
30
```

Example 3: Find the square root of a number using the built-in method.

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

        // Finding the square root of a number using Math.sqrt()
        double sqrt = Math.sqrt(144);
        System.out.println(sqrt);
    }
}
```

Output:

```
12.0
```

Example 4: Find string length using the built-in method.

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

                String str = "Coding Ninjas";

                // Printing string length using length()
                System.out.println(str.length());
        }
}
```

Output:

```
13
```

Example 5: Convert the string into the upper case using the built-in method.

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

                String str = "Coding Ninjas";

                // Printing upper case string using toUpperCase()
                System.out.println(str.toUpperCase());
        }
}
```

Output:

```
CODING NINJAS
```

Previous
Next
User-defined Method

User-defined Method

The method written by the user or programmer is called the user-defined method. We can modify these methods based on our requirements. Let’s discuss the user-defined method with all four combinations of arguments and return type.

- **No argument(s) passed and no return value**

When a function has no arguments, it doesn’t receive any data from the calling method. Similarly, when it doesn’t return a value, the calling method doesn’t receive any data from the called method.

Syntax:

```
Function declaration : void function();
Function call : function();
Function definition : void function()
                      {
                          Statements;
                      }
```

Example: In this example, we have created a method checkEvenOdd() and we don't pass any parameters to it. We simply write the code to check whether a number is even or odd as the body of the function. If the number is even, we simply print "Even Number", else we print "Odd Number" and doesn't return any value.

```
public class Solution {

    // Function to check a number
    // is even or odd
    public static void checkEvenOdd() {

        // Number to be checked
        int num = 24;

        if(num % 2 == 0) {
            System.out.println("Even Number");
        }
        else {
            System.out.println("Odd Number");
        }
    }

    public static void main(String args[]) {
        // Method Calling
        checkEvenOdd();
    }

}
```

Output:

```
Even Number
```

- **No arguments passed but return a value**

There could be a requirement in our program where we may need to design a method that takes no argument(s) but returns a value to the calling method.

Syntax:

```
Function declaration : int function();
Function call : function();
Function definition : int function()
                      {
                          Statements;
                          return x;
                      }
```

Example:

```
public class Solution {

        // Function to return the
        // sum of two numbers
        public static int sumOfTwoNumbers() {

                int a = 10;
                int b = 20;
                int sum = a + b;

                return sum;
        }

        public static void main(String args[]) {

                // No arguments passed in the method
                int sum = sumOfTwoNumbers();
                System.out.println(sum);
        }

}
```

Output:

```
30
```

- **Arguments passed but don't return a value**

In this example, we have created a method to check whether a number is even or odd. This method doesn't return any value but when we call this function we need to pass argument(s) to it.

Syntax:

```
Function declaration : int function(int x);
Function call : function(x);
Function definition : int function(x)
                      {
                           Statements;
                      }
```

Example:

```
public class Solution {

        // Method to check a number is even or odd
        public static void findEvenodd(int num) {

                if(num % 2 == 0) {
                        System.out.println("Even Number");
                }

                else {
                        System.out.println("Odd Number");
                }
        }

        // Driver Method
        public static void main(String args[]) {

                int num = 24;

                // argument passed in the method
                findEvenodd(num);
        }
}
```

Output:

```
Even Number
```

- **Arguments passed and do return a value**

In this example, we have created a method that returns the sum of the two numbers and accepts argument(s).

Syntax:

```
Function declaration : int function(int x);
Function call : function(x);
Function definition : int function(x)
                      {
                          Statements;
                          return x;
                      }
```

Example:

```
public class Solution {

        // Function to return the
        // sum of two numbers
        public static int sumOfTwoNumbers(int num1, int num2) {

                int sum = num1 + num2;

                // Return sum
                return sum;
        }

        public static void main(String args[]) {

                int a = 10;
                int b = 20;

                // Method calling with arguments
                System.out.println(sumOfTwoNumbers(a, b));
        }

}
```

Output:

Previous
Next
Passing Parameters to Methods

Passing Parameters to Methods

Pass by value in java:
The formal parameters are allocated to a new memory when the parameters are passed to a function using the pass-by-value method. The value of these parameters is the same as the value of the actual parameters. Changes in formal parameters would not represent changes in individual parameters because they are assigned to the new memory.

Example:

```
public class Solution {

        public static void increase(int x, int y) {
                x++;
                y = y + 2;

                // x and y are formal parameters
                System.out.println(x + ":" + y);
        }

        public static void main(String args[]) {

                int a = 20;
                int b = 10;
                increase(a, b);

                // a and b are actual parameters
                System.out.println(a + ":" + b);

        }
}
```

Output:

```
21:12
```

20:10

Changes in x and y values are not reflected in a and b in the above code because x and y are formal parameters and are local to function increment; thus, any changes in their values here will not impact variables a and b within the key function.

Previous

Next

Method Overloading

Method Overloading

Method overloading in Java is when a class has multiple methods of the same name but different parameters. The main advantage of the method overloading is to increase the readability of the program. Method overloading is related to compile-time polymorphism. You will learn more about this in OOPs as well.

Ways to overload a method: There are two ways to overload a method in java

- By changing the number of arguments.
- By changing the data type.

Method overloading with changing the number of arguments.
In this example, we have created two methods, the first add() method performs the addition of the two numbers, and the second add() method performs the addition of the three numbers. Let's look at the example

```
public class Solution {

        // Function with two parameters
        public static int add(int num1, int num2) {
                return num1 + num2;
        }

        // Function with three parameters
        public static int add(int num1, int num2, int num3) {
                return num1 + num2 + num3;
        }

        public static void main(String args[]) {

                // Method calling by passing arguments
                int sumOfTwoNumbers = Solution.add(10, 20);
```

```
        int sumOfThreeNumbers = Solution.add(10, 20, 30);

        System.out.println(sumOfTwoNumbers);
        System.out.println(sumOfThreeNumbers);
    }

}
```

Output:

```
30
60
```

- **Method overloading with changing the data type of arguments.**

In this example, we have created two add() methods with different data types. The first add() method takes two integer arguments and the second add() takes two double arguments.

```
public class Solution {

    // Function with two integer parameters
    public static int add(int num1, int num2) {
        return num1 + num2;
    }

    // Function with two double parameters
    public static double add(double num1, double num2) {
        return num1 + num2;
    }

    public static void main(String args[]) {

        // Method calling by passing arguments
        int sumOfTwoNumbers = Solution.add(10, 20);
        double sumOfThreeNumbers = Solution.add(10.5, 20.5);

        System.out.println(sumOfTwoNumbers);
        System.out.println(sumOfThreeNumbers);
    }

}
```

Output:

```
30
31.0
```

Previous
Next
Introduction

Introduction {Memory Allocation }

Memory allocation in java specifies the mechanism where the computer programs and services are assigned dedicated virtual memory spaces. The Java Virtual Machine splits the memory into Stack and Heap Memory.

Stack Memory

In Java, stack memory is used for static memory allocation and thread execution.
Methods, local variables, and reference variables are all stored in the Stack portion of memory.

Since they are interpreted in a Last-In-First-Out way, the values in this memory are temporary and restricted to particular methods. When a new method is created, a new block is generated on top of the stack that includes values specific to that method, such as primitive variables and object references.

When a method completes its execution, the resulting stack frame is cleared, the flow returns to the calling method, and space for the next method becomes available.

Characteristics of Stack memory

- It expands and contracts as new methods are called and returned.
- Variables within the stack only last as long as the method's scope remains.
- When the method is executed, it is properly allocated and deallocated.
- Java throws *java.lang.StackOverFlowError* if this memory is full.
- Since each thread runs in its own stack, this memory is thread-safe.
- As compared to heap memory, access to this memory is fast.

Heap Memory

Any time an object is created and allocated in Java Heap Space, it is used. In heap memory, new objects are often formed, and references to these objects are stored in stack memory. Garbage Collection, a discrete function, keeps flushing the memory used by previous objects that have no reference. A Heap Space object can have unrestricted access throughout the program.

These objects are accessible from anywhere in the program and have global access.

This memory model is divided into generations, which are as follows:

- **Young Generation – All new objects are allocated to and aged in this memory. When this is complete, a minor garbage collection occurs.**
- **Old or Tenured Generation – This is the memory where long-lasting items are kept. When objects are stored in the Young Generation, an age threshold is set, and when that threshold is met, the object is transferred to the Old Generation.**
- **Permanent Generation – This is a collection of JVM metadata for runtime classes and application methods.**

Characteristics of Heap memory

- Complex memory storage methods such as Young Generation, Old or Tenured Generation, and Permanent Generation are used to access it.
- Java throws *java.lang.OutOfMemoryError,* if the heap memory is full.
- Access to this memory is slower than access to stack memory.
- Unlike the stack, this memory is not immediately deallocated. Garbage Collector is used to free up unused objects in order to maintain memory efficiency.
- A heap, unlike a stack, is not thread-safe and must be protected by synchronizing the code properly.

Only a reference is created in Java when we only declare a variable of a class type (memory is not allocated for the object). We must use new() to assign memory to an object. As a result, the heap memory is always assigned to the object.

Example:

```
class Test {
    void show() {
        System.out.println("Inside Test::show()");
    }
}

public class Main {

    public static void main(String[] args) {
        // all objects are dynamically allocated
        Test t = new Test();
        t.show(); // No error
    }
}
```

```
Output:
Inside Test::show()
```

```
NOTE: here 't' is a reference variable which refers to the test object which is in the heap memory.
```

Previous
Next
Garbage Collection

Garbage Collection in Java

Garbage collection in java is a process of destroying runtime unused objects. Garbage collectors destroy the objects automatically. A garbage collector's key goal is to allow effective use of memory.

Ways to make an object eligible for the garbage collector

There are primarily three ways to make an object eligible for garbage collection.

- Nullifying the reference variable

```
Student obj = new Student();
obj = null;
```

- By assigning a reference variable to another.

```
Student obj1 = new Student();
Student obj2 = new Student();
obj1 = obj2;
```

- By anonymous object.

```
new Student();
```

Ways for requesting JVM to run garbage collector

There are two ways for requesting a JVM to run a garbage collector.

- Using System.gc() method.
- Using Runtime.getRuntime().gc() method.

Example:

```java
public class Test
{
	public static void main(String[] args) throws ExceptionInterrupted
	{
		Test t1 = new Test();
		Test t2 = new Test();

		// Nullifying the reference variable
		t1 = null;

		// requesting JVM for running Garbage Collector
		System.gc();

		// Nullifying the reference variable
		t2 = null;

		// requesting JVM for running Garbage Collector
		Runtime.getRuntime().gc();
	}

	@Override
	// finalize method is called on object once
	// before garbage collecting it
	protected void finalize() throws Throwable
	{
		System.out.println("Garbage collector called");
		System.out.println("Object garbage collected : " + this);
	}
}
```

Output:

```
Garbage collector called
Object garbage collected : Test@6fe026ef
Garbage collector called
Object garbage collected : Test@6127c563
```

Previous

Next
Introduction and Declaration of Arrays

Introduction to Array

A collection of elements of the same type placed in contiguous memory locations that can be accessed randomly using an array’s indices is called an array. They can store collections of primitive data types such as int, float, double, char, etc., of any particular type.
Instead of declaring different variables for each value, it is used to store several values in a single variable.

Are arrays needed?

We can use variables like temp1, temp2, temp3, etc. when we have a small number of objects. However, if we need to store a large number of instances, managing them with regular variables becomes difficult. An array's purpose is to represent multiple instances in a single variable.
Using arrays saves us from the time and effort required to declare each element of the array individually.

Declaration of Arrays

To use an array in a program, we must declare a variable to refer to the array and specify the form of the array the variable may represent (which cannot be modified once specified).

Syntax:

```
data_type [] arrayRefVar; // preferred way.
OR
data_type arrayRefVar [];
```

Example:

```
int[] arr;
OR
int arr[];
```

Previous

Next

Creating an Array

Creating An Array

Declaring an array variable does not create an array (i.e., no space is reserved for the array).

Syntax:

```
arrayRefVar = new data_type [array Size];
```

Example:

```
arr = new int[20];
```

The above statement does two things:

- It creates an array using the new keyword [array Size].
- It assigns the reference of the newly created array to the variable arrayRefVar.

The declaration of an array variable, the creation of an array, and the assignment of the array's relation to the variable can all be done in one statement, as seen below:

```
data_type[] arrayRefVar = new data_type[array Size];
```

Example:

```
int[] arr = new int[20];
/* This statement will allocate contiguous space for 20 integers*/
```

Previous
Next
Initializing an Array

Initializing an Array

In Single line

The syntax for creating and initializing an array in a single line is as follows:

```
dataType [] arrayRefVar = {value0, value1, ..., value};
```

Example:

```
int [] arr= {1,2,3,4,5,6,7};
```

Using loop

An array can be initialized using a loop.

Example:

```
import java.io.*;
import java.util.*;
public class Example {
    public static void main(String[] args) {
        int[] arr = new int[5];
        Scanner Scan = new Scanner(System.in);
        for (int i = 0; i < arr.length; i++) {
            arr[i] = Scan.nextInt();
        }
    }
}
Input: 2 3 4 5 6
```

Default values for different data types with which an array is initialized

All the arrays' elements after creating arrays are initialized to default values (if we do not initialize them while creating).
Following is a table showing default values for various data_types.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Previous

Next

Accessing elements of an Array

Accessing elements of an array

An element of the array could be accessed using indices, and indexing starts from 0. So if there are 5 elements in the array, then the first index will be 0, and the last one will be 4. Similarly, if we have to store n values in an array, then indexes will range from 0 to n-1.

Note: Trying to retrieve an element from an invalid index will give an *ArrayIndexOutOfBondsException.*
Syntax:

```
Array_name[index];
```

Example:

```
import java.io.*;
```

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        int[] age = {12, 4, 5, 2, 5};
        // access each array elements
        System.out.println("Accessing Elements of Array:");
        System.out.println("First Element: " + age[0]);
        System.out.println("Second Element: " + age[1]);
        System.out.println("Third Element: " + age[2]);
        System.out.println("Fourth Element: " + age[3]);
        System.out.println("Fifth Element: " + age[4]);
    }
}
```

```
Output:
Accessing Elements of Array:
First Element: 12
Second Element: 4
Third Element: 5
Fourth Element: 2
Fifth Element: 5
```

Example:

```java
import java.io.*;
import java.util.*;
public class Example {
    public static void main(String[] args) {
        int[] arr = new int[5];
        Scanner Scan = new Scanner(System.in);
        for (int i = 0; i < arr.length; i++) {
            arr[i] = Scan.nextInt();
        }
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i]);
            System.out.print(" ");
        }
    }
}
```

```
Input: 2 3 4 5 6
Output: 2 3 4 5 6
```

For-Each Loop

This is a special type of loop to access array elements of the array. But this loop can be used only to traverse an array, and nothing can be changed in the array using this loop.

Example:

```
import java.io.*;
import java.util.*;
public class Solutions {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        for (int i: arr) {
            System.out.print(i + " ");
        }
    }
}
Output:
10 20 30 40 50
```

Previous

Next

How are arrays stored

How are arrays stored?

Arrays in Java store either of the two things: either primitive values (int, char,) or references (a.k.a pointers).
When you use the “new” command to build an object, memory is reserved on the heap and a reference is returned. Since arrays are objects, this is also so.

Example:

```
int arr [] = new int [10];
//here arr is a reference to the array and not the name of the array.
```

Reassigning references and Garbage collector

All the reference variables (not final) can be reassigned repeatedly, but their data type to whom they will refer is fixed at the time of their declaration.

Example:

```
import java.io.*;
import java.util.*;
public class Solutions {
    public static void main(String[] args) {
        int[] arr = new int[20];
        // HERE arr is a reference not array name
        int [] arr1=new int [10];
        arr = arr1;
        /* We can re-assign arr to the arrays which referred
          by arr1. Both arr and arr1 refer to the same arrays now.*/
    }
}
```

In the above example, we create two reference variables, arr, and arr1. So now there are also two objects in the garbage collection heap.

Suppose you assign the arr1 reference variable to arr. In that case, no reference will be present for the 20 integer space created earlier, so the Garbage collector can now free this block of memory.

Garbage collector

Live objects(We can think of them as blocks of memory for now) are tracked, and everything else is designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.

- The garbage collector reclaims the underlying memory after an object is no longer in operation and reuses it for future object allocation. This ensures that no memory is returned to the operating system and no explicit delete occurs.

New objects are essentially reserved at the end of the heap that is currently in use.

Previous
Next
Passing Arrays to functions

Passing arrays to a function

Passing Array as a function parameter

In the Java programming language, the parameter passing is always, ALWAYS, made by value. Whenever we create a variable of a type, we pass a copy of its value to the method.

Passing Reference Type // In case of reference of the array is passed.
We store references of Non--Primitive data types and access them via references, So in such cases, the references of Non--Primitives are passed to function.

Example:

```
import java.io.*;
import java.util.*;


public class Solutions {


    public static void print(int[] arr)


    {
        for (int i = 0; i < 5; i++) {


            System.out.print(arr[i] + " ");
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};

        print(arr); // Reference to array is passed
    }
}
```

```
Output:
1 2 3 4 5
```

Similarly, When we pass an array to the increment function, the reference(address) to the array is passed and not the array itself.

Example:

```
import java.io.*;
import java.util.*;


public class Solutions {


    public static void increment(int[] arr)
```

```
    {
        for (int i = 0; i < 5; i++) {
            arr[i]++;


        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        increment(arr);
        for (int i = 0; i < 5; i++)

        {
            System.out.print(arr[i] + " ");
        }

    }
}
```

```
Output:
2 3 4 5 6
```

Here, the reference to the array was passed. Thus inside increment function arr refers to the same array which was created in main. Hence, in the increment function, changes are performed on the same array, and they will reflect in the main.
Now, let’s change the increment function a little and make a pointer pointing to another array, as shown in the example given below.

```
import java.io.*;
import java.util.*;


public class Solutions {


    public static void increment(int[] arr)
```

```
    {
        int[] arr1 = {1, 2, 3, 4, 5};
        arr = arr1;
        for (int i = 0; i < 5; i++) {

            arr[i]++;
        }

    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        increment(arr);
        for (int i = 0; i < 5; i++)

        {
            System.out.print(arr[i] + " ");
        }

    }
}
```

```
Output:
1 2 3 4 5
```

Here, the changes done in the main didn't reflect. Although here the reference to the array was passed but in the first line inside the function, we created another array of size five and made arr refer to that array(without affecting the array created in main). Thus the changes this time won't reflect.

Returning Array from a method

Similarly, as we pass reference as a function parameter we will return reference too in case of array.
Example:

```
import java.io.*;
import java.util.*;
```

```java
public class ArrayUse {

    public static void main(String[] args) {
        int[] A = numbers();
        for (int i = 0; i < 3; i++)
            System.out.print(A[i] + " ");
    }

    public static int[] numbers() {

        int[] A = new int[3];
        A[0] = 2;
        A[1] = 3;
        A[2] = 4;
        return A;
    }
}
```

```
Output:
2 3 4
```

Previous

Next

Introduction to two Dimensional Arrays

Two-Dimensional Array

In Java, we can define multidimensional arrays in simple words as an array of arrays. A two-dimensional array is the simplest kind of multidimensional array. A list of one-dimensional arrays makes up a two-dimensional array.
Syntax to declare a 2-D array of size x,y :

```
int[][] twoD_arr = new int[x][y];
```

Here, x is the row number, and y is the column number.
A two–dimensional array can be seen as a table with x rows and y columns where the row number ranges from 0 to (x-1), and the column number ranges from 0 to (y-1).

Initialization of two-dimensional array

Indirect Method of Initialization:

Syntax:

```
array_name[row_index][column_index] = value;
```

Example:

```
arr[0][0] = 1;
```

Direct Method of Initialization:
Syntax:

```
data_type[][] array_name = {
                           {valueR1C1, valueR1C2, ....},
                           {valueR2C1, valueR2C2, ....}
                         };
```

Example:

```
data_type[][] array_name = {
   {0, 1, 2, 3} ,   /*  initializers for row 0 */
   {4, 5, 6, 7} ,   /*  initializers for row 1 */
   {8, 9, 10, 11}   /*  initializers for row 2 */
};
```

There are three rows and four columns in the following series. The elements of the braces are stored in the table from left to right as well. The array would be filled in the order of the first four elements from the left in the first row, the next four elements in the second row, and so on.

Previous

Next

Accessing 2D-Array Elements

Accessing Two-Dimensional Array Elements

An element in the 2-dimensional array is accessed using the subscripts, i.e., row index and column index of the array.

Syntax:

```
array_name[row_index][column_index];
```

Example:

```
int x[2][1];
```

The element in the third row and second column is shown by the illustration above.

Note: If the array size is N, in arrays. It will have an index ranging from 0 to N-1. As a result, the row number for row index 2 is 2+1 = 3.

We can use nested for loops to output all the elements of a Two-Dimensional sequence. Two for loops will be required: one to traverse the rows and the other to traverse the columns.

Example:

```
import java.io.*;
import java.util.*;


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

                int[][] arr = {{2, 5},
                               {4, 0},
                               {9, 1}};

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 2; j++) {
                System.out.println("arr[" + i + "][" + j + "] = " + arr[i][j] + " ");
            }


        }
    }
}
```

```
Output:
arr[0][0] = 2
arr[0][1] = 5
arr[1][0] = 4
arr[1][1] = 0
arr[2][0] = 9
arr[2][1] = 1
```

Previous

Next

Advantages and Disadvantages of Arrays

Advantages and Disadvantages of Array in Java

Advantages:

- The elements in arrays can be accessed at random using the index number.
- Since it generates a single array of several elements, it requires fewer lines of code.
- All of the array's elements are easily accessible.
- Using a single loop, traversing the array becomes easy.
- Sorting becomes simple because it can be done with fewer lines of code.
- Matrices are represented using two-dimensional arrays.

Disadvantages:

- It only helps one to enter a limited number of elements. We can't change the size of an array until it's been declared. As a result, if we need to enter more numbers documents than are declared, we will be unable to do so. At compile time, we should know the size of the list.
- Since the elements must be handled according to the current memory distribution, insertion and deletion of elements can be expensive.
- The array is homogeneous, which means it can only hold one kind of value.
- When compiling the array, it does not check the indexes. We can get run-time errors instead of finding them at compile time if there are any indexes pointed that are more than the dimension defined.

Example Problems

Question: Add elements of two arrays:

```
import java.io.*;
import java.util.*;

public class Example {
public static void main (String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt(); // size of first array1
    int m = sc.nextInt(); // size of first array2
    int[] array1 = new int[n];
        int[] array2 = new int[m];

        for(int i = 0 ; i < n ; i++){
            array1[i] = sc.nextInt(); // taking input in array1
        }

        for(int i = 0 ; i < m ; i++){
            array2[i] = sc.nextInt(); // taking input in array2
        }

        int result = 0;
        for (int i = 0; i < array1.length; i++) {
            result += array1[i];
        }

        for (int i = 0; i < array2.length; i++) {
            result += array2[i];
```

```
        }

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

```
Input:
4 5
2 3 4 5
10 20 30 40 50
```

```
Output:
164
```

Question: Find if an element is present in the array or not.

```
import java.io.*;
import java.util.*;

public class Example {
public static void main (String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] arr = new int[n];
    for(int i = 0 ; i < n ; i++) {
        arr[i] = sc.nextInt();
    }
          int x = sc.nextInt(); //Element to be searched

          int result = search(arr, arr.length, x);
          if (result == -1)
              System.out.println("Element is not present in array");
          else
              System.out.println("Element is present at index " + result);
```

```
}
static int search(int[] arr, int n, int x) {
          int i;
          for (i = 0; i < n; i++)
              if (arr[i] == x)
                  return i;
          return -1;
      }
}
```

```
Input:
3
1 2 10
10
```

```
Output:
Element is present at index 2
```

Question: Find maximum and minimum in the array.

```
import java.io.*;
import java.util.*;

public class Example {
public static void main (String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] arr = new int[n];
    for(int i = 0 ; i < n ; i++) {
        arr[i] = sc.nextInt();
    }

        int max = arr[0], min = arr[0];
        for (int i = 1; i < arr.length; i++) {
```

```
            if (max < arr[i])
                max = arr[i];
            if (min > arr[i])
                min = arr[i];
        }

        System.out.println("Maximum Value = " + max);
        System.out.println("Minimum Value = " + min);
}
}
```

```
Input:
4
1 13 10 21
```

```
Output:
Maximum Value = 21
Minimum Value = 1
```

Question: Count pairs with given sum in the array.

```
import java.io.*;
import java.util.*;


public class Example {
public static void main (String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] arr = new int[n];
    for(int i = 0 ; i < n ; i++) {
        arr[i] = sc.nextInt();
    }

        int sum = sc.nextInt();
        int count = 0; // Initialize result

        // Consider all possible pairs and check their sums
```

```
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if (arr[i] + arr[j] == sum)
                    count++;
        System.out.println("Count of pairs is " + count);


}
}
```

```
Input:
5
1 1 2 2 3
3


Output:
Count of pairs is 3
```

Previous

Next

Introduction to Strings

Introduction to Strings

Strings in Java

Strings are a sequence of characters, and in the Java programming language, they are considered objects. The String class is provided by the Java platform for creating and manipulating the strings.

We can directly create a string by writing:

```
String str = "Hello world";
```

In the above example, "Hello world!" is a string literal—a sequence of characters enclosed in double quotations. The compiler generates a String object with its value—in this case, Hello world!, whenever it finds a string literal in code.

Note: Strings in Java are immutable that implies we cannot modify their value. Whenever required, StringBuffer and StringBuilder classes can be used to create mutable strings.

In Java, there are two ways to create a string:

1. By assigning a string *literal* directly to a String reference - just like a primitive, or
2. As for every other type, the "new" operator and constructor are used (like arrays and scanners). This, however, is not widely used and is not recommended.

Example:

```
// Implicit construction
String str1 = "Learning Java";

// Explicit construction
String str2 = new String("Java is Interesting!");
```

In the above code, str1 is declared as a String reference and initialized with the string literal "Learning Java" in the first statement. The second statement declares str2 as a String reference and uses the “new” operator to initialize it with "Java is Interesting!"

String literals are stored in a common pool called ***String pool,*** where the strings with the same contents share the same space (resources). This helps to save and utilize the space efficiently.
String objects created using new operators, on the other hand, are stored in the heap memory. No storage sharing is possible in the heap memory, and every object, thus created, has its own allocated space.

Char Arrays and Strings

An array of characters behaves in the same manner as a Java string. Let us see an example to understand this.

Example:

```
public class StringsExample {

    public static void main(String args[]) {
        char[] str = {'C', 'o', 'd', 'i', 'n', 'g', ' ', 'N', 'i', 'n', 'j', 'a', 's'};
        System.out.println(str);
    }

}
```

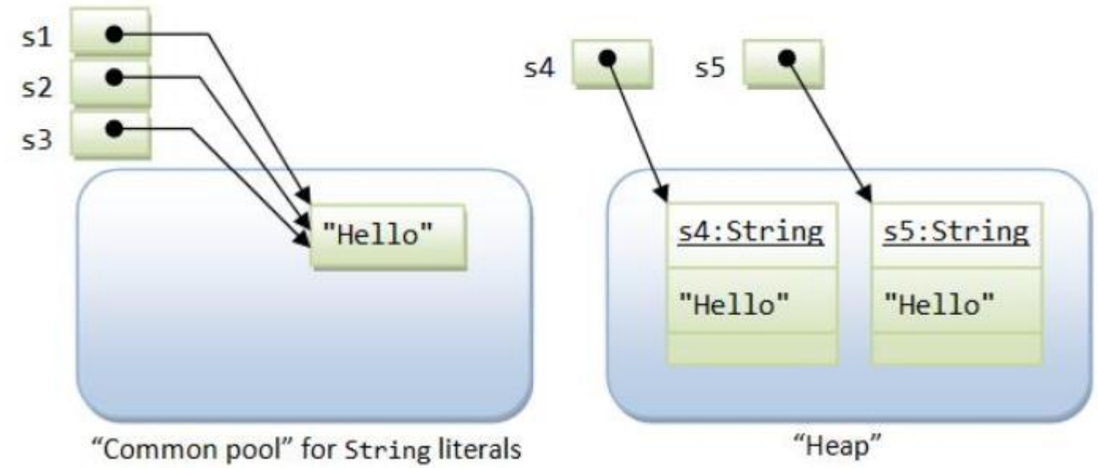
Output:

```
Coding Ninjas
```

String Literal v/s String Object

As previously stated, there are two methods for constructing a string: implicitly, by assigning a string literal or explicitly, by creating a String object using the new operator and constructor.

Java has a specialized feature for storing String literals - in what is known as a string common pool. If the contents of two string literals are the same, they will share the same storage space within the common pool. This strategy is used to save storage space for commonly used strings. String objects created with the new operator and constructor, on the other hand, are stored in heap memory. Each String object in a heap, like every other object, has its storage, and even if two String objects have the same contents, their storage spaces are unique and distinct in the heap.



String is Immutable

Since String literals having the same contents share storage in the same pool, Java's String is intended to be immutable. In other words, once a String is created, its contents cannot be changed. Otherwise, the modification would affect all other String references that use the same storage space, which can be uncertain and undesirable. Methods like toLowerCase() and toUpperCase() might appear to change the contents of a String object. But actually, a completely new String object is created and returned to the caller. Once there are no further references to the original String object, it will be deallocated and garbage collected.

Since String is immutable, it is inefficient to use it if we constantly need to change the String because that would create many new Strings occupying new storage areas, resulting in complete wastage of space.

String Concatenation

Concatenation is the joining of two strings to form a single string. There are several methods for concatenating strings, which are as follows:

- ‘+’ operator
- concat() method

i) ‘+’ operator

The ‘+' operator joins the two input strings and returns a new string containing the concatenated string.

Syntax:

```
String newString = string1 + string2;
```

Example:

```
public class ConcatenationExample{

    public static void main(String args[]) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        str1 = str1 + str2;

        System.out.println("Concatenated String: " + str1);
    }

}
```

Output:

```
Concatenated String: Coding Ninjas!
```

ii) concat() method

The String concat() appends the specified string to the end of the current string.

Syntax:

```
string1.concat(string2);
```

Example:

```
public class ConcatenationExample{

    public static void main(String args[]) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        str1 = str1.concat(str2);

        System.out.println("Concatenated String: " + str1);
    }

}
```

Output:

```
Concatenated String: Coding Ninjas!
```

Note: The concat() method is better than the '+' operator since it produces a new object only when the length of the String to be appended is greater than zero, while the '+' operator always creates a new string regardless of the length of the String.

String Comparison

String comparison is performed to determine whether or not the two strings are equal. Strings in Java can be compared using the following methods:

- equals() method
- == method
- compareTo() method

i) equals() method

The String equals() method compares the string's original content. It compares the string values for equality. For this purpose, the String class has two methods:

- public boolean equals(Object anObject): Returns true if and only if the argument is a String object that represents the same sequence of characters as this object.
- public boolean equalsIgnoreCase(String another): Returns true if and only if the argument is a String object that represents the same sequence of characters as this object, ignoring differences in case.

Example:

```
public class ComparisonExample{
```

```
    public static void main(String args[]) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        String str3 = "Coding";
        String str4 = "coding";

        System.out.println(str1.equals(str2));
        System.out.println(str1.equalsIgnoreCase(str3));
        System.out.println(str1.equalsIgnoreCase(str4));
    }

}
```

Output:

```
false
true
true
```

ii) == method

The == operator compares references rather than values.

Example:

```
public class ComparisonExample{

    public static void main(String args[]) {
        String str1 = "Coding Ninjas";
        String str2 = "Coding Ninjas";
        String str3 = new String("Coding Ninjas");
        System.out.println(str1 == str2); //true (as both str1 and str2 refer to same instance)
        System.out.println(str1 == str3); //false(because str3 refers to instance created in heap)
    }

}
```

Output:

```
true
false
```

iii) compareTo() Method

The String compareTo() method compares two strings lexicographically. It returns an integer indicating whether the first String is greater than, equal to, or less than the second String.

If str1 and str2 are two string variables, then

- **Result = 0 for str1 == str2**
- **Result > 0 for str1 > str2**
- **Result < 0 for str1 < str2**

If we don't want the result to be case sensitive, we can use the **compareToIgnoreCase()** method.

Example:

```
public class ComparisonExample{

    public static void main(String args[]) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        String str3 = "Coding";
        String str4 = "coding";

        System.out.println(str1.compareTo(str2));
        System.out.println(str1.compareTo(str3));
        System.out.println(str1.compareToIgnoreCase(str4));
    }

}
```

Output:

```
35
0
0
```

Previous

StringBuffer and StringBuilder

StringBuffer

We have seen that a Java String is immutable. A StringBuffer is similar to a String but with more functionality. A StringBuffer is mutable, and its length and sequence content can be modified at any time by using specific method calls. Multiple threads can use StringBuffers safely. The methods are synchronized as required such that all operations on any specific instance behave as if they occurred in any sequential order consistent with the order of the method calls made by any of the individual threads involved.

StringBuffer Class Constructors

- **StringBuffer()**: Creates an empty string buffer with an initial capacity of 16 characters.

```
StringBuffer sb = new StringBuffer();
```

- **StringBuffer(int capacity)**: Creates an empty string buffer with the specified initial capacity.

```
StringBuffer sb = new StringBuffer(25);
```

- **StringBuffer(CharSequence seq)**: Creates a string buffer with the same characters as the specified CharSequence.

```
CharSequence seq = "Coding Ninjas";
StringBuffer sb = new StringBuffer(seq);
```

- **StringBuffer(String str)**: Creates a string buffer that is initialized by the contents of the specified string.

```
String str = "Coding Ninjas";
StringBuffer sb = new StringBuffer(str);
```

When an operation involving a source sequence occurs (for example, inserting or appending from a source sequence), the StringBuffer does not synchronize on the source. It only synchronizes on the string buffer executing the operation.

StringBuilder

StringBuilder, an equivalent class, built for use by a single thread, has been introduced to JDK 5's StringBuffer class.
The StringBuilder class is consistent with StringBuffer, but no synchronization is guaranteed. This class is intended to be used as a drop-in substitute for StringBuffer in places where a single thread previously used the string buffer.

StringBuilder Class Constructors

- **StringBuilder()**: Creates an empty string builder with an initial capacity of 16 characters.

```
StringBuilder sb = new StringBuilder();
```

- **StringBuilder(int capacity)**: Creates an empty string builder with the specified initial capacity.

```
StringBuilder sb = new StringBuilder(25);
```

- **StringBuilder(CharSequence seq)**: Creates a string builder with the same characters as the specified CharSequence.

```
CharSequence seq = "Coding Ninjas";
StringBuilder sb = new StringBuilder(seq);
```

- **StringBuilder(String str)**: Creates a string builder which is initialized by the contents of the specified string.

```
String str = "Coding Ninjas";
StringBuilder sb = new StringBuilder(str);
```

Every StringBuffer and StringBuilder has a capacity. It is not necessary to assign a new internal buffer array as long as the length of the character sequence in the StringBuffer, or StringBuilder does not surpass the capacity. When the internal buffer overflows, it is automatically increased in size.

It is recommended that the StringBuilder class be used instead of StringBuffer wherever possible because it is faster in most implementations. StringBuilder instances, on the other hand, are not secure to use by multiple threads. If such synchronization is needed, StringBuffer is the best choice.

Previous
Next
Important Java Methods

Important Java Methods

1. String "Length" Method

String class provides length() method to determine the length of the Java String. It returns the number of characters contained in the string object.

Example:

```
public class HelloWorld {

    public static void main(String args[]) {
        String str = "Coding Ninjas";
        System.out.println("Length of String: " + str.length());
    }

}
```

Output:

```
Length of String: 13
```

2. String "indexOf" Method

String class provides indexOf() method to get the index of a character in Java String. It returns the index within this string of the first occurrence of the specified character.

Example:

```
public class HelloWorld {

    public static void main(String args[]) {
        String str = "Coding Ninjas";

        System.out.println("Index of character 's': " + str.indexOf('s'));
        System.out.println("Index of character 'i': " + str.indexOf('i'));      // gives index of first 'i' character
    }
}
```

Output:

```
Index of character 's': 12
Index of character 'i': 3
```

3. String "contains" Method

String class provides the contains() method to check if a string contains another string (sequence of char values). It returns **true** if and only if the string contains the specified sequence of char values.

Example:

```
public class HelloWorld {

    public static void main(String args[]) {
        String str = "Coding Ninjas";
```

```
        System.out.println("Contains sequence ing: " + str.contains("ing"));
        System.out.println("Contains sequence ings: " + str.contains("ings"));
    }

}
```

Output:

```
Contains sequence ing: true
Contains sequence ings: false
```

4. String "endsWith" Method

String class provides endsWith() method to check if the string ends with the specified suffix. It returns true if the string ends with the given suffix; else returns false.

Example:

```
public class HelloWorld {
    public static void main(String args[]) {
        String str = "Coding Ninjas";

        System.out.println("EndsWith character 's': " + str.endsWith("s"));
        System.out.println("EndsWith character 'g': " + str.endsWith("g"));
    }

}
```

Output:

```
EndsWith character 's': true
EndsWith character 'g': false
```

5. String Java "toLowerCase" & Java "toUpperCase"

String class provides toLowerCase() and toUpperCase() methods to convert the string to lowercase and uppercase characters, respectively.

Example:

```
public class HelloWorld {

    public static void main(String args[]) {
        String str = "Coding Ninjas";

        System.out.println("Convert to LowerCase: " + str.toLowerCase());
        System.out.println("Convert to UpperCase: " + str.toUpperCase());
    }

}
```

Output:

```
Convert to LowerCase: coding ninjas
Convert to UpperCase: CODING NINJAS
```

Example Problems:

Question: Given a string check whether it is Palindrome or not.

```
import java.util.*;

class solution {

    public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    String s = sc.next();

    int low = 0;
    int high = s.length() - 1;
    int flag = 0;

    while (low < high) {
        if (s.charAt(low) != s.charAt(high)) {
            flag = 1;
            break;
        }
```

```
            low++;
            high--;
        }

        if (flag == 0)
            System.out.println("String is a palindrome");
        else
            System.out.println("String is not a palindrome");
        }

}
```

```
Input1:
Hello
Output1:
String is not a palindrome


Input2:
madam
Output2:
String is a palindrome
```

Question: Given a string, find the character which occurred maximum in the string.

```
import java.util.*;

public class solution {

    public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    String s = sc.next();

    int max = 0;
    char result = 'a';
    int[] count = new int[26];

    // Traversing through the string and maintaining the count of each character
    for (int i = 0; i < s.length(); i++) {
```

```
        count[s.charAt(i)  - 'a']++;
    }

    for(int i = 0; i < 26 ; i++){
        if(count[i] > max){
            max = count[i];
            result = (char)('a' + i);
        }
    }

    System.out.println(result);
    }

}
```

```
Input1:
hello
Output1:
l
```

```
Input2:
welcome
Output2:
e
```

Previous

Next

Introduction to OOPs

An Introduction to Object Oriented Programming

OOPs in Java

Object-Oriented programming, or OOPs, as the term implies, refers to a programming language that utilizes the concepts of class and object. The main aim of OOPs is to implement real-world entities such as polymorphism, inheritance, encapsulation, abstraction, etc. The popular object-oriented programming languages are c++, java, python, PHP, c#, etc. The simula is the first object-oriented programming language.

Why do we use object-oriented programming?

- OOPs, make the development and maintenance of projects easier.

- OOPs provide the feature of data hiding that is good for security concern.
- We can provide the solution to real-world problems if we are using object-oriented programming.

OOPs Concepts:

Let’s look at the topics which we are going to discuss.

- Package
- Access Modifiers
- Class
- Object
- Constructor
- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Previous
Next
Package in Java

Package in Java

Package

A package in Java is a grouping of related classes and interfaces into a single unit that provides access control and namespace management.

It is recommended to combine related classes and interfaces into packages to make them easy to locate and use, prevent naming conflicts, and control access.

The Java platform's classes and interfaces are members of numerous packages that group classes by function: fundamental classes are in Java.lang, classes for reading and writing (input and output) are in java.io, classes for sql operations are in java.sql.

As a result, we may conclude that packages serve as an encapsulation mechanism for grouping similar classes and interfaces into a single category.

Need of Packages in Java

We should use packages in Java because of the following reasons:

1. **Naming Conflicts:** Since the package creates a new namespace, the names of the classes and interfaces will not conflict with the names of classes and interfaces in other packages. In Java, for example, there are two date classes: one in the SQL package and one in the util package. So, when using the package, we can distinguish between the Java.util and java.sql packages.
2. **Security:** We can provide unrestricted access to classes and interfaces within the package while restricting access to classes and interfaces outside the package, which secures our component.
3. **Modularity** and Maintainability**:** Since packages are groups of similar classes and interfaces, using them improves the application's modularity and maintainability.

Types of Packages in Java

In Java, there are two kinds of packages:

- **Built-in Package:** A built-in package is one that is already specified and is part of the Java API. For instance, java.io, java.lang, Java.util, java.io, java.net, and so on are some of the built-in packages.
- **User-defined Package:** The user-defined packages are those that are defined by the user or programmer.

Creating a Package in Java

We can create a package in Java by using the 'package' keyword followed by a package name and add it at the top of any source file that includes the classes and interfaces that we want to include in the package.

```
package packagename;
```

Here, the 'package' is a keyword and 'packagename' is the name of the package.

Note: The package declaration must appear on the first line of the source file. Each source file can only have one package declaration, and it must apply to all classes and interfaces in the file.

Naming a Package in Java

With programmers all over the world using the Java programming language to create classes and interfaces, it is possible that many programmers will use the same package name for different classes and interfaces.
To avoid conflicts with the names of classes or interfaces, package names are written in all lower case.
Companies begin their package names with their reversed Internet domain name, such as 'com.example.packagename' for a package called 'packagename' created by a programmer at 'example.com'.

Compile and Run the Java Package

Assume we have a file ‘PackageExample.java’ that we want to place in the package ‘package1’. Consider the following example.

```
// Creating a package named package1
package package1;

public class PackageExample {

   public static void main(String args[]) {
      System.out.println("Hello World");
   }

}
```

We can compile this file using:

```
javac -d . PackageExample.java
```

The -d specifies where to save the generated class file, and the .(dot) specifies the current working directory.
To run the file, we can use:

```
java package1.PackageExample.java
```

To run this class, we must use a fully qualified name. 'package1' is the package in this case, and 'PackageExample'is a class of it.

Accessing Package Members Outside its Package

There are three ways to access the public package member from outside its package:

- Import the member's entire package using 'import package.*;'
- Import the package member using 'import package.classname;'
- Refer to the member by its fully qualified name.

1. Importing an Entire Package:

To import all the classes and interfaces contained in a particular package, we use the import statement with the packagename.*;
Any class or interface in the package can now be referred to by its simple name.

Example:

```
// File Name: PackageExample1.java
```

```
// Creating a package package1
package package1;

public class PackageExample1 {
  public void display() {
      System.out.println("Coding Ninjas");
   }

}

// File Name: PackageExample2.java

// Creating a package package2
package package2;

// Importing entire package, package1
import package1.*;

public class PackageExample2 {

   public static void main(String args[]) {
      PackageExample1 obj = new PackageExample1();
      obj.display();
   }

}
```

Output:

```
Coding Ninjas
```

2. Importing a Package Member:

To import a specific member into the current file, we use import packagename.classname;
This import statement is placed at the beginning of the file, before any class defintions, but after the package statement, if one exists.

Example:

```
// File Name: PackageExample1.java

// Creating a package package1
```

```
package package1;

public class PackageExample1 {
  public void display() {
       System.out.println("Coding Ninjas");
   }

}

// File Name: PackageExample2.java

// Creating a package package2
package package2;

// Importing entire package, package1
import package1.PackageExample1;

public class PackageExample2 {

   public static void main(String args[]) {
       PackageExample1 obj = new PackageExample1();
       obj.display();
   }

}
```

Output:

```
Coding Ninjas
```

3. Referring to a Package Member by Its Fully Qualified Name:

We can also use a fully qualified name for a package member to access it. In this case, there is no need to import the package and whenever we want to access a class, we must use a fully qualified name.

Example:

```
// File Name: PackageExample1.java

// Creating a package package1
package package1;
```

```java
public class PackageExample1 {
  public void display() {
       System.out.println("Coding Ninjas");
   }

}

// File Name: PackageExample2.java

// Creating a package package2
package package2;

public class PackageExample2 {

   public static void main(String args[]) {
          // Using fully qualified name
       package1.PackageExample1 obj = new package1.PackageExample1();
       obj.display();
   }
}
```

Output:

```
Coding Ninjas
```

Previous

Next

Access Modifiers in Java:

Access Modifiers in Java

Access Modifiers

In Java, access modifiers define the accessibility of a method, variable, constructor, or class. There are four types of access modifiers available in Java.

- private

- default
- protected
- public

i) private

The private access modifiers are only accessible inside the class of which they are declared. Private data members and methods are not accessible from outside the class. If we attempt to access them from outside the class, the compiler will generate a compile-time error.
Let us look at the example below.

Example:

```
class PrivateExample {
        // Private data member
        private int val = 10;

        // Private method
        private void show() {
                System.out.println(val);
        }
}

public class CodingNinjas {

        public static void main(String args[]) {
                PrivateExample obj = new PrivateExample();

                // Trying to access the private variable
                obj.val += 10;

                // Trying to access the private method
                obj.display();
        }

}
```

Output:

```
CodingNinjas.java:17: error: val has private access in PrivateExample
                obj.val += 10;
```

```
                    ^
CodingNinjas.java:18: error: cannot find symbol
                obj.display();
                    ^
  symbol:   method display()
  location: variable obj of type PrivateExample
2 errors
```

ii) default

If a class, data member, or method is not declared with any modifiers, it is considered as a default modifier in Java. The specialty of default access modifiers is that they are only accessible inside the same package. If we attempt to access the default data members and methods from outside the package, the compiler will generate a compile-time error. Have a look at the example below.

Example:

In the example provided below, we have created two packages pack1 and pack2. The scope of class and method is the default, therefore we can't access it from outside the package.

```
// File Name: DefaultExample.java

package package1;

public class DefaultExample {
        void show() {
                System.out.println("Coding Ninjas");
        }
}

// File Name: CodingNinjas.java

package package2;

// Importing the DefaultExample class from package, 'package1'
import pack1.DefaultExample ;

public class CodingNinjas{

        public static void main(String args[]) {
                DefaultExample obj = new DefaultExample();

                // Trying to access the default method
```

```
                obj.show();
        }

}
```

Compilation and Run Process:

```
> javac -d . DefaultExample.java                // Compile the DefaultExample.java file
> javac -d . CodingNinjas.java                  // Compile the CodingNinjas.java file
> java package2.CodingNinjas                    // Run CodingNinjas.java
```

Output:

```
CodingNinjas.java:14: error: val is not public in DefaultExample; cannot be accessed from outside package
        obj.val += 10;
            ^
CodingNinjas.java:17: error: show() is not public in DefaultExample; cannot be accessed from outside package
                obj.show();
                    ^
2 errors
```

iii) protected

The protected access modifier allows access of data members and methods inside the same package and outside the package through inheritance. We'll learn about inheritance later.

Example:

```
// File Name: ProtectedExample.java

package package1;

public class ProtectedExample {
    protected int val = 10;
        protected void show() {
                System.out.println(val);
        }
}

// File Name: CodingNinjas.java

package package2;
```

```
// Importing the ProtectedExample class from package, 'package1'
import package1.ProtectedExample;

public class CodingNinjas extends ProtectedExample {

        public static void main(String args[]) {
                ProtectedExample obj = new ProtectedExample();

        // Trying to access the protected variable, 'val'
        obj.val += 10;

                // Trying to access the protected method
                obj.show();
        }

}
```

Compilation and Run Process:

```
> javac -d . ProtectedExample.java            // Compile the ProtectedExample.java file
> javac -d . CodingNinjas.java                // Compile the CodingNinjas.java file
> java package2.CodingNinjas                  // Run CodingNinjas.java
```

Output:

```
20
```

iv) public

The public data members, classes, and methods can be accessed from anywhere in the program. This access modifier can be accessed from inside the class, from outside the class, from within the package, and from outside the package. Consider the following example.

Example:

```
// File Name: PublicExample.java

package package1;

public class PublicExample {
    public int val = 10;
```

```
        public void show() {
                System.out.println(val);
        }
}

// File Name: CodingNinjas.java

package package2;

// Importing the PublicExample class from package, 'package1'
import package1.PublicExample ;

public class CodingNinjas {

        public static void main(String args[]) {
                PublicExample obj = new PublicExample();

        // Trying to access the public variable, 'val'
        obj.val += 10;

                // Trying to access the public method
                obj.show();
        }

}
```

Compilation and Run Process:

```
> javac -d . PublicExample.java                        // Compile the PublicExample.java file
> javac -d . CodingNinjas.java                 // Compile the CodingNinjas.java file
> java package2.CodingNinjas                   // Run CodingNinjas.java
```

Output:

```
20
```

Previous

Next

Classes in Java

Classes in Java

Class

A class is a logical entity that serves as the basis for the definition of a new data type. Once defined, we can use this new data type to create objects of that type. As a result, we can define a class as a template for an object. Variables, methods, and constructors are all part of a Java class.

Syntax:

```
Modifier class className {
    /*
    Class Body
    */
}
```

In the above syntax:

- **Modifiers:** A class can have public or default access modifiers**.**
- **class:** The 'class' keyword is used to create a class in Java**.**
- **ClassName:** The name of the class should begin with a capital letter**.**
- **Class Body:** The class body is surrounded by curly braces.

Previous

Next

Objects in Java

Objects in Java

Object

An object is an entity that has a state and behavior. To access the members who are defined inside the class, we need to create the object of that class.
A dog, for example, is an entity because it has states such as colour, name, breed, and so on, as well as actions such as barking, chewing, sleeping, and so on.

Creating an Object in Java

In Java, we can create an object by using the new keyword. The new keyword is used to allocate memory for an object dynamically and return a reference to it. As an example,

```
Dog obj = new Dog();
```

In this case, we have created an object of the **Dog** class, and **obj** is a reference to an object of the **Dog** type.

Consider the following example of how we should use class and object in our program.

Example:

```
class Box {

        // Member variables
        double width;
        double height;
        double depth;
}

public class CodingNinjas {
        public static void main(String args[]) {
                // Creating an object of Box class
                Box obj = new Box();

                // Assigning values to obj instance variables
                obj.width = 5;
                obj.height = 10;
                obj.depth = 15;

                // Computing the volume of the box
                double volume = obj.width * obj.height * obj.depth;

                System.out.println("Volume of Box: " + volume);
        }

}
```

Output:

```
Volume of Box: 750.0
```

Previous

Next

Static keyword in java

Static Keyword in Java

Static Keyword

The static keyword in Java is mostly used for memory management. Static keywords can be used for variables, methods, blocks, and nested classes.

Static fields or class variables are fields that have the static modifier in their declaration. They are associated with the class rather than any specific object. Every instance of the class has the same class variable, which is stored in a single fixed location in memory. A class variable's value can be changed by any object, but class variables can also be modified without making an instance of the class.

Static Block in Java

There might be a situation where we need to compute something to initialize our static variables. For this, we can declare a static block that is executed only once when the class is first loaded.

Example:

```
public class StaticBlockExample
{
        // Static variables
        static int a = 10;
        static int b;

        // Static block
        static {
                System.out.println("Static Block Initialized");
                b = a * 4;
        }

        public static void main(String[] args)
        {
                        System.out.println("Inside Main Method");
```

```
                        System.out.println("Value of a : "+a);
                        System.out.println("Value of b : "+b);
        }

}
```

Output:

```
Static Block Initialized
Inside Main Method
Value of a : 10
Value of b : 40
```

Static Variable in Java

When a variable is declared static, a single copy of the variable is created and shared by all objects at the class level. Static variables can only be created at the class level. Memory allocation for such variables occurs only once when the class is loaded into memory.

Example:

```
public class StaticVariableExample
{
        // Static variable
        static int a = fun();

        // Static block
        static {
                System.out.println("Inside Static Block");
        }

        // Static Method
        static int fun() {
                System.out.println("Inside fun() Function");
                return 10;
        }

        public static void main(String args[])
        {
                System.out.println("Value of a : "+a);
```

```
                System.out.println("Inside Main");
        }
    }
```

Output:

```
Inside fun() Function
Inside Static Block
Value of a : 10
Inside Main
```

Static Method in Java

When a method is declared with the static keyword, it is referred to as a static method. A static method is a class member rather than a class object. A static method can be called without first creating a class instance. The static method cannot refer to 'this' or 'super' in any way.

Example:

```
public class StaticMethodExample
{
    // Static method
        public static void add(int a, int b) {
                int sum = a + b;
                System.out.println(sum);
        }

        public static void main(String args[])
        {
                // Calling static method without creating an object
                add(10, 20);
        }
    }
}
```

Output:

```
30
```

Previous

Next

Final Keyword in java:

Final Keyword in Java

Final Keyword

In Java, the final keyword is only applicable to methods, variables, and classes. It is primarily used to control the user's ability to use variables, methods, and classes.

final variable

In Java, a variable is a final variable if it is declared with the final keyword. The final variable's value is assigned only once. We cannot modify the value of the final variable after it has been assigned. If we attempt to reassign the value, we will get a compile-time error.

Example:

```
public class FinalVariableExample {

        // Final variable.
        final int val = 50;

        void show() {

         // Trying to change the value of the final variable, 'val'
                val = 100;

                System.out.println(val);
        }

        public static void main(String args[]) {

                FinalVariableExample obj = new FinalVariableExample();
                obj.show();

        }
```

```
}
```

Output:

```
FinalVariableExample.java:9: error: cannot assign a value to final variable val
                val = 100;
                ^
1 error
```

final method

When a method is declared with the final keyword, it is referred to as the final method. The characteristic of the final method is that the child class cannot override it.
If we do not want our method's implementation to be changed, we should make it final.

Example:

```
class FinalMethodExample {

        // Final method
        final void show() {
                System.out.println("Inside FinalMethodExample Class Method");
        }

}

public class CodingNinjas extends FinalMethodExample {

        // Trying to override the final method, 'show'
        void show() {
                System.out.println("Inside Coding Ninjas Class Method");
        }

        public static void main(String args[]) {

                CodingNinjas obj = new CodingNinjas();
                obj.show();

        }
```

```
}
```

Output:

```
CodingNinjas.java:13: error: show() in CodingNinjas cannot override show() in FinalMethodExample
        void show() {
             ^
  overridden method is final
1 error
```

final class

When a class is declared with the final keyword, it is referred to as the final class. The special thing about a final class is that it cannot be extended. Therefore, to prevent a class from being inherited, we should use a final class.

Example:

```
// Final Class
final class FinalClassExample {
        /*
        Methods and Data Members
         */
}

// Trying to extend the final class, 'FinalClassExample'
public class CodingNinjas extends FinalClassExample {

        void show() {
                System.out.println("Inside Coding Ninjas Class Method");
        }

        public static void main(String args[]) {

                CodingNinjas obj = new CodingNinjas();
                obj.show();

        }

}
```

Output:

```
CodingNinjas.java:9: error: cannot inherit from final FinalClassExample
public class CodingNinjas extends FinalClassExample {
                                  ^
1 error
```

Previous

Next

Constructor in Java

Constructor in Java

What is a Constructor?

A constructor is a member function of a class, and its name is the same as the name of the class. When there is no constructor defined inside the class, the Java compiler provides a default constructor in the class. In Java, every class has a default constructor. The constructor has no return type. A constructor can be parameterized or overloaded.

When is a Constructor called?

When we use the new keyword to create an object of a class, a constructor is invoked to initialize the data members of the class.

Example:

```
public class ConstructorExample {
    ConstructorExample () {
        /*
        Body of Constructor
         */
    }

    public static void main(String[] args) {
        // Invoking Constructor
        ConstructorExample obj = new ConstructorExample();
```

```
    }

}
```

Rules for Creating a Constructor

- The constructor and class names must be the same.
- There is no return type in a constructor.
- Constructors in Java can never be static, final, or abstract.

Types of Constructor in Java

There are two types of constructors in Java.

- No-argument or default constructor
- Parameterized constructor

i) No-argument constructor

When a programmer does not define a constructor in the program, the java compiler creates a default constructor for the class. In Java, this constructor is referred to as the default constructor.

Example:

```
public class ConstructorExample {

    // Default Constructor
    ConstructorExample () {
        System.out.println("Coding Ninjas");
    }

    public static void main(String[] args) {
        // This would invoke Default Constructor of the class
        ConstructorExample obj = new ConstructorExample();
    }

}
```

Output:

```
Coding Ninjas
```

ii) Parameterized constructor

A parameterized constructor is one that accepts one or more parameters. Whenever we create an object of the class with a parameterized constructor, we need to pass the argument so that the constructor gets invoked after the object creation.

Example:

```
public class ConstructorExample {

    // Data Memnber
    String str;

    // Parameterize Constructor
    ConstructorExample (String s) {
        str = s;
    }

    public void show() {
        System.out.println(str);
    }

    public static void main(String[] args) {
        // This would invoke Parameterize Constructor of the class
        ConstructorExample obj = new ConstructorExample("Coding Ninjas");
        obj.show();
    }

}
```

Output:

```
Coding Ninjas
```

Constructor Overloading in Java

In Java, constructor overloading occurs when a class has several constructors with different parameters. In constructor overloading, every constructor performs different tasks.

Example:

```
class Employee {

    // Data Members
    String name;
        int employeeId;
        int salary;

        // Constructor with one argument
        Employee(int eId) {
                employeeId = eId;
        }

        // Constructor with two arguments
        Employee(int eId, String s) {
                employeeId = eId;
                name = s;
        }

        // Constructor with three arguments
        Employee(int eId, String s, int sal) {
                employeeId = eId;
                name = s;
                salary= sal;
        }

        // Method to display employeeId, name, and salary
        public void show() {
                System.out.println(employeeId + " " + name + " " + salary);
        }
}

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

                // Passing the values in the constructor
                Employee obj1 = new Employee(265);
```

```
                Employee obj2 = new Employee(895, "Shiv");
                Employee obj3 = new Employee(236, "Afzal", 70000);

                // Displaying the result
                obj1.show();
                obj2.show();
                obj3.show();
        }

}
```

Output:

```
265 null 0
895 Shiv 0
236 Afzal 70000
```

Previous

Next

super keyword in java

super keyword in Java

super keyword

- The super keyword can be used to refer to the instance variable of the immediate parent class.
- The super keyword can be used to call the methods of the immediate parent class.
- The super keyword can be used to call the constructor of the immediate parent class.

i) Referring to the Instance Variables of the Immediate Parent Class

Example:

```
// Base class 'Vehicle'
class Vehicle
{
```

```
        int maxSpeed = 120;
}

// Subclass 'Splendor' extending 'Vehicle'
class Splendor extends Vehicle
{
        int maxSpeed = 150;

        void show()
        {
                // Print 'maxSpeed' of base class, 'Vehicle'
                System.out.println("Maximum Speed: " + super.maxSpeed);
        }
}

public class Test
{

        public static void main(String args[])
        {
                Splendor sp = new Splendor();
                sp.show();
        }

}
```

Output:

```
Maximum Speed: 120
```

ii) Invoking the Method of the Immediate Parent Class

Example:

```
// Base class 'Person'
class Person
{
        void message()
        {
```

```
                System.out.println("This is base class method");
        }
}

// Subclass 'Student'
class Student extends Person
{
        void message()
        {
                System.out.println("This is subclass method");
        }

        // Note that display() is only in 'Student' class
        void display()
        {
                // This will invoke or call current class message() method
                message();

                // This will invoke or call parent class message() method
                super.message();
        }
}

public class Test
{

        public static void main(String args[])
        {
                Student s = new Student();

                // Calling display() of Student
                s.display();
        }

}
```

Output:

```
This is subclass method
This is base class method
```

iii) Invoking the Constructor of the Immediate Parent Class

Example:

```
// Parent class 'Person'
class Person
{
        // Default Constructor
        Person()
        {
                System.out.println("Parent class constructor");
        }
}

// Child class 'Student' extending the 'Person' class
class Student extends Person
{
        Student()
        {
                // Invoke or call parent class constructor
                super();
                System.out.println("Child class constructor");
        }
}

public class Test
{

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

}
```

Output:

```
Parent class constructor
```

```
Child class constructor
```

Constructor Chaining in Java

In Java, constructor chaining refers to the method of calling one constructor from another constructor with respect to the current object. Inheritance can result in constructor chaining in Java. When we want to perform several tasks in a single constructor, we use constructor chaining.

Important Points about Constructor Chaining:

- A constructor call from a constructor must appear in the first step.
- The first line of the constructor is either super() or this(). By default, it is super()
- This() and super() can never exist in the same constructor.
- Constructor chaining can be accomplished in two ways: using the this() keyword and using the super() keyword.

Constructor chaining within the same class using this() keyword.

```
public class Student {
        // Default constructor
        Student() {
                // Invoke parameterized constructor
                this("Coding Ninjas");

                System.out.println("Default constructor called");
        }

        // Parameterized constructor
        Student(String str) {
                System.out.println("Parameterized constructor called");
        }

        public static void main(String args[]) {
                // Invoke default constructor
                Student obj = new Student();
        }

}
```

Output:

```
Parameterized constructor called
Default constructor called
```

Constructor chaining to another class using the super() keyword.

```
class Person {

        // Person class default constructor
        Person() {
                // Invoke current class constructor 2
                this(10, 20);

                System.out.println("Person class default constructor called");
        }

        // Perosn class parameterized constructor
        Person(int a, int b) {
                System.out.println("Person class parameterized constructor called");
        }

}

class Student extends Person {

        // Student class default constructor
        Student() {
                // Invoke current class constructor
                this("Coding", "Ninjas");

                System.out.println("Student class default constructor called");
        }

        // Student class parameterized constructor
        Student(String str1, String str2) {
                // Invoke parent class constructor
                super();
```

```
                System.out.println("Student class parameterized constructor called");
        }
}


public class Test {

        public static void main(String args[]) {

                // Calls default constructor of 'Student' class
                Student obj = new Student();
        }

}
```

Output:

```
Person class parameterized constructor called
Person class default constructor called
Student class parameterized constructor called
Student class default constructor called
```

Previous

Next

Instance Initialization block in java

Instance Initialization Block in Java

Instance Initialization Block

The instance initialization block in java is used to initialize the instance data members. It is executed each time whenever an instance of the class is created. Instance Initializer block is executed before the constructor.

Rules for Instance Initialization Block

- The instance initialization block is executed when an instance of the class is created.
- The instance initialization block is invoked after the parent class constructor is invoked.

- It comes in the order in which they appear.

Example:

```
public class IIBExample {

        // Instance initialization block
        {
                System.out.println("Instance initialization block");
        }

        // Default constructor of class
        IIBExample() {
                System.out.println("Default constructor called");
        }

        public static void main(String args[])  {
                IIBExample obj = new IIBExample();
        }

}
```

Output:

```
Instance initialization block
Default constructor called
```

Multiple Instance Initializer Block

We can also create multiple instance initialization blocks in a single program. Whenever we create multiple instance initialization blocks in our program then they all are executed from top to down. i.e. the instance initialization block is written at the top will be executed first.

Example:

```
public class IIBExample {

        // Instance initialization block 1
        {
                System.out.println("Instance initialization block 1");
```

```
        }

        // Instance initialization block 2
        {
                System.out.println("Instance initialization block 2");
        }

        // Default constructor of class
        IIBExample() {
                System.out.println("Default constructor called");
        }

        // Instance initialization block 3
        {
                System.out.println("Instance initialization block 3");
        }

        public static void main(String args[])  {
                IIBExample obj = new IIBExample();
        }
}
```

Output:

```
Instance initialization block 1
Instance initialization block 2
Instance initialization block 3
Default constructor called
```

Instance Initialization Block with Parent Class

We can create an instance initialization block in the parent class also. The instance initialization block code executed immediately after the call to super() in a constructor. The compiler executes the parent class instance initialization block before executing the current class instance initialization block. Let’s look at the example below.

Example:

```
class IIBExample {
```

```
        IIBExample() {
                System.out.println("Parent class default constructor");
        }

        // Parent class instance initialization block
        {
                System.out.println("Parent class instance initialization block");
        }

}

public class Main extends IIBExample {

        Main() {
                super();
                System.out.println("Child class default constructor");
        }

        // Child class instance initialization block
        {
                System.out.println("Child class instance initialization block");
        }

        public static void main(String args[]) {
                IIBExample obj = new IIBExample();
        }

}
```

Output:

```
Parent class instance initialization block
Parent class default constructor
```

Previous

Next

Encapsulation in java

Encapsulation in Java

Encapsulation

The process of grouping data members and corresponding methods into a single unit is known as Encapsulation in Java. It is an important part of object-oriented programming. In Java, we can create a fully encapsulated class by making all the data members private. If the data members are private then it can only be accessible within the class, no other class can access the data members of that class. If we have declared all the data members of this class private then we can only use these data members by using getter and setter methods.

Example:

```
//File Name: Student.java

public class Student {

        // Private data members
        private String studentName;
        private int studentRollno;
        private int studentAge;

        // Getter method for student name to access private variable studentName
        public String getStudentName() {
                return studentName;
        }

        // Setter method for student name to set the value in private variable studentName
        public void setStudentName(String studentName) {
                this.studentName = studentName;
        }

        // Getter method for student rollno to access private variable studentRollno
        public int getStudentRollno() {
                return studentRollno;
        }

        // Setter method for student rollno to set the value in private variable studentRollno
        public void setStudentRollno(int studentRollno) {
                this.studentRollno = studentRollno;
        }

        // Getter method for student age to access private variable studentAge
```

```
        public int getStudentAge() {
                return studentAge;
        }

        // Setter method for student age to set the value in private variable studentAge
        public void setStudentAge(int studentAge) {
                this.studentAge = studentAge;
        }
        
}


// File Name: StudentDetails.java

public class StudentDetails {

        public static void main(String args[]) {
                Student obj = new Student();

                // Setting the values of the variables
                obj.setStudentName("Rahul");
                obj.setStudentRollno(101);
                obj.setStudentAge(22);

                // Printing the values of the variables

                System.out.println("Student Name : " + obj.getStudentName());
                System.out.println("Student Rollno : " + obj.getStudentRollno());
                System.out.println("Student Age : " + obj.getStudentAge());
        }

}
```

Output:

```
Student Name : Rahul
Student Rollno : 101
Student Age : 22
```

Advantages of Encapsulation in Java

1. Encapsulation is a way to achieve data hiding because other classes will not be able to access the data through the private data members.
2. In Encapsulation, we can hide the internal information of the data which is better for security concerns.
3. By encapsulation, we can make the class read-only i.e. a class that has only the getter method, and write only i.e. a class that has only the setter method.
4. The code reusability is also an advantage of encapsulation.
5. Encapsulated code is better for unit testing.

Previous

Next

Inheritance in java

Inheritance in Java

Inheritance

Inheritance is a crucial part of object-oriented programming. The process of inheriting the properties and behavior of an existing class into a new class is known as inheritance. When we inherit the class, we will reuse the original class's methods and fields in a new class. Inheritance may also be described as the **Is-A** relationship, also known as the **parent-child** relationship.

Let's understand the inheritance with the help of a real-world example.
Assume **Human** is a class with properties such as height, weight, age, and so on, as well as functionalities (or methods) such as eating(), sleeping(), dreaming(), working(), and so on.
We now want to create **Male** and **Female** classes. Males and Females are both humans, and since they share certain common properties (such as height, weight, age, and so on) and behaviors (such as eating(), sleeping(), and so on), they will inherit these properties and functionalities from the **Human** class. **Male** and **Female** each have their own set of characteristics (like men have short hair and females have long hair). Such properties may be added separately to the **Male** and **Female** classes.

This technique requires writing fewer codes since both classes inherited some properties and behaviors from the superclass and thus did not need to be rewritten. This also makes the code easier to understand.

Why we use inheritance?

- The primary advantage of inheritance is **code reuse**. When we inherit the methods and fields of an existing class into a new class, we can reuse the code.
- Only inheritance can achieve runtime polymorphism (method overriding).

Important Terminology in Inheritance

Super class:

The super class is the class whose features are inherited by a subclass. Super class is also known as the parent class or base class.

Sub class:

Sub class is the class that inherits the other class. Sub class is also known as the child class or derived class.

Java Inheritance Syntax:

```
class SuperClass {
   // Methods and Fields
}

class SubClass extends SuperClass {
      // Methods and Fields
}
```

The **extends** is a keyword used to inherit the class.

Types of Inheritance in Java

Java only supports three types of inheritance: single inheritance, multilevel inheritance, and hierarchical inheritance. Multiple inheritance and hybrid inheritance are only possible in Java via interfaces.

1. Single Inheritance

In single inheritance, one class can extend the functionality of another class. In single inheritance, there is only one parent class and one child class.

Syntax:

```
class SuperClass{
   //methods and fields
}
class SubClass extends SuperClass{
      //methods and fields
}
```

Example:

```java
// Parent class
class Animal {

        public void eat() {
                System.out.println("eating");
        }

}

// Child class
class Dog extends Animal {

        public void bark() {
                System.out.println("barking");
        }

}

public class Test {

        public static void main(String args[]) {

        // Creating an object of the child class
        Dog obj = new Dog();

        // Calling methods
        obj.eat();
        obj.bark();
        }

}
```

Output:

```
eating
barking
```

2. Multilevel Inheritance

When a class inherits from a derived class, and the derived class becomes the base class of the new class, it is called multilevel inheritance in Java. In multilevel inheritance, there is more than one level.

Syntax:

```
class A
{
    // Methods and Fields
}

class B extends A
{
   // Methods and Fields
}

class C extends B
{
   // Methods and Fields
}
```

Example:

```
class Animal {

        public void eat() {
                System.out.println("eating");
        }

}

class Dog extends Animal {

        public void bark() {
                System.out.println("barking");
        }

}

class BabyDog extends Dog {

        public void weep() {
                System.out.println("weeping");
        }
```

```
}

public class Test {

        public static void main(String args[]) {
                BabyDog obj = new BabyDog();

                // Calling methods
                obj.eat();
                obj.bark();
                obj.weep();
        }
}
```

Output:

```
eating
barking
weeping
```

3. Hierarchical Inheritance

In hierarchical inheritance, one class serves as a base class for more than one derived class.

Syntax:

```
class A
{
    // Methods and Fields
}

class B extends A
{
   // Methods and Fields
}

class C extends A
```

```
{
    // Methods and Fields
}
```

Example:

```
class Animal {

        public void eat() {
                System.out.println("eating");
        }

}

class Dog extends Animal {

        public void bark() {
                System.out.println("barking");
        }

}

class Cat extends Animal {

        public void meow() {
                System.out.println("meowing");
        }

}

public class Solution {

        public static void main(String args[]) {
                Cat obj = new Cat();

                // Calling Methods
                obj.eat();
                obj.meow();
                // obj.bark(); // compile time error

        }
```

```
}
```

Output:

```
eating
meowing
```

The Diamond Problem in Java

Because of the diamond problem, Java does not support multiple inheritance. Let's break down the diamond problem with the help of an example.

Consider the following scenario: A, B, and C are the three classes. Both the A and B classes are inherited by the C class. If two classes, A and B, have the same method, and we call it from a child class object (class C object). In this case, it would be unclear whether to call the methods of class A or class B. As a result, if we inherit two classes, the compiler will generate a compile-time error. Let's look at the example below to understand better.

Example:

```
class A {
        public void display() {
                System.out.println("In Class A");
        }
}

class B extends A {

        public void display() {
                System.out.println("In Class B");
        }
}

public class C extends A, B {

        public static void main(String args[]) {

                C obj = new C();

                // Creates ambiguity which display() method to call
```

```
                obj.display();
        }

}
```

Output:

```
C.java:14: error: '{' expected
class C extends A, B {
                         ^
1 error
```

The solution to the Diamond problem is to use default methods and interfaces. Using interfaces, we can achieve multiple inheritance in Java. We will cover interfaces in the latter section of the module.

Previous

Next

Polymorphism in java

Polymorphism in Java

Polymorphism

Polymorphism is viewed as an essential element of Object-Oriented Programming. Polymorphism is the ability to perform a single action in multiple ways.
Polymorphism is a combination of two Greek words. Poly means "many," and morphs means "forms." So polymorphism signifies a variety of forms. Let's look at a real-world example of polymorphism.

Real-life example:

A person may have multiple characteristics at the same time.
A woman, for example, is a sister, a mother, a wife, and an employee all at the same time. As a result, the same person behaves differently in different contexts. This is known as polymorphism.

Types of Polymorphism in Java

In Java, there are two types of polymorphism:

- Compile-time polymorphism
- Runtime polymorphism

1. Compile-time Polymorphism:

Compile-time polymorphism, also known as static polymorphism, is a type of polymorphism that occurs during the compilation process. We can use method overloading to accomplish this form of polymorphism.

Method Overloading:

When several methods with the same name but different parameters exist in a class, these methods are said to be overloaded. The key benefit of method overloading is that it improves program readability. Methods can be overloaded by using different numbers of arguments or (and) by using different types of arguments.

i) With different numbers of arguments

In the example given below, we have written two methods: the first multiply() method, which takes two arguments and multiply the two numbers, and the second multiply() method, which takes three arguments and multiplies the three numbers. Consider the following example:

Example:

```
class MethodOverloadingExample {

        // Function with two parameters
        public int multiply(int num1, int num2) {
                return num1 * num2;
        }

        // Function with three parameters
        public int multiply(int num1, int num2, int num3) {
                return num1 * num2 * num3;
        }
}

public class Main {

    public static void main(String args[]) {

        MethodOverloadingExample obj = new MethodOverloadingExample();

                // Method calling by passing arguments
                int productOfTwoNumbers = obj.multiply(10, 20);
                int productOfThreeNumbers = obj.multiply(10, 20, 30);

                System.out.println(productOfTwoNumbers);
```

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

Output:

```
200
6000
```

ii) With different types of arguments

We've created two add() methods in the example given below, each with a different data type.
The first add() method requires two integer arguments, while the second add() method requires two double arguments. Consider the following example.

Example:

```
class MethodOverloadingExample {

        // Function with integer parameters
        public int multiply(int num1, int num2) {
                return num1 * num2;
        }

        // Function with double parameters
        public double multiply(double num1, double num2) {
                return num1 * num2;
        }
}

public class Main {

    public static void main(String args[]) {

        MethodOverloadingExample obj = new MethodOverloadingExample();

                // Method calling by passing arguments
                int productInt = obj.multiply(10, 20);
                double productDouble = obj.multiply(10.0, 20.0);

                System.out.println(productInt);
```

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

Output:

```
200
200.0
```

Runtime Polymorphism

Dynamic polymorphism is another name for runtime polymorphism. In Java, method overriding is a technique for implementing runtime polymorphism. It is also sometimes known as Dynamic method dispatch.

Method Overriding

Method overriding is a feature that allows you to redefine the base class method in the derived class depending on the requirement of the derived class. In other words, whatever methods are available in the base class are by default also available in the derived class. However, a derived class can be dissatisfied with the base class method implementation at times. Then, the derived class can redefine the method based on its requirements. This process is called method overriding.

Rules for Method Overriding:

- One class (derived class) should inherit another class (base class).
- The base class's method and the derived class's method must have the same name.
- The parameters of the base class's method and the derived class's method must be the same.

Example:

```
class Parent {
        public void show()
        {
                System.out.println("Inside parent class");
        }
}

class subclass1 extends Parent {
        public void show()
        {
                System.out.println("Inside subclass1");
        }
```

```
}

class subclass2 extends Parent {
        public void show()
        {
                System.out.println("Inside subclass2");
        }
}


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

                // Upcasting
                p = new subclass1();
                p.show();

                p = new subclass2();
                p.show();
        }
}
```

Output:

```
Inside subclass1
Inside subclass2
```

Method Hiding in Java

If a subclass defines a static method with the same signature as a static method in a superclass, the subclass method hides the superclass method. It is only possible to hide methods if both the parent and child classes have static methods.

Example:

```
class Parent {
        public static void show() {
                System.out.println("Inside parent class");
        }
}
```

```
class Child extends Parent {
        public static void show() {
                System.out.println("Inside subclass");
        }
}

public class Main {
        public static void main(String args[]) {
                Parent p = new Parent();
                p.show();

                Parent c = new Child();
                c.show();
        }
}
```

Output:

```
Inside parent class
```

In the above example, we assume that p.show() will call the show() method from the parent class and c.show() will call the show() method from the child class, just like in overriding, but since show() is a static method, we have hidden the show() method rather than overriding it.

Previous

Next

Abstraction in java

Abstraction in Java

Abstraction

Abstraction is the mechanism of hiding implementation details and displaying only essential functionalities to the user.
Let’s consider a real-life example to understand the abstraction.

Consider the following real-life scenario: a man is watching television. The man only knows that by pressing the buttons on the remote, he can change the channel, increase/decrease the volume, increase/decrease the brightness, and so on. Yet he has no idea how, as he presses the buttons on the remote, the volume actually increases; he is unaware of the inner mechanism of the television. This is what abstraction is.

In Java, there are two ways to achieve abstraction:

- Abstract class
- Interface (achieve 100 percent abstraction)

Abstract Class and Abstract Method

- An abstract class is one that has been declared with the abstract keyword.
- An abstract method is one that is declared but does not have a method body. It does not have a complete implementation.
- Both abstract methods may or may not be present in an abstract class. Some of them might be concrete methods.
- Any class that includes one or more abstract methods must also include the abstract keyword in its declaration.
- We cannot use a new keyword to instantiate an abstract class.

When to use abstract class and abstract method?

When we are unsure about the implementation of a method in our program, we declare it abstract. For example, assume we have a class Vehicle with a method **getNoOfWheels()** that returns a set of wheels. But we have no idea how many wheels we have to return since each vehicle has a unique number of wheels. As a result, we abstract this method.

In a program, there are instances where the implementation of the class is incomplete; this sort of partially implemented class is declared as abstract.

Example:

Suppose we have a class **School** with a method **getAdmissionFee()** and subclasses of banks such as DAV, DPS, JNV, etc. Since admission fees vary from school to school, implementing this method in the parent class (**School** class) is meaningless. This is due to the fact that each class must override this method in order to have its own implementation details.

```
// Abstract class
abstract class School {

        // Non-abstract method
        public void show() {
                System.out.println("School Application");
        }

        // Abstract method
        abstract int getAdmissionFee();
}
```

```
class DAV extends School {

        // Overriding the method from 'School' class
        int getAdmissionFee() {
                return 25000;
        }
}

class DPS extends School {

        // Overriding the method from 'School' class
        int getAdmissionFee() {
                return 35000;
        }
}

class JNV extends School {

        // Overriding the method from 'School' class
        int getAdmissionFee() {
                return 1000;
        }
}

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

                School s;

                s = new DAV();
                s.show();

                System.out.println("DAV Admission Fee: " + s.getAdmissionFee());

                s = new DPS();
                System.out.println("DPS Admission Fee: " + s.getAdmissionFee());

                s = new JNV();
                System.out.println("JNV Admisssion Fee: " + s.getAdmissionFee());
        }
```

```
}
```

Output:

```
School Application
DAV Admission Fee: 25000
DPS Admission Fee: 35000
JNV Admisssion Fee: 1000
```

Interface in Java

An interface is a blueprint of a class. An interface, like a class, may have methods and variables, but the methods declared in an interface are always abstract. Thus, the interface is a way to achieve complete abstraction.

Why we use interface in Java?

- Complete abstraction is achieved by the use of the interface.
- Since Java does not allow multiple inheritance in the case of classes, we can achieve multiple inheritance by using interfaces.
- It can be used for loose coupling.

Why use interfaces when we have abstract classes:

Abstraction is achieved by the use of interfaces and abstract classes. However, the reason for using interfaces is that abstract classes may contain non-final variables, while interface variables are public, final, and static.

Syntax:

```
interface interface_name {
        /*
        Constant Fields and Abstract Methods
         * /
}
```

Here, the **interface** is a keyword used to create an interface.

Example:

```
interface InterfaceExample {

        // public, static and final data member
        int val = 10;
```

```
        // public and abstract method
        void show();
}

public class Main implements InterfaceExample {

        // Overriding the abstract method
        public void show() {
                System.out.println("Coding Ninjas");
        }

        public static void main(String args[]) {
                Main t = new Main();
                t.show();
                System.out.println(val);
        }
}
```

Output:

```
Coding Ninjas
10
```

Multiple Inheritance in Java using an Interface

Because of the ambiguity problem, Java does not support multiple inheritance in the case of classes. However, we can use interfaces to implement multiple inheritance in Java since there is no ambiguity in the case of interfaces. It is because of its implementation provided by the implementation class.

Example:

```
interface Interface1 {

        // Default method
        default void show() {
                System.out.println("Default Interface 1");
        }
}

interface Interface2 {
```

```
        // Default method
        default void show() {
                System.out.println("Default Interface 2");
        }
}

public class Main implements Interface1, Interface2 {

        // Overriding default method
        public void show() {
                // Using super keyword to call the show() method of interface, 'Interface1'
                Interface1.super.show();

                // Using super keyword to call the show() method of interface, 'Interface2'
                Interface2.super.show();
        }

        public static void main(String args[]) {
                Main obj = new Main();
                obj.show();
        }
}
```

Output:

```
Default Interface 1
Default Interface 2
```

Previous

Next

Introduction

Introduction to Exception Handling

Introduction

An exception in Java is an unexpected occurrence that occurs during the execution of a program. It disrupts the regular flow of the program, and an exception terminates the program execution.
When an exception occurs, we receive a system-generated error message. However, we can handle the exception in Java to provide a meaningful message to the user rather than a system-generated error.

Why does an Exception occur?

Exceptions in Java can happen for a variety of reasons such as:

- The user entered incorrect information.
- Attempting to open a file that does not exist in the program.
- Network connection problem
- Server down problem

What is Exception Handling in Java?

Exception handling is a mechanism in Java used to handle run-time errors such that we can maintain the program's regular flow of execution. If an exception occurs, the remaining code is not executed. As a result, the java compiler creates an exception object, and this exception object directly jumps to the default catch mechanism. Thus, the program is terminated after a default message is printed on the screen.

Advantage of Exception Handling

The primary advantage of exception handling is that it keeps the program's regular flow preserved. By using the exception handling mechanism, all of the statements in our program will be executed, and the normal flow of the program will be sustained as well as a user-friendly message will be generated rather than a system-generated error message. That is why Exception Handling is used in Java.

Exception vs. Error

Error:

Our programs do not cause errors the majority of the time. Instead, these are caused by a lack of system resources.
Also, errors cannot be recovered.
For e.g., if our program encounters OutOfMemory, we are helpless to respond, and the program will terminate abnormally.

Exception:

An exception is an unwanted event that occurs during program execution. It interrupts the regular flow of the program.
Unlike errors, exceptions are recoverable.
Exceptions include *NullPointerException*, *IOException*, and *ArithmeticException*, to name a few.

Previous

Next
Keywords in exception handling

Keywords in Exception Handling

Exception Handling Keywords in Java

There are five keywords that we can use in exception handling.

1. try

We can write code that might throw an exception in the try block. A try block in Java must be followed by a catch block or finally block.

2. catch

In Java, the catch block is used to handle any exceptions that might occur in our software. We can only use it after the try block. For a single try block, we can use multiple catch blocks.

3. finally

In Java, the finally block is used to clean up code or release resources that are being used by the program. Whether or not the exceptions are dealt, the finally block is always executed.

4. throw

In Java, the keyword throw is used to throw an exception. It has the ability to throw exceptions explicitly. Using the throw keyword, we can throw either checked or unchecked exceptions.

5. throws

In Java, the throws keyword is used to declare an exception. Using the throws keyword, we can only declare checked exceptions. When we declare an exception, it is the programmer's responsibility to write the exception handling code.

Previous

Next

Exception Class Hierarchy

Java Exception Class Hierarchy

Following is the hierarchy of the Java Exception Classes:

- In Java, the throwable class is the parent class of all exception classes.
- There are two child classes in the throwable class, i.e., Error and Exception.
- The Exception class represents exceptions that the programmer can handle using the try-catch block.
- RuntimeException is a class that is a subclass of the Exception class and comprises the majority of the unchecked exceptions. Unchecked Exceptions include ArithmeticException, NullPointerException, NumberFormatException, etc.

Previous
Next
Types of exceptions

Types of Exceptions

Types of Exceptions in Java

In Java, there are two types of exceptions:

- Checked Exception
- Unchecked Exception

Checked Exception

Except for Error and RuntimeException classes, all other classes that inherit the Throwable class are the checked exceptions.
At compile time, the compiler checks the checked exceptions.
If the compiler does not handle the exception, we will get a compile-time error in the case of checked exceptions. These exceptions can be handled using try-catch blocks or by declaring the exception with the throws keyword.

Example:

```
import java.io.*;

public class CheckedExceptionExample {

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

```
        // Throws FileNotFoundException
                FileInputStream fileInStream = new FileInputStream("E:/CodingNinjas.txt");;

                int c;

                // Throws an IOException
                while((c = fileInStream.read()) != -1 ){
                        System.out.println((char)c);
                }

                // Throws an IOException
                fileInStream.close();
        }
}
```

Output:

```
CheckedExceptionExample.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
                FileInputStream fileInStream = new FileInputStream("E:/CodingNinjas.txt");;
                                               ^
CheckedExceptionExample.java:13: error: unreported exception IOException; must be caught or declared to be thrown
                while((c = fileInStream.read()) != -1 ){
                                           ^
CheckedExceptionExample.java:18: error: unreported exception IOException; must be caught or declared to be thrown
                fileInStream.close();
                             ^
3 errors
```

How to Handle Checked Exceptions?

In Java, there are two ways to handle checked exceptions:

- Using try-catch
- Using throws keyword

Using try-catch:

To handle the checked exception, we can use try-catch. Handling exceptions is always a good idea because we should provide a meaningful message to the user for each exception type so that the user can easily understand the exception message.

Example:

In this example, we have created a file called CodingNinjas.txt in our current working directory. Now using the program, we are reading the text from it, and printing it to the console. In this program, we have used try-catch to manage all possible exceptions.

```
File Name: CodingNinjas.txt

254 Shiv 22
587 Afzal 30
```

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class HandleCheckedExceptionExample {

    public static void main(String[] args) {
        FileReader fileReader = null;

        try {
            fileReader = new FileReader("CodingNinjas.txt");
        } catch (FileNotFoundException e1) {
            System.out.println("File does not exist in the current directory");
        }

        int c;

        try {
            while ((c = fileReader.read()) != -1) {
                System.out.print((char) c);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
254 Shiv 22
587 Afzal 30
```

Declare the exception using throws:

Checked exceptions, as we know, occur within the main() method. To prevent the compilation error, declare the exception in main() using the throws keyword. For example, we can declare the exception as follows.

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

}
```

The parent class of FileNotFoundException is IOException. As a result, it can handle the FileNotFoundException by default.

Example:

```
import java.io.FileInputStream;
import java.io.IOException;

public class HandleCheckedExceptionExample {

    // Declaring exception using a throws keyword
    public static void main(String args[]) throws IOException {

        FileInputStream fileInStream = new FileInputStream("CodingNinjas.txt");

        int c;

        while ((c = fileInStream.read()) != -1) {
            System.out.println((char) c);
        }

        fileInStream.close();
    }
}
```

Unchecked Exception

Unchecked Exception refers to all classes that inherit from RuntimeException. The compiler does not check Unchecked Exceptions at compile time. However, they are checked at runtime. In unchecked exceptions, we do not get a compile-time error if the programmer does not handle the exception. The majority of the time, this exception happens due to incorrect data entered by the user (divide by zero). Unchecked exceptions include ArithmeticException, NullPointerException, ArrayIndexOutOfBoundException, and others.

Example:

In the program given below, we are dividing the two numbers that the user has entered. If the user enters correct data (i.e., if the divisor is not zero), the program will display the quotient of two numbers; otherwise, it will display ArithmeticException caused by division by zero.

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Taking input for the first value
        System.out.println("Enter the first value: ");
        int val1 = sc.nextInt();

        // Taking input for the second value
        System.out.println("Enter the second value: ");
        int val2 = sc.nextInt();

        /*
        Dividing the 'val1' by 'val2'. If division is not successful, the program will terminate here by throwing an exception.
        Otherwise, the program will continue.
        */
        int quotient = val1 / val2;

        System.out.println("Quotient of " + val1 + "/" + val2 + " is: " + quotient);

        System.out.println("\nProgram executed successfully!");
    }
}
```

Output: When a user enters the correct data

```
Enter the first value:
52
```

```
Enter the second value:
6
Quotient of 52/6 is: 8

Program executed successfully!
```

Output: When a user enters the incorrect data

```
Enter the first value:
5
Enter the second value:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:21)
```

How to Handle Unchecked Exceptions?

Using try-catch, we can handle the unchecked exception. The try statement allows users to specify a block of code that will be checked for errors. In contrast, the catch block will capture the provided exception object and perform the necessary operations. Hence, the program will not terminate.

Example:

```
public class HandleUncheckedExceptionExample {

    public static void main(String[] args) {

        // Try block for possible exceptions
        try {
            System.out.println(5 / 0);
        }

        // Catch block for catching exception object
        catch (ArithmeticException e) {

            // Printing the error message
            System.out.println("Number can not be divided by zero!");
        }

        System.out.println("Program executed successfully!");
```

```
    }
}
```

Output:

```
Number can not be divided by zero!
Program executed successfully!
```

Previous

Next

Multiple catch block

Multiple Catch Block

Multiple Catch Block in Java

One or more catch blocks can be used after a try block. So, in Java, if we want to perform different tasks in response to different exceptions, we can use multiple catch blocks.

Note that only one exception can occur at a time, and only one capture block can be executed at a time. All catch blocks must be executed in the order specified, beginning with the most specific and moving to the most general.

Example:

In this example, the try block contains two exceptions. But at a time, only one exception occurs, and one catch block is executed.

```
public class MultipleCatchBlockExample {

        public static void main(String[] args) {

                try {
                        int array[] = new int[5];
                        array[6] = 8;

                        int val1 = 10;
                        int val2 = 0;
                        int quotient = val1 / val2;

                        System.out.println("Quotient of " + val1 + " / " + val2 + " is: " + quotient);
                }
```

```
                catch(ArithmeticException e) {
                        System.out.println("Number can not divide by zero");
                }
                catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index out of bounds");
                }
                catch(Exception e) {
                        System.out.println("Other types of exception");
                }
        }
}
```

Output:

```
Array index out of bounds
```

Previous

Next

Explicitly throw an exception

Explicitly Throw an Exception

Explicitly Throw an Exception in Java

In Java, the throw keyword is used to explicitly throw an exception. Using the throw keyword, we can throw either checked or unchecked exceptions. The throw is mostly used to throw user-defined exceptions.

Syntax:

```
throw new exception_class("Error Message");
```

Example:

We've created two variables in this example: **availBalanace** and **withdrawAmount**. An Arithmetic exception occurred because the **availBalance** was less than the **withdrawAmount**. As a result, we throw an object of the **ArithmeticException** class, which displays the message **Insufficient Balance** on the screen.

```
public class ExceptionExample {
```

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

        int availBalance = 5000;
        int withdrawAmount = 6000;

        try {
            if (availBalance < withdrawAmount) {
                throw new ArithmeticException("Insufficient Balance");
            } else {
                availBalance -= withdrawAmount;
                System.out.println("Transactions completed successfully:");
                System.out.println("Available Balance : " + availBalance);
            }
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
Insufficient Balance
```

Previous

Next

Finally block

Finally Block

Finally Block in Java

In Java, a finally block is used to execute important code such as closing a connection, stream, etc. Whether or not the exceptions are handled, the finally block is always executed.

Usage of Java finally Block

There are 3 possible scenarios where we can use the finally block:

Exception does not occur

In this scenario, the program runs without throwing an exception, and the finally block is executed after the try block.

Example:

```
public class ExceptionExample {

	public static void main(String[] args) {

		try {
			System.out.println("try block");
			System.out.println(36 / 2);
		}
		catch(ArithmeticException e) {
			System.out.println("Arithmetic Exception Occurred");
		}
		finally {
			System.out.println("Finally block is always executed");
		}
	}
}
```

Output:

```
try block
18
Finally block is always executed
```

Exception occurs and not handled by the catch block

In this scenario, the program throws an exception but is not handled by the catch block. So, the finally block executes after the try block, and the program terminates abnormally.

Example:

```
public class ExceptionExample {

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

```
                try {
                        System.out.println(36 / 0);
                }
                catch(NullPointerException e) {
                        System.out.println("Exception not handled by the catch block");
                }
                finally {
                        System.out.println("Finally block is always executed");
                }
        }
}
```

Output:

```
Exception in thread "main" Finally block is always executed
java.lang.ArithmeticException: / by zero
        at org.jdbc.io.main.ExceptionExample.main(ExceptionExample.java:8)
```

Exception occurs and handled by the catch block

In this scenario, the program throws an exception but is not handled by the catch block. So, the finally block executes after the try block, and the program terminates abnormally.

Example:

```
public class ExceptionExample {

        public static void main(String[] args) {

                try {
                        System.out.println(12/ 0);
                }
                catch(ArithmeticException e) {
                        System.out.println("Exception handled by the catch block");
                }
                finally {
                        System.out.println("Finally block is always executed");
                }
        }
}
```

Output:

```
Exception handled by the catch block
Finally block is always executed
```

Previous

Next

User-defined exception

User-defined Exception

User-defined Exception in Java

If we create our own exception, it is referred to as a Custom or User-defined Exception. Java allows us to create our own exception, which is essentially a derived class of Exception. To create our own exception, we must first create a class that extends the Exception class and represents user-defined exceptions. We must pass the string to the constructor of the super class, which is obtained by calling the getMessage() function of the newly created object.

Example:

```
// A class that represents user-defined exception
public class InvalidPinCodeException extends Exception {

    // Call constructor of parent Exception
    public InvalidPinCodeException(String s) {
        super(s);
    }

}

public class ExceptionExample {

    // Creating a method to validate pin code
    public void validatePinCode(int pinCode) throws InvalidPinCodeException {
        if(!(pinCode >= 100000 && pinCode <= 999999)) {

            // Throw an object of user defined exception
            throw new InvalidPinCodeException("Pin Code must be of 6 digits");
        }
```

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

        public static void main(String[] args) throws InvalidPinCodeException {
                ExceptionExample obj = new ExceptionExample();
                obj.validatePinCode(12345);
        }
}
```

Output:

```
Exception in thread "main" org.jdbc.io.main.InvalidPinCodeException: Pin Code must be of 6 digits at ExceptionExample.validatePinCode(ExceptionExample.java:7) at
ExceptionExample.main(ExceptionExample.java:14)
```

Previous

Next

Introduction to Collections

Introduction to Collections

Introduction

Before diving into the "technical stuff," it's important for a beginner to understand what a "Collection" is. Quite intriguing, isn't it? However, you might already know what it is, or you can look it up in the dictionary - it means "*a group of things or people.*"

Now, let us try to relate the meaning of the word in the light of technical learning.
"Things or People" - Can you recall and relate it to any of the lectures from our Data Structures and Algorithms course? Okay, for those of you who have not taken the course with us, let me take this opportunity to explain.
"Things or People" have a lot of **attributes/properties and functionalities**. Are you confused? Don't worry; let me get this through.

For instance, take a car, which you can say is a "thing" or an "object"; doesn't that sound familiar to you? Anyway, let's look closely. A car has quite a few attributes or properties, such as the color, model name, manufacturer's name, whether the car runs on petrol or diesel, etc. A car has also gotten some functionalities. One of the significant functionalities is carrying passengers to point A to point B.

So, in Object-Oriented Programming(OOP), we see everything as an object! An object has attributes and functions. Moreover, they are closely bound or coupled, which is the idea of Encapsulation to represent a real-world entity. Similarly, a person can also be referred to as an object. Example, Employee!
In OOP, we have data all around. We store and edit them as objects and do manipulation to carry out a specific task, say, sorting objects based on a property!

Let’s revisit the definition of *“Collection”* from the dictionary now. So, Collection is “a group of things or people”, but now we know things or people are Objects in the world of programming. Hence, can I not say that a Collection is a group of objects? I certainly can.

Now, let’s look at it in the light of Java. Collections are a group of objects that help us to carry out some specific tasks *“efficiently”*.

Let’s dive deeper into the implementation of this concept.

Previous

Next

Implementation of Collections

Implementation of Collections

Implementation of Collections in Java

The collection is the root interface in the collection hierarchy and represents a group of objects known as its elements. Some allow duplicate elements, while others do not. Some are ordered and others are not. The JDK does not provide direct implementations of this interface; instead, it provides implementations of more specific subinterfaces such as Set and List. This interface is usually used to pass collections around and manipulate them in situations where maximum generality is needed.

The collections framework consists of three categories:

1. Collection Interfaces
2. Collection classes that implement the interfaces
3. Collection Algorithms

Previous
Next
Why Java Collections ?

Why Java Collection?

Why do we use a Java collection?

There are many advantages of using Java collections, such as:

- Since these data structures and algorithms have already been written, they can be reused.
- It reduces the time and effort taken to implement these data structures and algorithms.

- Since these are already in existence, it reduces the time and effort required to learn, use, and design new APIs.
- Java collections provide high-performance and high-quality data structures and algorithms, increasing speed and quality.
- Unrelated APIs can pass collection interfaces back and forth.

Previous
Next
Java Collection Framework Hierarchy

Java Collection Framework Hierarchy

As we have learned, the Java collection framework includes interfaces, classes, and algorithms. Now, let us see the Java collections framework hierarchy.

Previous

Next

Collection Interface

Collection Interface

1. Collection Interface in Java

The Collection interface is the interface that is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Methods of the Collection Interface

This interface includes a number of methods that can be used directly by all collections that implement it. They are as follows:

Method	Description

boolean add(E e)	It ensures that the specified element is present in this collection (optional operation).
boolean addAll(Collection<? extends E> c)	It adds all of the elements from the specified collection to this collection (optional operation).
void clear()	It removes all of the elements from this collection (optional operation).
boolean contains(Object o)	If this collection contains the specified element, this method returns true.
boolean containsAll(Collection<?> c)	If this collection contains all of the elements in the specified collection, it returns true.
boolean equals(Object o)	This method compares the defined object to this collection to see if they are equal.
int hashCode()	This method returns the hash code value for this collection.
boolean isEmpty()	If this collection has no elements, this method returns true.
Iterator<E> iterator()	It returns an iterator over the elements in this collection.
default Stream<E> parallelStream()	It returns a possibly parallel Stream with this collection as its source.
boolean remove(Object o)	It removes a single instance of this collection's specified element if it is present (optional operation).
boolean removeAll(Collection c)	It removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean removeIf(Predicate<? super E> filter)	This method is used to remove all elements from this collection that meet the specified predicate.
boolean retainAll(Collection<?> c)	Only the elements in this collection that are found in the specified collection are retained (optional operation).
int size()	It returns the number of elements in this collection.
default Spliterator<E> spliterator()	This method creates a Spliterator over the elements in this collection.
default Stream<E> stream()	This method is used to return a sequential Stream whose source is this collection.
Object[] toArray()	This method returns an array that contains all of the elements in this collection.

Previous

Next

Iterator Interface

Iterator Interface

2. Iterator Interface in Java

An iterator is an interface that iterates through the elements. It's used to traverse the list and modify the elements. The Iterator interface includes three methods, which are listed below:

public void remove()	The last element returned by the iterator is removed by this method.
public Object next()	It returns the element and moves the cursor pointer to the next element.
public boolean hasNext()	If the iterator has more elements, then this method returns true.

Previous

Next

Iterable Interface

Iterable Interface

3. Iterable Interface in Java

The Iterable interface serves as the root interface for all collection classes. Since the Collection interface extends the Iterable interface, all Collection interface subclasses must implement the Iterable interface as well.

It only has one abstract method. i.e., **Iterator<T> iterator()**
It returns an iterator over the elements of type T.

Previous

Next

List Interface

List Interface

4. List Interface in Java

The collection interface has a child interface called the list interface. Classes that require ordered data implement this interface, and these classes include ArrayList, Vector, Stack, etc. This also allows for the existence of duplicate data.
Since all of the subclasses implement the list, we can create a list object with any of them. As an example,

```
List <T> arrayList = new ArrayList<> ();
List <T> linkedList = new LinkedList<> ();
List <T> vector = new Vector<> ();

Where T is the type of Object.
```

We will only discuss ArrayList in this section, and we will talk more about other classes after learning data structures.

ArrayList Class

The ArrayList class is a resizable array found in the Java.util package.

The main distinction between an ArrayList and an array is that we cannot change the size of an array. To add elements to a full array, we must create a new array and copy all of the old elements from the old array to the new array. However, in an ArrayList, the elements can be added and removed at any time, due to which it grows and shrinks dynamically.

ArrayLists in Java, like arrays, may have duplicate objects. Since it implements the List interface, we can use all of the methods of the List interface with an array list. Internally, the ArrayList keeps track of the insertion order. **It derives from AbstractList and implements the List Interface.**

The Java ArrayList class has the following key features:

- It can contain duplicate elements.
- The insertion order is maintained.
- The ArrayList class is non-synchronized.
- Random access to any index is permitted, just as it is in an array.

Constructors of ArrayList

Constructor	Description
ArrayList()	It is used to create an empty array list.
ArrayList(int capacity)	It is used to create an array list with the required initial capacity.
ArrayList(Collection<? extends E> c)	It is used to create an array list that is initialized with the elements of the collection c.

Example to Show Methods and Constructor of ArrayList:

```
import java.io.*;
```

```java
// Importing the ArrayList class
import java.util.ArrayList;


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

        // Declaring the ArrayList using default constructor.
        ArrayList<Integer> list = new ArrayList<Integer>();

        // Adding items to the 'list'.
        list.add(3);
        list.add(2);
        list.add(1);
        list.add(4);

        // Printing the 'list'.
        System.out.println("The list is: " + list);

        // Accessing elements at index '0' and '1' from the 'list'.
        int element1 = list.get(0);
        int element2 = list.get(1);
        System.out.println("First element is: " + element1);
        System.out.println("Second element is: " + element2);

        // Modifying the element at index '0' by 100 in the 'list'.
        list.set(0, 100);
        System.out.println("The list after updating the first element by 100: " + list);

        // Removing an element at index '2' from the 'list'.
        list.remove(2);
        System.out.println("The list after removing the third element: " + list);

        // Finding the size of the 'list'.
        int size = list.size();
        System.out.println("Size of the list is: " + size);

        // Traversing the 'list'.
        for(int i = 0 ; i < list.size() ; i++) {
            System.out.println("Element at index " + i + " is :" + list.get(i));
```

```
        }

        // Removing all the elements from the 'list'.
        list.clear();
        System.out.println("The list after removing all the elements: " + list);
    }
}
```

Output:

```
The list is: [3, 2, 1, 4]
First element is: 3
Second element is: 2
The list after updating the first element by 100: [100, 2, 1, 4]
The list after removing the third element: [100, 2, 4]
Size of the list is: 3
Element at index 0 is :100
Element at index 1 is :2
Element at index 2 is :4
The list after removing all the elements: []
```

Internal Working of an Array List

While creating an array list, we do not specify any size because an array list is a dynamic array that grows and shrinks as we add and delete items dynamically.
The actual library implementation of the array list is more complicated. However, we can describe the fundamental idea to illustrate how the array list behaves when it is full, and we want to add an element to it.

- On heap memory, a larger-sized memory (for example, a memory of double size) is created.
- The contents of the present memory are then copied to the new memory.
- The new item is finally inserted in the array list because now, there is more memory available.
- Delete the previous memory.

Methods of ArrayList

Methods	Description
void add(int index, E element)	This method inserts the specified element into this list at the specified position.
boolean add(E e)	This method appends the specified element to the end of this list.
boolean addAll(Collection<? extends E> c)	This method appends all of the elements in the specified collection to the end of this list, in the order in which they are returned by the specified collection's Iterator.
boolean addAll(int index, Collection<? extends E> c)	This method inserts all of the elements in the specified collection into this list, starting at the specified position.

void clear()	This method removes all of the elements from this list.
void ensureCapacity(int requiredCapacity)	This method is used to enhance the capacity of an ArrayList instance.
E get(int index)	This method returns the element at the specified position in this list.
boolean isEmpty()	If this list contains no elements, this method returns true. Otherwise, returns false.
int lastIndexOf(Object o)	This method returns the index of the last occurrence of the specified element in this list, or -1 if the element does not exist.
Object[] toArray()	This method returns an array that contains all of the elements in this list in the proper order (from first to last element).
<T> T[] toArray(T[] a)	This method returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
Object clone()	This method returns a shallow copy of the current ArrayList instance.
Boolean contains(Object o)	If the list contains the specified element, this method returns true.
int indexOf(Object o)	This method returns the index of the first occurrence of the specified element in this list, or -1 if the element does not exist in the list.
E remove(int index)	This method is used to remove the element from the list present at the specified position.
boolean remove(Object o)	This method removes the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	This method removes all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	This method removes all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	This method removes all the elements that lies within the given range.
void replaceAll(UnaryOperator<E> operator)	This method replaces all the elements from the list with the specified element.
void retainAll(Collection<?> c)	This method retains all the elements in the list that are present in the specified collection.
E set(int index, E element)	This method replaces the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	This method sort the elements of the list on the basis of the specified comparator.
Spliterator<E> spliterator()	This method creates spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	This method returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
int size()	This method returns the number of elements present in the list.
void trimToSize()	This method trims the capacity of this ArrayList instance to be the list's current size.

Previous

Next

Queue Interface

Queue Interface

4. Queue Interface in Java

In Java, the Queue Interface is found in java.util package. It extends the Collection interface.

The queue interface follows the FIFO (first-in-first-out) order.
It is an ordered list that is used to store the items that are about to be processed.
For example, When we want to purchase a train ticket, the seats are available on a first-come, first-served basis. As a result, the person who is first in queue receives the ticket first.

The Queue interface is implemented by classes such as PriorityQueue and ArrayDeque.

The Queue interface can be instantiated as follows:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

We will learn more about these classes in the data structure-guided path.

Previous

Next

Set Interface

Set Interface

5. Set Interface in Java

The Set Interface in Java is included in java.util package. It also extends the Collection interface.
It represents an unordered set of elements and does not facilitate the storage of duplicate objects. In Set, we can only store one null value.
The set interface is implemented by classes such as HashSet, LinkedHashSet, and TreeSet, etc.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

We will learn more about these classes in the data structure-guided path.

Previous

Next

Collection Algorithms

Collection Algorithms

Collection Algorithms in Java

Several algorithms are defined as static methods in the Java collection context, and they can be used without knowing how they are implemented.

In Java, the Collections class contains all the collection algorithms which is defined in java.util package.

The following algorithms are available in the collection framework as methods.

Method	Description
void sort(List list)	This method is used to sort the elements of the list as determined by their natural ordering.
void sort(List list, Comparator comp)	This method is used to sort the elements of the list as determined by Comparator comp.
void reverse(List list)	This method is used to reverse all the elements sequence in the list.
void rotate(List list, int n)	This method is used to rotate the list by n places to the right. To rotate left, use a negative value for n.
void shuffle(List list)	This method is used to shuffle the elements in the list.
void shuffle(List list, Random r)	This method is used to shuffle the elements in the list by using r as a source of random numbers.
void copy(List list1, List list2)	This method is used to copy the elements of list2 to list1.
List nCopies(int num, Object obj)	This method is used to return num copies of obj contained in an immutable list. num can not be zero or negative.

void swap(List list, int idx1, int idx2)	This method is used to exchange the elements in the list at the indices specified by idx1 and idx2.
int binarySearch(List list, Object value)	This method is used to return the position of value in the list (must be in the sorted order), or -1 if the value is not found.
int binarySearch(List list, Object value, Comparator c)	This method is used to return the position of value in the list ordered according to c, or -1 if the value is not found.
int indexOfSubList(List list, List subList)	This method is used to return the index of the first match of subList in the list, or -1 if no match is found.
int lastIndexOfSubList(List list, List subList)	This method is used to return the index of the last match of subList in the list, or -1 if no match is found.
Object max(Collection c)	This method is used to return the largest element from the collection c as determined by natural ordering.
Object max(Collection c, Comparator comp)	This method is used to return the largest element from the collection c as determined by Comparator comp.
Object min(Collection c)	This method is used to return the smallest element from the collection c as determined by natural ordering.
Object min(Collection c, Comparator comp)	This method is used to return the smallest element from the collection c as determined by Comparator comp.
void fill(List list, Object obj)	This method is used to assign obj to each element of the list.
boolean replaceAll(List list, Object old, Object new)	This method is used to replace all occurrences of old with new in the list.

Example using few of above methods:

```
import java.util.*;

public class AlgorithmExample {

    public static void main(String args[]) {

        // Declaring an ArrayList of Integer, 'list'.
        ArrayList<Integer> list = new ArrayList<Integer>();

        // Inserting elements in the 'list'.
        list.add(-4);
        list.add(21);
        list.add(-21);
        list.add(4);

        System.out.println("Original list:");
        System.out.println(list);

        // Creating a reverse order comparator.
        Comparator r = Collections.reverseOrder();
```

```
        // Sorting the 'list' using the comparator.
        Collections.sort(list, r);

        // Getting iterator to the 'list'.
        Iterator li = list.iterator();

        System.out.print("List sorted in reverse: ");
        while(li.hasNext()) {
            System.out.print(li.next() + " ");
        }
        System.out.println();


        // Shuffling the 'list'.
        Collections.shuffle(list);

        // Getting iterator to the 'list'.
        li = list.iterator();

        System.out.print("List shuffled: ");
        while(li.hasNext()) {
            System.out.print(li.next() + " ");
        }

        System.out.println();

        // Minimum element in the 'list'.
        System.out.println("Minimum: " + Collections.min(list));

        // Maximum element in the 'list'.
        System.out.println("Maximum: " + Collections.max(list));
    }
}
```

Output:

```
Original list:
[-4, 21, -21, 4]
List sorted in reverse: 21 4 -4 -21
List shuffled: -21 -4 4 21
Minimum: -21
```

```
Maximum: 21
```

Previous

Next

Example Problems

Example Problems

1. What is the output of the following program?

```
import java.util.*;

public class linkedList {
    public static void main(String[] args)
    {
        List<String> list1 = new LinkedList<>();
        list1.add("CN");
        list1.add("For");
        list1.add("CN");
        list1.add("CodingNinjas");
        list1.add("Code");

        List<String> list2 = new LinkedList<>();
        list2.add("CN");

        list1.removeAll(list2);

        for (String temp : list1)
            System.out.printf(temp + " ");

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

1. CN CN
2. CN For CN

3. CodingNinjas
4. For CodingNinjas Code

Answer: Option D

Explanation: list1.removeAll(list2) function deletes all the occurrences of the string in list2 from list1. Here, the string “CN” appears in list2, so all the nodes of the linked list in list1 that contains “CN” as its data are removed from list1.

2. What will be the output of the following Java program?

```
import java.util.*;
class Arraylist
{
    public static void main(String args[])
    {
        ArrayList obj = new ArrayList();
        obj.add("A");
        obj.add("B");
        obj.add("C");
        obj.add(1, "D");
        System.out.println(obj);
    }
}
```

A. [A, B, C, D]
B. [A, D, B, C]
C. [A, D, C]
D. [A, B, C]

Answer: Option B

Explanation: obj is an object of class ArrayList hence it is a dynamic array that can increase and decrease its size. obj.add(“X”) adds to the array element X and obj.add(1, ”X”) adds element x at index position 1 in the list, Hence obj.add(1, ” D”) stores D at index position 1 of obj and shifts the previous value stored at that position by 1.

Previous
Next