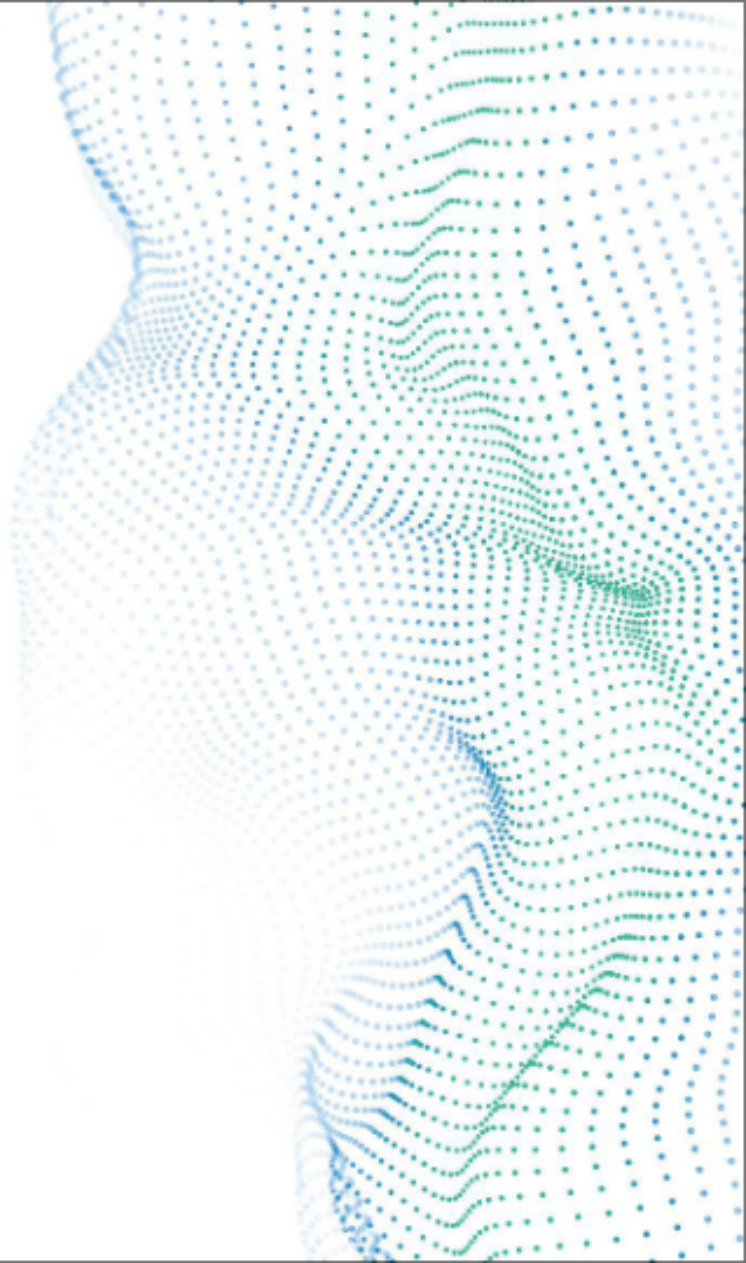
CORE JAVA



Control Flow and Decision Making

Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

Decision Making statements

if statements

switch statement

Loop statements

do while loop

while loop

for loop

for-each loop

Jump statements

- break statement
- continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

Simple if statement

if-else statement

if-else-if ladder

Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

x + y is greater than 20

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Nested if-statement

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java

Cases cannot be duplicate

Default statement is executed when any of the case doesn't match the value of expression. It is optional.

Break statement terminates the switch block when the condition is satisfied.

It is optional, if not used, next case is executed.

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
    case value1:
     statement1;
     break;
    .
    .
    .
    case valueN:
     statementN;
     break;
    default:
     default statement;
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {

//block of statements

}
```

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

The sum of first 10 natural numbers is 55

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}    } }
```

Output:

```
Printing the content of the array names:
Java
C
C++
Python
JavaScript
```

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3
```

Calculation .java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

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

Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Printing the list of first 10 even numbers

0

2

4

6

8

10

Java Collection Framework

Hierarchy of Collection Framework

Collection interface

Iterator interface

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java?

It provides readymade architecture.

It represents a set of classes and interfaces.

It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

Interfaces and its implementations, i.e., classes

Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for the Collection framework.

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

Iterator<T> iterator()

Collection Interface

The Collection interface is the interface which 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.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
```

Output:

Ravi
Vijay
Ravi
Ajay
Vector
Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
```

```
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

Variables, Datatypes, Operators

Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
public class A
{
   static int m=100;//static variable
   void method()
   {
      int n=90;//local variable
   }
   public static void main(String args[])
   {
      int data=50;//instance variable
   }
}//end of class
```

Java Variable Example: Add Two Numbers

```
public class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}
}
```

Output:

20

Java Variable Example: Widening

```
public class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Java Variable Example: Add Two Numbers

```
.public class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}
}
```

Java Variable Example: Widening

```
public class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Datatypes

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

boolean data type

byte data type

char data type

short data type

int data type

long data type

float data type

double data type

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
Long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, **byte** b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

short s = 10000, **short** r = -5000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3
```

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

```
char letterA = 'A'
```

Operators

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

The Unary Operator,

Arithmetic Operator,

Shift Operator,

Relational Operator,

Bitwise Operator,

Logical Operator,

Ternary Operator

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
prefix		++expr --expr +expr -expr ~ !
Arithmetic	multiplicative	* / %
additive	+ -	
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
equality	== !=	
Bitwise	bitwise AND	&
bitwise exclusive	OR	^
bitwise inclusive	OR	|
Logical	logical AND	&&
logical OR		||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java AND Operator Example: Logical && vs Bitwise &

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

false

10

false

11

Output:

true

true

true

10

true

11

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Arrays and Strings

An Array is an essential and most used data structure in Java. It is one of the most used data structure by programmers due to its efficient and productive nature; The Array is a collection of similar data type elements. It uses a contiguous memory location to store the elements.

A String Array is an Array of a fixed number of String values. A String is a sequence of characters. Generally, a string is an immutable object, which means the value of the string can not be changed. The String Array works similarly to other data types of Array.

In Array, only a fixed set of elements can be stored. It is an index-based data structure, which starts from the 0th position. The first element will take place in Index 0, and the 2nd element will take place in Index 1, and so on.

The main method {Public static void main[String [] args]; } in Java is also an String Array.

It is an object of the Array.

It can be declared by the two methods; by specifying the size or without specifying the size.

It can be initialized either at the time of declaration or by populating the values after the declaration.

The elements can be added to a String Array after declaring it.

The String Array can be iterated using the for loop.

The searching and sorting operation can be performed on the String Array.

Declaration:

The Array declaration is of two types, either we can specify the size of the Array or without specifying the size of the Array. A String Array can be declared as follows:

String[] stringArray1 //Declaration of the String Array without specifying the size

String[] stringArray2 = new String[2]; //Declararion by specifying the size

Another way of declaring the Array is String strArray[], but the above-specified methods are more efficient and recommended.

Initialization:

The String Array can be initialized easily. Below is the initialization of the String Array:

```
1. String[] strAr1=new String[] {"Ani", "Sam", "Joe"}; //inline initialization
2. String[] strAr2 = {"Ani", "Sam", " Joe"};
3. String[] strAr3= new String[3]; //Initialization after declaration with specific size
   strAr3[0]= "Ani";
   strAr3[1]= "Sam";
   strAr3[2]= "Joe";
```

The 3rd method is a specific size method. In this, the value of the index can be found using the (**arraylength - 1**) formula if we want to access the elements more than the index 2 in the above Array. It will throw the **Java.lang.ArrayIndexOutOfBoundsException** exception.
Let's see an example of String Array to demonstrate it's behavior:
Iteration of String Array
The String Array can be iterated using the for and foreach loop. Consider the below code:

```
String[] strAr = {"Ani", "Sam", "Joe"};
for (int i=0; i<StrAr.length; i++)
{
System.out.println(strAr[i]);
}
for ( String str: strAr)
{
Sytem.out.println(str);
}
```

Adding Elements to a String Array

We can easily add the elements to the String Array just like other data types. It can be done using the following three methods:

Using Pre-Allocation of the Array
Using the Array List
By creating a new Array

let's understand the above methods:
Using Pre-Allocation of the Array:
In this method, we already have an Array of larger size. For example, if we require to store the 10 elements, then we will create an Array of size 20. It is the easiest way to expand the Array elements.

Consider the below example to add elements in a pre-allocated array.

```
// Java Program to add elements in a pre-allocated Array
import java.util.Arrays;
public class StringArrayDemo {
        public static void main(String[] args) {
            String[] sa = new String[7]; // Creating a new Array of Size 7
            sa[0] = "A"; // Adding Array elements
            sa[1] = "B";
            sa[2] = "C";
            sa[3] = "D";
            sa[4] = "E";
            System.out.println("Original Array Elements:" + Arrays.toString(sa));
            int numberOfItems = 5;
            String newItem = "F"; // Expanding Array Elements Later
            String newItem2 ="G";
            sa[numberOfItems++] = newItem;
            sa[numberOfItems++] = newItem2;
            System.out.println("Array after adding two elements:" +
                    Arrays.toString(sa));
        }
    }
```

Output:

Original Array Elements:[A, B, C, D, E, null, null]
Array after adding two elements:[A, B, C, D, E, F, G]

Using ArrayList:
The ArrayList is a fascinating data structure of the Java collection framework. We can easily add elements to a String Array using an ArrayList as an intermediate data structure.

Consider the below example to understand how to add elements to a String Array using ArrayList:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
public class StringArrayDemo1 {
    public static void main(String[] args)
    {
        // Defining a String Array
        String sa[] = { "A", "B", "C", "D", "E", "F" };
      //
        System.out.println("Initial Array:\n"
                           + Arrays.toString(sa));
        String ne = "G"; // Define new element to add
        List<String>l = new ArrayList<String>(
                Arrays.asList(sa)); // Convert Array to ArrayList
        l.add(ne); // Add new element in ArrayList l
        sa = l.toArray(sa); // Revert Conversion from ArrayList to Array
```

```
// printing the new Array
        System.out.println("Array with added Value: \n"
                + Arrays.toString(sa)) ;
    }
  }
```

Output:

Initial Array:

[A, B, C, D, E, F]

Array with added value:

[A, B, C, D, E, F, G]

By Creating a New Array:

In this method, we will create a new Array with a larger size than the initial Array and accommodate the elements in it. We will copy all the elements to the newly added Array.

Consider the below example:

```
// Java Program to add elements in a String Array by creating a new Array
import java.util.Arrays;
public class StringArrayDemo2 {
    public static void main(String[] args) {
        //Declaring Initial Array
        String[] sa = {"A", "B", "C" };
        // Printing the Original Array
        System.out.println("Initial Array: " + Arrays.toString(sa));
```

```
int length_Var = sa.length; //Defining the array length variable
        String newElement = "D"; // Defining new element to add
        //define new array with extended length
        String[] newArray = new String[ length_Var + 1 ];
        //Adding all the elements to initial Array
        for (int i=0; i <sa.length; i++)
        {
            newArray[i] = sa [i];
         }
        //Specifying the position of the added elements ( Last)
        newArray[newArray.length- 1] = newElement;
        //make it original and print
        sa = newArray;
        System.out.println("updated Array: " + Arrays.toString(sa));
    } }
```

Output:

Initial Array: [A, B, C]

updated Array: [A, B, C, D]

This is how we can add elements to a String Array. Let's understand how to search and sort elements in String Array.

Searching in String Array

For searching a String from the String Array, for loop is used. Consider the below example:

```
public class StringArrayExample {
    public static void main(String[] args) {
        String[] strArray = { "Ani", "Sam", "Joe" };
        boolean x = false; //initializing x to false
        int in = 0; //declaration of index variable
        String s = "Sam";  // String to be searched
        // Iteration of the String Array
        for (int i = 0; i < strArray.length; i++) {
            if(s.equals(strArray[i])) {
                in = i; x = true; break;
            }
        }
```

```
 if(x)
                System.out.println(s +" String is found at index "+in);
            else
                System.out.println(s +" String is not found in the array");
        }
}
```

Output:

Initial Array: [A, B, C]

updated Array: [A, B, C, D]

This is how we can add elements to a String Array. Let's understand how to search and sort elements in String Array.

Searching in String Array

For searching a String from the String Array, for loop is used. Consider the below example:

```
public class StringArrayExample {
        public static void main(String[] args) {
            String[] strArray = { "Ani", "Sam", "Joe" };
            boolean x = false; //initializing x to false
            int in = 0; //declaration of index variable
            String s = "Sam";  // String to be searched
            // Iteration of the String Array
            for (int i = 0; i < strArray.length; i++) {
                if(s.equals(strArray[i])) {
                    in = i; x = true; break;
                }
            }
```

```
if(x)
            System.out.println(s +" String is found at index "+in);
        else
            System.out.println(s +" String is not found in the array");
    }
}
```

Output:

Sam String is found at index 1

In the above example, we have initialized a boolean variable x to false and an index variable to iterate through the string. Also, we have declared a local variable String variable s to be searched. Here, the break keyword will exit the loop as soon as the string is found.

Sorting in String Array

The sorting in the String array is quite easy. It is performed like in a traditional array. We use a sort() method to sort the Array elements. Sorting is easier than searching.

Consider the below example to sort a String Array:

```
//Java Program to sort elements in a String Array

import java.util.Arrays;

public class StringArraySorting {

    public static void main(String[] args)

    {

        // Adding String values

        String[] colors = {"Cricket","Basketball","Football","Badminton","Tennis"};
```

```
// Print Original values
        System.out.println("Entered Sports: "+Arrays.toString(colors));
         Arrays.sort(colors); // Sorting Elements
        // Print Sorted Values
         System.out.println("Sorted Sports: "+Arrays.toString(colors));
    }
  }
```

Output:

Entered Sports: [Cricket, Basketball, Football, Badminton, Tennis]

Sorted Sports: [Badminton, Basketball, Cricket, Football, Tennis]

From the above example, we can see the elements from a String Array is sorted using the sort() method.

We can also convert String Array to other data structures such as List, int Array, ArrayList, and more and vice-versa.

Oops Concepts 2

Method Overloading in Java

Method Overloading in Java

Different ways to overload the method

By changing the no. of arguments

By changing the datatype

Why method overloading is not possible by changing the return type

Can we overload the main method

method overloading with Type Promotion

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments
In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

output
22
24.9

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{tatic int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Output:

Compile Time Error: method add(int,int) is already defined
in class Adder

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}
```

```
public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}
```

Example of Method Overloading with Type Promotion if matching found
If there are matching type arguments in the method, type promotion is not performed.

```
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

Output:int arg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity
If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

Method Overriding in Java

Understanding the problem without method overriding

Can we override the static method

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

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

Usage of Java Method Overriding

Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class.

There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
/Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
```

```
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

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

Exception Handling in Java

What is Exception?

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

FILE I/0 OPERATIONS

Java I/O (Input and Output) is used to process the input and produce the output.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform file handling in Java by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print output and an error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Methods and Functions

Method in Java
In general, a method is a way to perform some task. Similarly, the method in Java is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using methods. In this section, we will learn what is a method in Java, types of methods, method declaration, and how to call a method in Java.

What is a method in Java?
A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.
The most important method in Java is the **main()** method. If you want to read more about the main() method, go through the link https://www.javatpoint.com/java-main-method.

```
public class Display
{
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

InstanceMethodExample.java

Output:

It is an example of a static method.

Instance Method

The method of the class is known as an instance method. It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

```
public class InstanceMethodExample

{

public static void main(String [] args)

{
```

```
//Creating an object of the class
InstanceMethodExample obj = new InstanceMethodExample();
//invoking instance method
System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-defined method because we have not used static keyword
public int add(int a, int b)
{
s = a+b;
//returning the sum
return s;
}
}
```

Output:

The sum is: 25

There are two types of instance method:

Accessor Method

Mutator Method

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word get. It is also known as getters. It returns the value of the private field. It is used to get the value of the private field.

Types of Variables

There are three types of variables in Java:

local variable

instance variable

static variable

Example

```
public int getId()
{
return Id;
}
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word set. It is also known as setters or modifiers. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
public void setRoll(int roll)
{
this.roll = roll;
}
```

Generics and Enumerations

Generics in Java

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

1)Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error
  String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

Ravi
Vijay
Ravi
Ajay
Vector
Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();
```

Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
import java.util.*;

class TestGenerics2{

public static void main(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();

map.put(1,"vijay");

map.put(4,"umesh");

map.put(2,"ankit");


//Now use Map.Entry for Set and Iterator
```

```
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}

}}
```

Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

Creating a generic class:

```
class MyGen<T>{

T obj;

void add(T obj){this.obj=obj;}

T get(){return obj;}

}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

Output

2

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

T- Type

E - Element

K - Key

N - Number

V - Value

Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{


  public static < E > void printArray(E[] elements) {
       for ( E element : elements){
          System.out.println(element );
        }
        System.out.println();
   }
  public static void main( String args[] ) {
       Integer[] intArray = { 10, 20, 30, 40, 50 };
       Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
```

Output

Printing Integer Array

10

20

30

40

50

Printing Character Array

J

A

V

A

T

P

O

I

N

T

Java Enums

The Enum in Java is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Multi Threading and Concurrency

Multithreading in Java

Multithreading

Multitasking

Process-based multitasking

Thread-based multitasking

What is Thread

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Java Thread class

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

Following are the stages of the life cycle −

New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic

steps −

Step 1

As a first step, you need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method −

```
public void run( )
```

Step 2

As a second step, you will instantiate a Thread object using the following constructor −

```
Thread(Runnable threadObj, String threadName);
```

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method −

```
void start();
```

Example

Here is an example that creates a new thread and starts running it −

Live Demo

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;
```

```
System.out.println("Thread: " + threadName + ", " + i);


            // Let the thread sleep for a while.
            Thread.sleep(50);
        }
    } catch (InterruptedException e) {
        System.out.println("Thread " +  threadName + " interrupted.");
    }
    System.out.println("Thread " +  threadName + " exiting.");
}
```

```
 public void start () {
      System.out.println("Starting " +  threadName );

      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```
public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo("Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo("Thread-2");
      R2.start();
   }
}
```

END MODULE-8 CORE JAVA