Java Programming: Data types, Operators and expressions, Input/Output mechanism, Control statements, Classes, Objects and methods, Final Variables, Static variables, Static methods, Instance methods, Constructors, Types of constructors, Constructor Overloading, method overloading, Access qualifiers, Array.

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:

1. Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.
2. Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

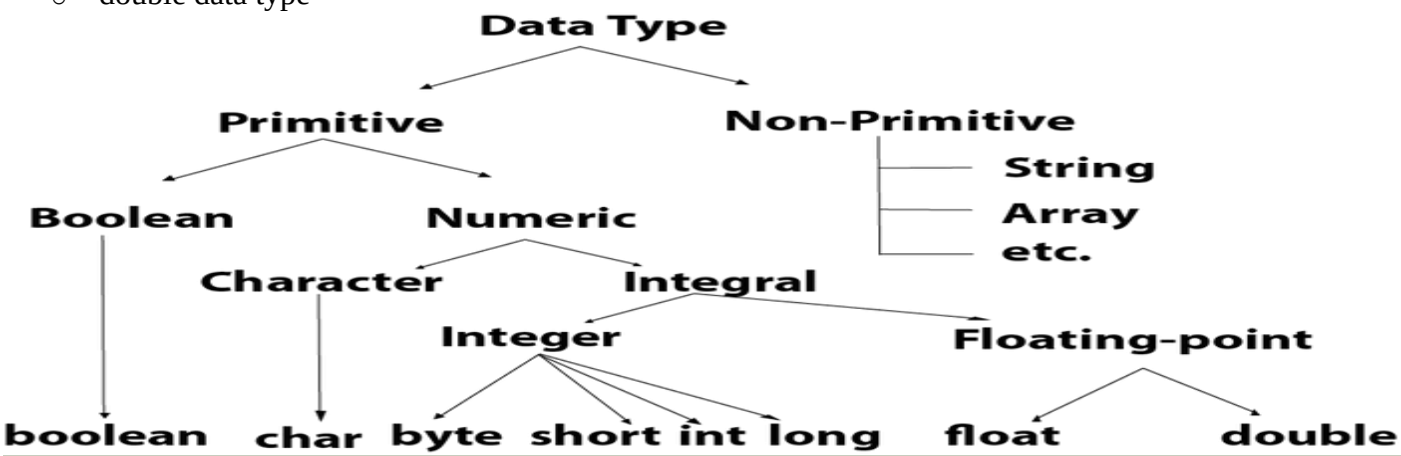
Java Primitive Data Types

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

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

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:

```
1. 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:

```
1. 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:

```
1. short s = 10000, short r = -5000
```

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

```
1. int a = 100000, int b = -200000
```

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:

```
1. 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:

```
1. float f1 = 234.5f
```

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:

```
1. 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:

```
1. char letterA = 'A'
```

Operators in Java

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

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

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 Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=10;
```

```
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
7.
8. }}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=-10;
5. boolean c=true;
6. boolean d=false;
7. System.out.println(~a);//-11 (minus of total positive value which starts from 0)
8. System.out.println(~b);//9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);//false (opposite of boolean value)
10. System.out.println(!d);//true
11. }}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}
```

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10*10/5+3-1*4/2);
4. }}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10<<2);//10*2^2=10*4=40
4. System.out.println(10<<3);//10*2^3=10*8=80
5. System.out.println(20<<2);//20*2^2=20*4=80
6. System.out.println(15<<4);//15*2^4=15*16=240
7. }}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
1. public OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
1. public class OperatorExample{
2. public static void main(String args[]){
3.     //For positive number, >> and >>> works same
4.     System.out.println(20>>2);
5.     System.out.println(20>>>2);
6.     //For negative number, >>> changes parity bit (MSB) to 0
7.     System.out.println(-20>>2);
8.     System.out.println(-20>>>2);
9. }}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

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

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}
```

Output:

```
true
true
true
10
true
11
```

Java Ternary Operator

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

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

```
2
```

Another Example:

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

```
5
```

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
1. public class OperatorExample{
```

```
2. public static void main(String[] args){
3. int a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a*=2;//9*2
9. System.out.println(a);
10. a/=2;//18/2
11. System.out.println(a);
12. }}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. short a=10;
4. short b=10;
5. //a+=b;//a=a+b internally so fine
6. a=a+b;//Compile time error because 10+10=20 now int
7. System.out.println(a);
8. }}
```

Output:

```
Compile time error
```

After type cast:

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. short a=10;
4. short b=10;
5. a=(short)(a+b);//20 which is int now converted to short
6. System.out.println(a);
7. }}
```

Output:

```
20
```

Java Control Statements | 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.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements

- do while loop
- while loop
- for loop
- for-each loop

3. 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.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. 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.

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
```

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

Student.java

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. int x = 10;
4. int y = 12;
5. if(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }
```

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:

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
4. else{
5. statement 2; //executes when condition is false
6. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. int x = 10;
4. int y = 12;
5. if(x+y < 10) {
6. System.out.println("x + y is less than      10");
7. }   else {
8. System.out.println("x + y is greater than 20");
9. }
10. }
11. }
```

Output:

```
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.

```
1. if(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. else if(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. else {
8. statement 2; //executes when all the conditions are false
9. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. String city = "Delhi";
4. if(city == "Meerut") {
5. System.out.println("city is meerut");
```

```
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

```
Delhi
```

4. 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
}
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

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.

Points to be noted about switch statement:

- 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.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

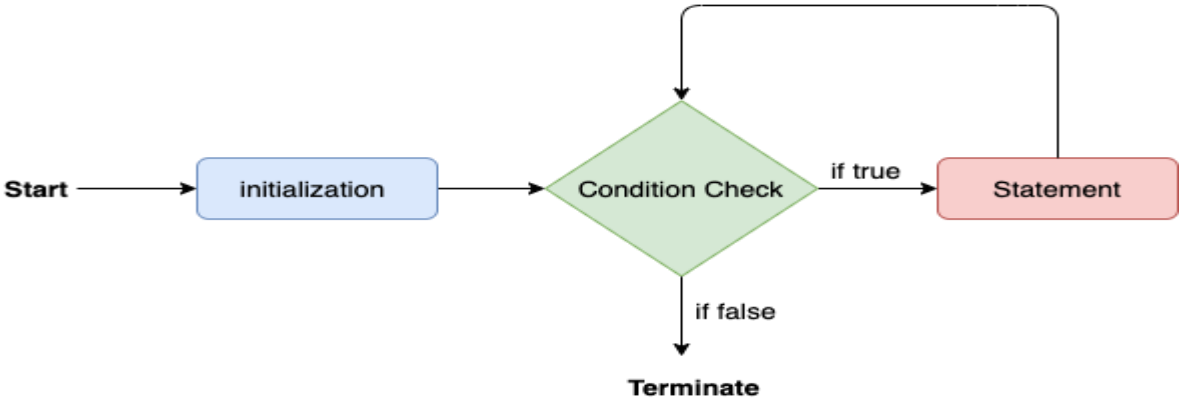
Let's understand the loop statements one by one.

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
}
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
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.

```
1. for(data_type var : array_name/collection_name){
2. //statements
3. }
```

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

Calculation.java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. for(String name:names) {
7. System.out.println(name);
8. }
9. }
10. }
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

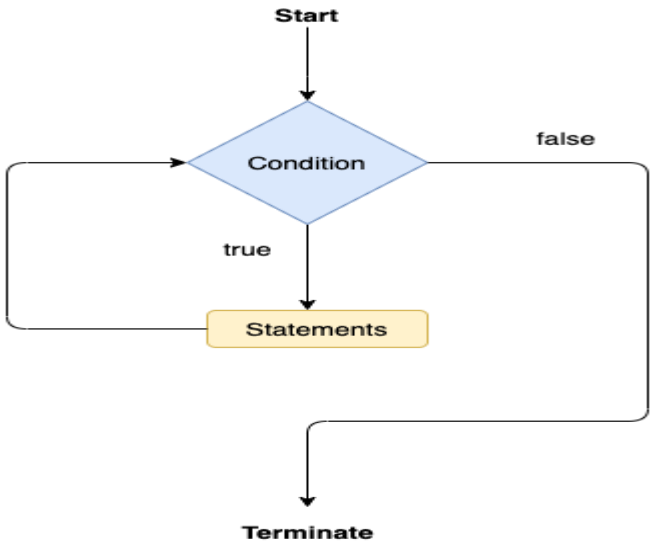
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
1. while(condition){
2. //looping statements
3. }
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. while(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10. }
11. }
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

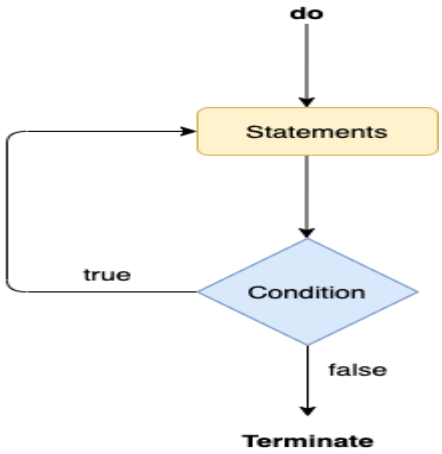
Java do-while loop

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.

```
1. do
2. {
3. //statements
4. } while (condition);
```

The flow chart of the do-while loop is given in the following image.



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

Calculation.java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. do {
7. System.out.println(i);
8. i = i + 2;
9. }while(i<=10);
10. }
11. }
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
1. public class BreakExample {
2.
3. public static void main(String[] args) {
4. // TODO Auto-generated method stub
5. for(int i = 0; i<= 10; i++) {
6. System.out.println(i);
7. if(i==6) {
8. break;
9. }
10. }
11. }
12. }
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

Calculation.java

```
1. public class Calculation {
2.
3. public static void main(String[] args) {
4. // TODO Auto-generated method stub
5. a:
6. for(int i = 0; i<= 10; i++) {
7. b:
8. for(int j = 0; j<=15;j++) {
9. c:
10. for (int k = 0; k<=20; k++) {
11. System.out.println(k);
12. if(k==5) {
13. break a;
14. }
15. }
16. }
17.
18. }
19. }
20.
21.
22. }
```

Output:

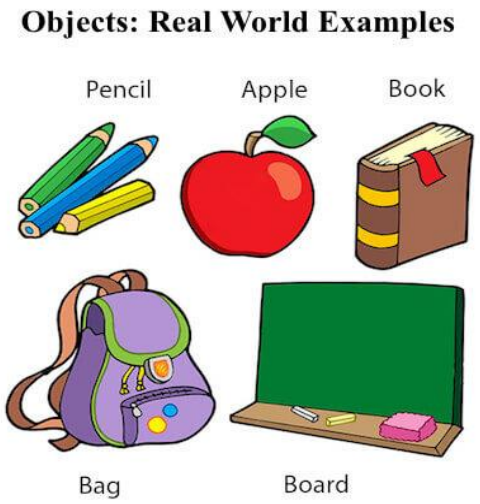
```
0
1
2
3
4
5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
1.  public class ContinueExample {
2.
3.  public static void main(String[] args) {
4.  // TODO Auto-generated method stub
5.
6.  for(int i = 0; i<= 2; i++) {
7.
8.  for (int j = i; j<=5; j++) {
9.
10.         if(j == 4) {
11.         continue;
12.         }
13.         System.out.println(j);
14.         }
15.         }
16.         }
17.
18.         }
```



Output:

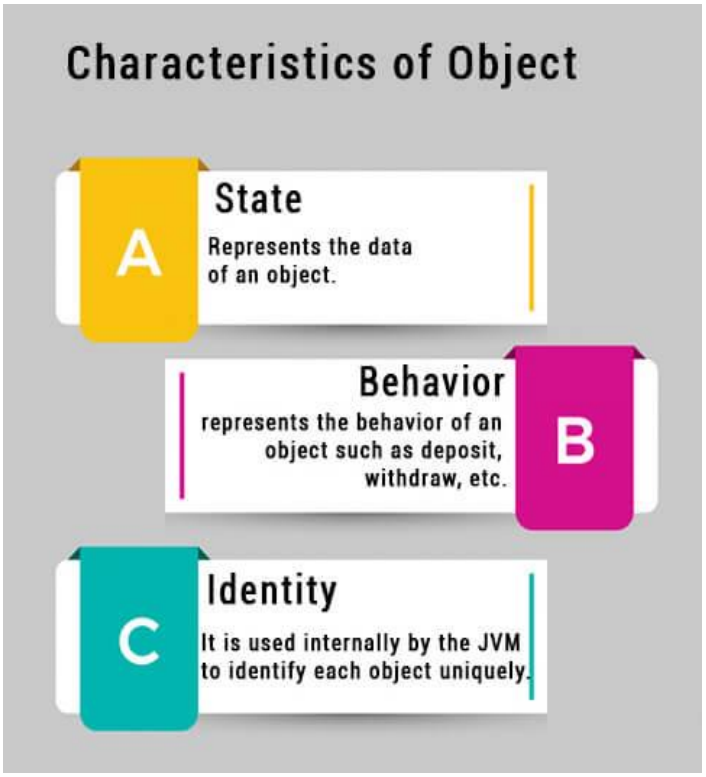
```
0
1
2
3
5
1
2
3
5
2
3
5
```

What is an object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- State: represents the data (value) of an object.
- Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

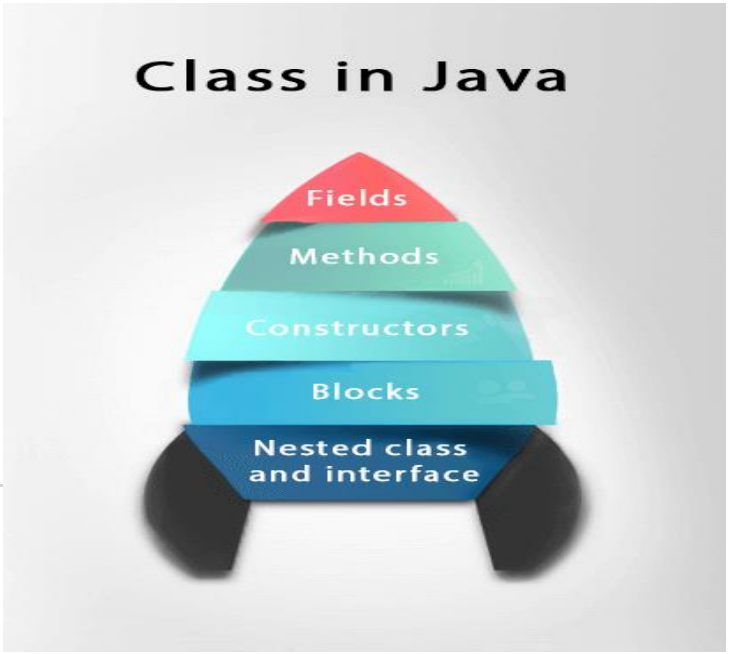


For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.



What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
1. class <class_name>{
2.     field;
3.     method;
4. }
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

---

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. class Student{
4.  //defining fields
5.  int id;//field or data member or instance variable
6.  String name;
7.  //creating main method inside the Student class
8.  public static void main(String args[]){
9.   //Creating an object or instance
10.  Student s1=new Student();//creating an object of Student
11.  //Printing values of the object
12.  System.out.println(s1.id);//accessing member through reference variable
13.  System.out.println(s1.name);
14. }
15. }
```

Test it Now

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
1. //Java Program to demonstrate having the main method in
2. //another class
3. //Creating Student class.
4. class Student{
5.  int id;
6.  String name;
```

```
7.  }
8.  //Creating another class TestStudent1 which contains the main method
9.  class TestStudent1{
10.  public static void main(String args[]){
11.   Student s1=new Student();
12.   System.out.println(s1.id);
13.   System.out.println(s1.name);
14.  }
15. }
```

Test it Now

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1.  class Student{
2.   int id;
3.   String name;
4.  }
5.  class TestStudent2{
6.   public static void main(String args[]){
7.    Student s1=new Student();
8.    s1.id=101;
9.    s1.name="Sonoo";
10.   System.out.println(s1.id+" "+s1.name);//printing members with a white space
11.  }
12. }
```

Test it Now

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
1.  class Student{
2.   int id;
3.   String name;
4.  }
5.  class TestStudent3{
6.   public static void main(String args[]){
7.    //Creating objects
```

```
8.    Student s1=new Student();
9.    Student s2=new Student();
10.   //Initializing objects
11.   s1.id=101;
12.   s1.name="Sonoo";
13.   s2.id=102;
14.   s2.name="Amit";
15.   //Printing data
16.   System.out.println(s1.id+" "+s1.name);
17.   System.out.println(s2.id+" "+s2.name);
18.  }
19. }
```

Test it Now

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

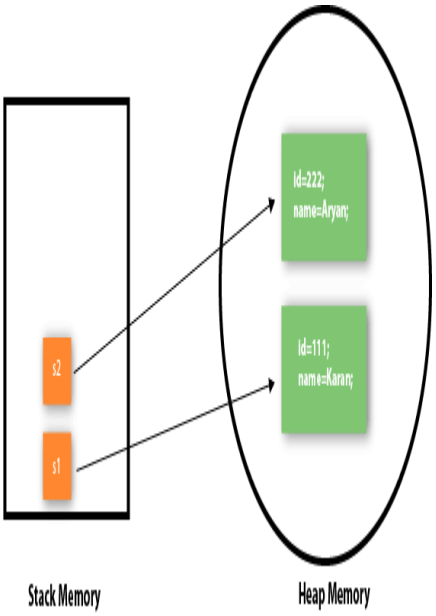
```
1.  class Student{
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.    rollno=r;
6.    name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9.  }
10. class TestStudent4{
11.  public static void main(String args[]){
12.   Student s1=new Student();
13.   Student s2=new Student();
14.   s1.insertRecord(111,"Karan");
15.   s2.insertRecord(222,"Aryan");
16.   s1.displayInformation();
17.   s2.displayInformation();
18.  }
19. }
```

Test it Now

Output:

```
111 Karan
```

```
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
1. class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {
6.         id=i;
7.         name=n;
8.         salary=s;
9.     }
10.    void display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.    Employee e1=new Employee();
15.    Employee e2=new Employee();
16.    Employee e3=new Employee();
17.    e1.insert(101,"ajeet",45000);
18.    e2.insert(102,"irfan",25000);
19.    e3.insert(103,"nakul",55000);
20.    e1.display();
21.    e2.display();
22.    e3.display();
23. }
24. }
```

Test it Now

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
1.  class Rectangle{
2.   int length;
3.   int width;
```

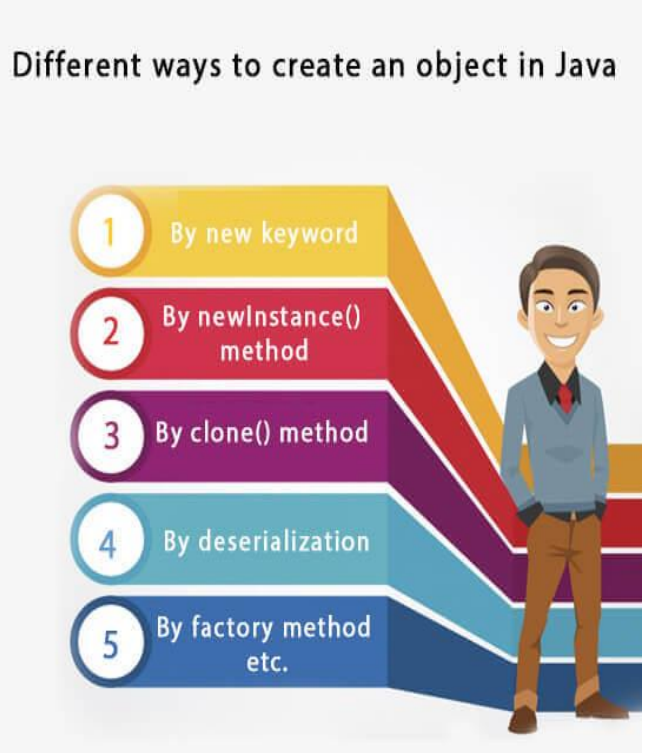
```
4.   void insert(int l, int w){
5.    length=l;
6.    width=w;
7.   }
8.   void calculateArea(){System.out.println(length*width);}
9.  }
10. class TestRectangle1{
11.  public static void main(String args[]){
12.   Rectangle r1=new Rectangle();
13.   Rectangle r2=new Rectangle();
14.   r1.insert(11,5);
15.   r2.insert(3,15);
16.   r1.calculateArea();
17.   r2.calculateArea();
18. }
19. }
```

Test it Now

Output:

```
55
45
```

What are the different ways to create an object in Java?



There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
1. new Calculation();//anonymous object
```

Calling method through a reference:

```
1. Calculation c=new Calculation();
2. c.fact(5);
```

Calling method through an anonymous object

```
1. new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
1. class Calculation{
2.  void fact(int  n){
```

```
    int fact=1;
    for(int i=1;i<=n;i++){
     fact=fact*i;
    }
   System.out.println("factorial is "+fact);
  }
  public static void main(String args[]){
   new Calculation().fact(5);//calling method with anonymous object
 }
}
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of refernce variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
 public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Test it Now

Output:

```
55
45
```

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
```

```
class Account{
int acc_no;
String name;
float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}
```

Test it Now

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Final Keyword In Java

1. Final variable
2. Final method

3. Final class
4. Is final method inherited ?
5. Blank final variable
6. Static blank final variable
7. Final parameter
8. Can you declare a final constructor

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
7.  Bike9 obj=new  Bike9();
8.  obj.run();
9.  }
10. }//end of class
```

Test it Now

Output:Compile Time Error

---

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
1. class Bike{
2.   final void run(){System.out.println("running");}
3. }
4.
5. class Honda extends Bike{
6.    void run(){System.out.println("running safely with 100kmph");}
```

```
7.
8.    public static void main(String args[]){
9.    Honda honda= new Honda();
10.   honda.run();
11.   }
12. }
```

Test it Now

Output:Compile Time Error

---

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
1.  final class Bike{}
2.
3.  class Honda1 extends Bike{
4.    void run(){System.out.println("running safely with 100kmph");}
5.
6.    public static void main(String args[]){
7.    Honda1 honda= new Honda1();
8.    honda.run();
9.    }
10. }
```

Test it Now

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1.  class Bike{
2.    final void run(){System.out.println("running...");}
3.  }
4.  class Honda2 extends Bike{
5.    public static void main(String args[]){
6.     new Honda2().run();
7.    }
8.  }
```

Test it Now

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
1.  class Student{
2.  int id;
3.  String name;
4.  final String PAN_CARD_NUMBER;
5.  ...
```

```
6.  }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1.  class Bike10{
2.    final int speedlimit;//blank final variable
3.
4.    Bike10(){
5.    speedlimit=70;
6.    System.out.println(speedlimit);
7.    }
8.
9.    public static void main(String args[]){
10.     new Bike10();
11.  }
12. }
```

Test it Now

Output: 70

---

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1.  class A{
2.    static final int data;//static blank final variable
3.    static{ data=50;}
4.    public static void main(String args[]){
5.      System.out.println(A.data);
6.   }
7.  }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1.  class Bike11{
2.    int cube(final int n){
3.     n=n+2;//can't be changed as n is final
4.     n*n*n;
5.    }
6.    public static void main(String args[]){
7.      Bike11 b=new Bike11();
8.      b.cube(5);
9.   }
10. }
```

Test it Now

Output: Compile Time Error

Java static keyword

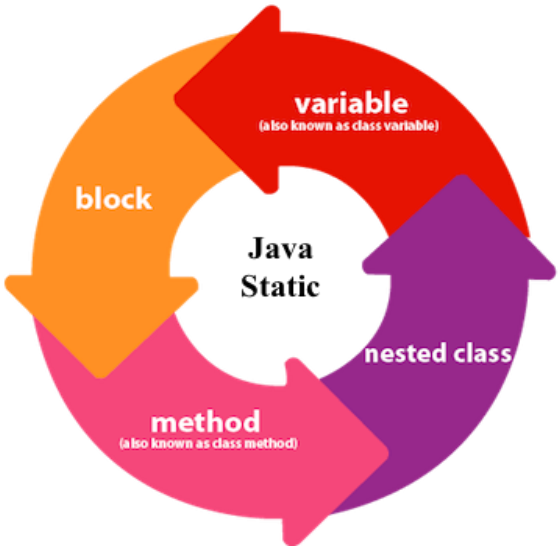
1. Static variable
2. Program of the counter without static variable
3. Program of the counter with static variable
4. Static method
5. Restrictions for the static method

6. Why is the main method static?
7. Static block
8. Can we execute a program without main method?

The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program memory efficient (i.e., it saves memory).

Understanding the problem without static variable

```
1. class Student{
2.     int rollno;
3.     String name;
4.     String college="ITS";
5. }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Example of static variable

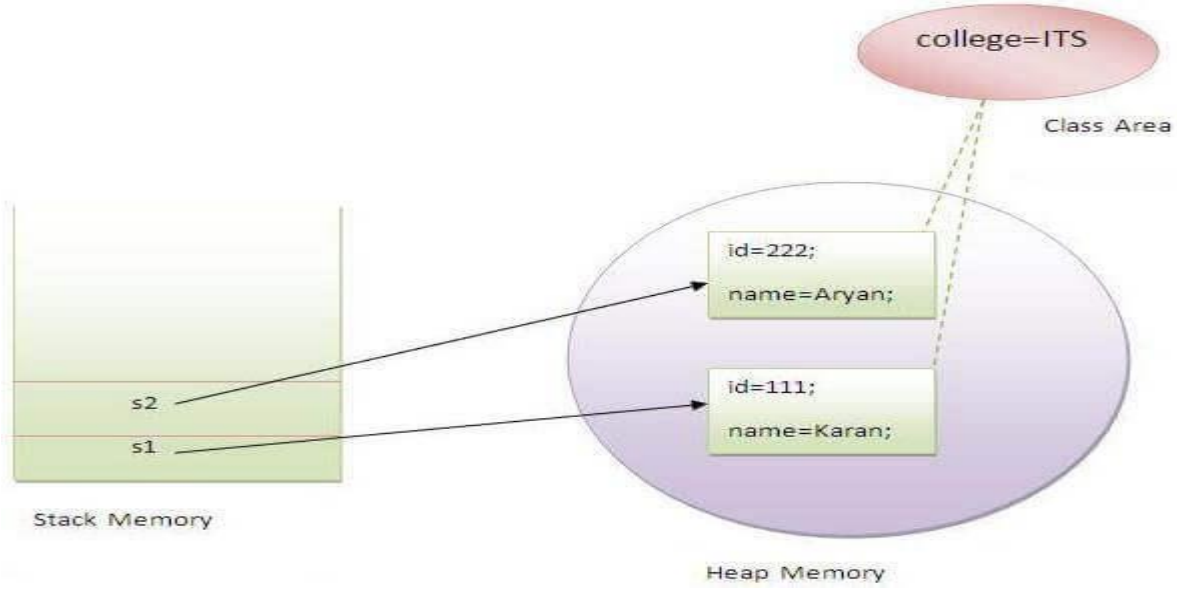
```
1. //Java Program to demonstrate the use of static variable
2. class Student{
3.   int rollno;//instance variable
4.   String name;
5.   static String college ="ITS";//static variable
6.   //constructor
7.   Student(int r, String n){
8.   rollno = r;
9.   name = n;
10.  }
11.  //method to display the values
12.  void display (){System.out.println(rollno+" "+name+" "+college);}
```

```
13. }
14. //Test class to show the values of objects
15. public class TestStaticVariable1{
16.  public static void main(String args[]){
17.  Student s1 = new Student(111,"Karan");
18.  Student s2 = new Student(222,"Aryan");
19.  //we can change the college of all objects by the single line of code
20.  //Student.college="BBDIT";
21.  s1.display();
22.  s2.display();
23.  }
24. }
```

Test it Now

Output:

```
111 Karan ITS
222 Aryan ITS
```



Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
1.  //Java Program to demonstrate the use of an instance variable
2.  //which get memory each time when we create an object of the class.
3.  class Counter{
4.  int count=0;//will get memory each time when the instance is created
5.
6.  Counter(){
7.  count++;//incrementing value
8.  System.out.println(count);
9.  }
10.
11. public static void main(String args[]){
12. //Creating objects
13. Counter c1=new Counter();
14. Counter c2=new Counter();
15. Counter c3=new Counter();
```

```
16. }
17. }
```
Test it Now

Output:
```
1
1
1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1.  //Java Program to illustrate the use of static variable which
2.  //is shared with all objects.
3.  class Counter2{
4.  static int count=0;//will get memory only once and retain its value
5.
6.  Counter2(){
7.  count++;//incrementing the value of static variable
8.  System.out.println(count);
9.  }
10.
11. public static void main(String args[]){
12. //creating objects
13. Counter2 c1=new Counter2();
14. Counter2 c2=new Counter2();
15. Counter2 c3=new Counter2();
16. }
17. }
```
Test it Now

Output:
```
1
2
3
```

---

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
1.  //Java Program to demonstrate the use of a static method.
2.  class Student{
3.      int rollno;
4.      String name;
5.      static String college = "ITS";
6.      //static method to change the value of static variable
7.      static void change(){
8.      college = "BBDIT";
9.      }
```

```
10.     //constructor to initialize the variable
11.     Student(int r, String n){
12.     rollno = r;
13.     name = n;
14.     }
15.     //method to display values
16.     void display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. public class TestStaticMethod{
20.    public static void main(String args[]){
21.    Student.change();//calling change method
22.    //creating objects
23.    Student s1 = new Student(111,"Karan");
24.    Student s2 = new Student(222,"Aryan");
25.    Student s3 = new Student(333,"Sonoo");
26.    //calling display method
27.    s1.display();
28.    s2.display();
29.    s3.display();
30.    }
31. }
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
1.  //Java Program to get the cube of a given number using the static method
2.
3.  class Calculate{
4.    static int cube(int x){
5.    return x*x*x;
6.    }
7.
8.    public static void main(String args[]){
9.    int result=Calculate.cube(5);
10.   System.out.println(result);
11.   }
12. }
```

Test it Now

```
Output:125
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1.  class A{
2.   int a=40;//non static
3.
4.   public static void main(String args[]){
5.    System.out.println(a);
6.   }
7.  }
```

Test it Now

```
Output:Compile Time Error
```

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

---

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
1. class A2{
2.   static{System.out.println("static block is invoked");}
3.   public static void main(String args[]){
4.    System.out.println("Hello main");
5.   }
6. }
```

Test it Now

```
Output:static block is invoked
       Hello main
```

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

```
1. class A3{
2.   static{
3.   System.out.println("static block is invoked");
4.   System.exit(0);
5.   }
6. }
```

Test it Now

Output:

```
static block is invoked
```

Since JDK 1.7 and above, output would be:

```
Error: Main method not found in class A3, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Instance method

Instance methods are the methods defined under a class and we can call such functions only after creating an object of that class. In fact, the call to the instance method is made through the created object itself.

An instance method has generally the following structure:

```
public return_type myMethod(parameter(s))
{
        // body

}
```

Instance methods generally reside in the permanent generation space section of the heap but the local variables and parameters have a separate location in the stack memory. We can invoke instance methods within the same class or from a different class defined under the same package or different package if the access type allows the same.

Some of the properties of the instance method are the following:

- Instance methods are related to an object rather than a class as they can be invoked after creating an object of the class.
- We can override an instance method as they are resolved using dynamic binding during run time.
- The instance methods are stored in a single memory location.

Let us consider the following program illustrating the working of instance method:

Source Code

```
class myClass {

    // Data member of type String
    String name = "";

    // Defining an instance method
    public void assignName(String name) { this.name = name; }
}
class HelloWorld {
    public static void main(String[] args) {

        // Create an object of the class
        myClass object = new myClass();

        // Calling an instance method in the class 'myClass'.
        object.assignName("Board Infinity");

        // Display the name
        System.out.println(object.name);
    }
}
```

Output:

```
Board Infinity
```

Static methods

An static method in Java is a method that can be called without creating an object of the class. We can reference such methods by the class name or direct reference to the object of that class.

Some of the properties of the static method are the following:

- An static method is related to a class rather than an object of the class. We can call the static method directly without creating an object of the class.
- Static methods are designed in such a way that they can be shared among all objects created using the same class.

- We cannot override static methods as they are resolved using the static binding by the compiler during compile time.

Let us consider the following program illustrating the working of static method:

Source Code

```
// Create a class
class myClass {

    // Create a data member
    public static String name = "";

    // Create static method
    public static void assignName(String passedName)
    {
        name = passedName;
    }
}
class HelloWorld {
    public static void main(String[] args) {

        // Accessing the static method assignName()
        myClass.assignName("Board Infinity");
        System.out.println(myClass.name);

        // Accessing the static method assignName()
        // having the object reference
        myClass object = new myClass();

        // Assign the name
        object.assignName("Java");

        // Display the name
        System.out.println(object.name);
    }
}
```

```
Board Infinity
Java
```

Output:

Constructors in Java

1. Types of constructors
    1. Default Constructor
    2. Parameterized Constructor
2. Constructor Overloading
3. Does constructor return any value?
4. Copying the values of one object into another

5. Does constructor perform other tasks instead of the initialization

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

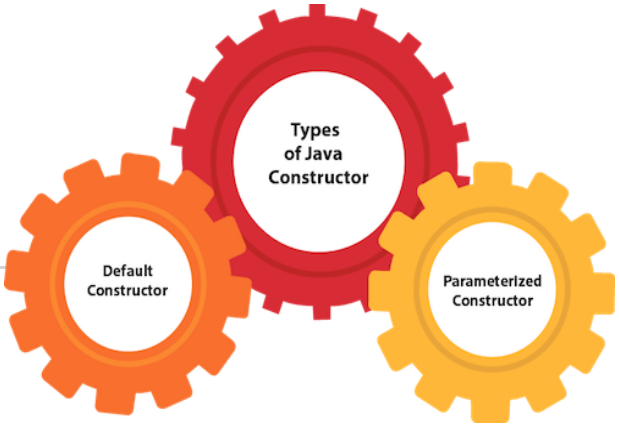
Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors



There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
1. <class_name>(){}
```

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

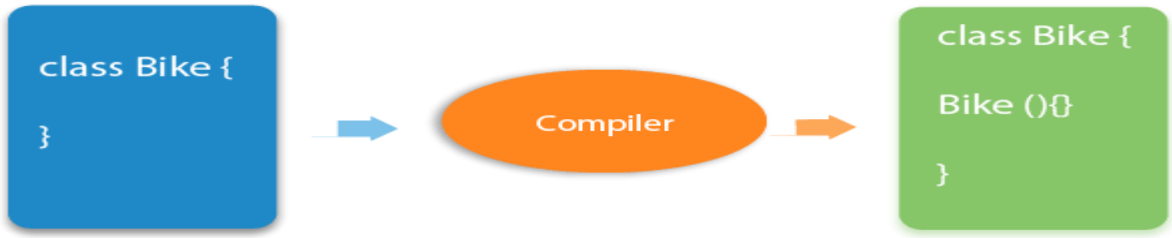
```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. }
```

```
10. }
```
Test it Now

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8.
9. public static void main(String args[]){
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
```
Test it Now

Output:

```
0 null
0 null
```

Explanation:In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1.  //Java Program to demonstrate the use of the parameterized constructor.
2.  class Student4{
3.      int id;
4.      String name;
5.      //creating a parameterized constructor
6.      Student4(int i,String n){
7.      id = i;
8.      name = n;
9.      }
10.     //method to display the values
11.     void display(){System.out.println(id+" "+name);}
12.
13.     public static void main(String args[]){
14.     //creating objects and passing values
15.     Student4 s1 = new Student4(111,"Karan");
16.     Student4 s2 = new Student4(222,"Aryan");
17.     //calling method to display the values of object
18.     s1.display();
19.     s2.display();
20.    }
21. }
```

Test it Now

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
1.  //Java program to overload constructors
2.  class Student5{
3.      int id;
4.      String name;
5.      int age;
6.      //creating two arg constructor
7.      Student5(int i,String n){
8.      id = i;
9.      name = n;
10.     }
11.     //creating three arg constructor
12.     Student5(int i,String n,int a){
13.     id = i;
14.     name = n;
15.     age=a;
```

```
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
    }
}
```

Test it Now

Output:

```
111 Karan 0
222 Aryan 25
```

Method Overloading in Java

1. Different ways to overload the method
2. By changing the no. of arguments
3. By changing the datatype
4. Why method overloading is not possible by changing the return type
5. Can we overload the main method
6. 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.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program.*

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

Test it Now

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.

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

Test it Now

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:

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static double add(int a,int b){return a+b;}
4. }
5. class TestOverloading3{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}
```

Test it Now

Output:

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

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

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:

```
1. class TestOverloading4{
2. public static void main(String[] args){System.out.println("main with String[]");}
3. public static void main(String args){System.out.println("main with String");}
4. public static void main(){System.out.println("main without args");}
5. }
```

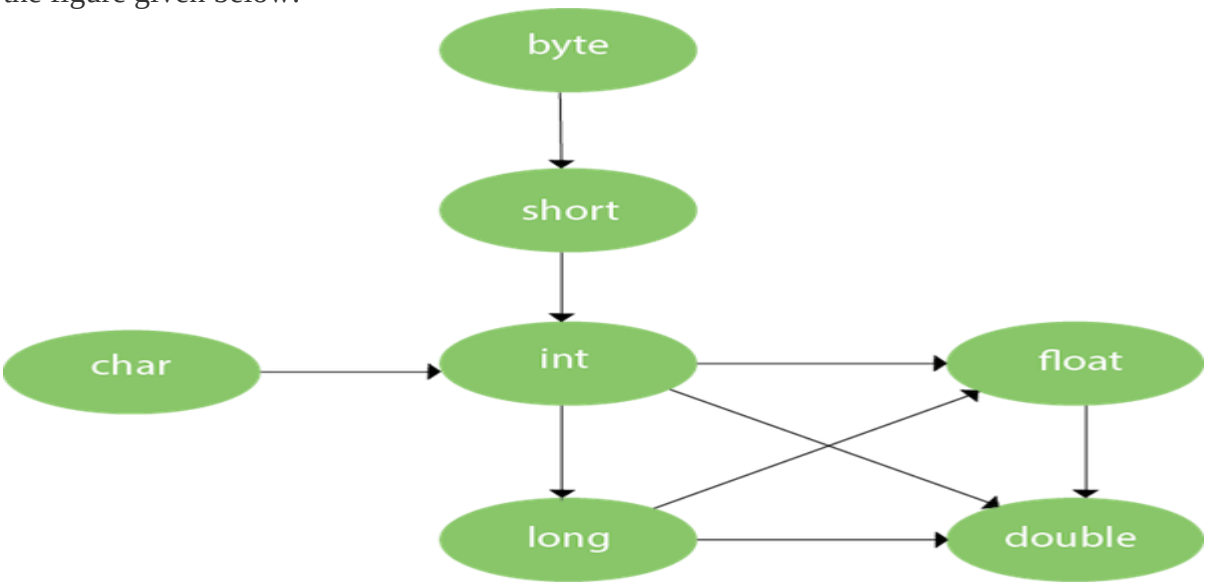
Test it Now

Output:

```
main with String[]
```

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

```
1. class OverloadingCalculation1{
2.   void sum(int a,long b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation1 obj=new OverloadingCalculation1();
7.   obj.sum(20,20);//now second int literal will be promoted to long
8.   obj.sum(20,20,20);
9.
10.  }
```

```
11. }
```
Test it Now

```
Output:40
       60
```

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1.  class OverloadingCalculation2{
2.    void sum(int a,int b){System.out.println("int arg method invoked");}
3.    void sum(long a,long b){System.out.println("long arg method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation2 obj=new OverloadingCalculation2();
7.    obj.sum(20,20);//now int arg sum() method gets invoked
8.    }
9.  }
```
Test it Now

```
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.

```
1.  class OverloadingCalculation3{
2.    void sum(int a,long b){System.out.println("a method invoked");}
3.    void sum(long a,int b){System.out.println("b method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation3 obj=new OverloadingCalculation3();
7.    obj.sum(20,20);//now ambiguity
8.    }
9.  }
```
Test it Now

```
Output:Compile Time Erro
```

Access Modifiers in Java

1. Private access modifier
2. Role of private constructor
3. Default access modifier
4. Protected access modifier
5. Public access modifier
6. Access Modifier with Method Overriding

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

---

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
```

```
8.    A obj=new A();
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12. }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1.  class A{
2.  private A(){}//private constructor
3.  void msg(){System.out.println("Hello java");}
4.  }
5.  public class Simple{
6.   public static void main(String args[]){
7.     A obj=new A();//Compile Time Error
8.   }
9.  }
```

Note: A class cannot be private or protected except nested class.

---

2) Default

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1.  //save by A.java
2.  package pack;
3.  class A{
4.    void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B{
5.    public static void main(String args[]){
6.     A obj = new A();//Compile Time Error
7.     obj.msg();//Compile Time Error
```

```
8.    }
9.  }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.  protected void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.
5.  class B extends A{
6.    public static void main(String args[]){
7.     B obj = new B();
8.     obj.msg();
9.    }
10. }
```

Output:Hello

---

4) Public

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1.  //save by A.java
2.
```

```
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11. }
```

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2. protected void msg(){System.out.println("Hello java");}
3. }
4.
5. public class Simple extends A{
6. void msg(){System.out.println("Hello java");}//C.T.Error
7.  public static void main(String args[]){
8.    Simple obj=new Simple();
9.    obj.msg();
10.   }
11. }
```

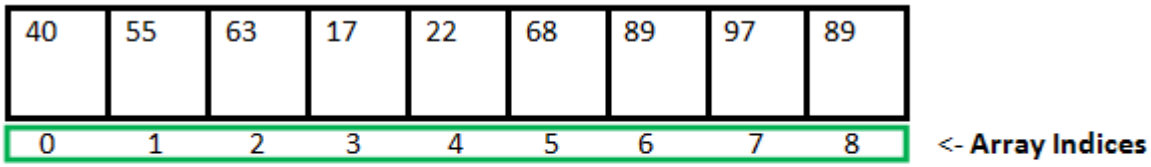
The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Arrays in Java

- In Java, all arrays are dynamically allocated. (discussed below)
- Arrays may be stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property *length*. This is different from C/C++, where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning with 0.
- Java array can also be used as a static field, a local variable, or a method parameter.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values might be

stored in contiguous memory locations(JVM does not guarantee this behavior). In the case of class objects, the actual objects are stored in a heap segment.



Array Length = 9
First Index = 0
Last Index = 8

Note: This storage of arrays helps us randomly access the elements of an array [Support Random Access].

Creating, Initializing, and Accessing an Arrays
One-Dimensional Arrays
The general form of a one-dimensional array declaration is
-- type var-name[];
-- type[] var-name;
An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc., or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.
Example:

```
// both are valid declarations
int intArray[];
int[] intArray;
// similar to int we can declare
// byte , short, boolean, long, float
// double, char
// an array of references to objects of
// the class MyClass (a class created by user)
MyClass myClassArray[];
// array of Object
Object[]  ao,
// array of Collection
// of unknown type
Collection[] ca;
```

Although the first declaration establishes that int Array is an array variable, no actual array exists. It merely tells the compiler that this variable (int Array) will hold an array of the integer type. To link int Array with an actual, physical array of integers, you must allocate one using new and assign it to int Array.
Instantiating an Array in Java
When an array is declared, only a reference of an array is created. To create or give memory to the array, you create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:
var-name = new type [size];
Here, *type* specifies the type of data being allocated, *size* determines the number of elements in the array, and *var-name* is the name of the array variable that is linked to the array. To use *new* to allocate an array, you must specify the type and number of elements to allocate.
Example:

```
//declaring array
int intArray[];
// allocating memory to array
intArray = new int[20];
```

```
// combining both statements in oneint[]
intArray = new int[20];
```

Note: The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types). Do refer to default array values in Java.
Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory to hold the array, using new, and assign it to the array variable. Thus, in Java, all arrays are dynamically allocated.

Array Literal in Java
In a situation where the size of the array and variables of the array are already known, array literals can be used.

```
// Declaring array literal
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java.

Accessing Java Array Elements using for Loop
Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

```
 // accessing the elements of the specified array
for (int i = 0; i < arr.length; i++)
        System.out.println("Element at index " + i + " : "+ arr[i]);
```

Implementation:

- Java

```
// Java program to illustrate creating an array
// of integers,  puts some values in the array,
// and prints each value to standard output.

class GFG {
   public static void main(String[] args)
   {
      // declares an Array of integers.
      int[] arr;

      // allocating memory for 5 integers.
      arr = new int[5];

      // initialize the first elements of the array
      arr[0] = 10;

      // initialize the second elements of the array
      arr[1] = 20;

      // so on...
      arr[2] = 30;
      arr[3] = 40;
      arr[4] = 50;

      // accessing the elements of the specified array
      for (int i = 0; i < arr.length; i++)
         System.out.println("Element at index " + i
```

```
                    + " : " + arr[i]);
    }
}
```

Output
Element at index 0 : 10

Element at index 1 : 20

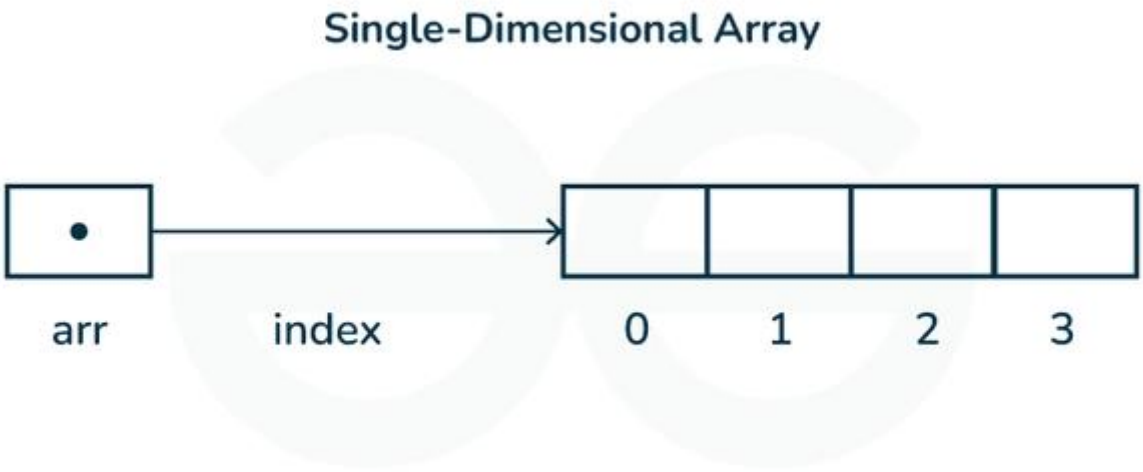
Element at index 2 : 30

Element at index 3 : 40

Element at index 4 : 50

Complexity of the above method:
Time Complexity: O(n)
Auxiliary Space: O(1)

You can also access java arrays using for each loops.



Arrays in Java

Arrays of Objects in Java
An array of objects is created like an array of primitive-type data items in the following way.
Student[] arr = new Student[5]; //student is a user-defined class
Syntax:
-- data type[] arrName;
-- datatype arrName[];
-- datatype [] arrName;
Example of Arrays of Objects
Example 1:
Below is the implementation of the above mentioned topic:

- Java

```
import java.io.*;

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

      int [] arr=new int [4];
      // 4 is the size of arr

      System.out.println("Array Size:"+arr.length);
    }
}
```

Output
Array Size:4

The student Array contains five memory spaces each of the size of student class in which the address of five Student objects can be stored. The Student objects have to be instantiated using the constructor of the Student class, and their references should be assigned to the array elements in the following way.
Example 2:
Below is the implementation of the above mentioned topic:

- Java

```
// Java program to illustrate creating
//  an array of objects

class Student {
    public int roll_no;
    public String name;
    Student(int roll_no, String name)
    {
        this.roll_no = roll_no;
        this.name = name;
    }
}

// Elements of the array are objects of a class Student.
public class GFG {
    public static void main(String[] args)
    {
        // declares an Array of Students
        Student[] arr;

        // allocating memory for 5 objects of type Student.
        arr = new Student[5];
```

```
        // initialize the first elements of the array
        arr[0] = new Student(1, "aman");

        // initialize the second elements of the array
        arr[1] = new Student(2, "vaibhav");

        // so on...
        arr[2] = new Student(3, "shikar");
        arr[3] = new Student(4, "dharmesh");
        arr[4] = new Student(5, "mohit");

        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at " + i + " : "
                               + arr[i].roll_no + " "
                               + arr[i].name);
    }
}
```

Output
Element at 0 : 1 aman

Element at 1 : 2 vaibhav

Element at 2 : 3 shikar

Element at 3 : 4 dharmesh

Element at 4 : 5 mohit

Complexity of the above method:
Time Complexity: O(n)
Auxiliary Space : O(1)

Example 3
An array of objects is also created like :

- Java

```
// Java program to illustrate creating
//  an array of objects

class Student
{

    public String name;
    Student(String name)
    {
        this.name = name;
    }
     @Override
    public String toString(){
```

```
        return name;
    }
}

// Elements of the array are objects of a class Student.
public class GFG
{
    public static void main (String[] args)
    {
          // declares an Array and initializing  the elements of the array
        Student[] myStudents = new Student[]{new Student("Dharma"),new Student("sanvi"),new
Student("Rupa"),new Student("Ajay")};

        // accessing the elements of the specified array
        for(Student m:myStudents){
            System.out.println(m);
        }
    }
}
```

Output
Dharma

sanvi

Rupa

Ajay

What happens if we try to access elements outside the array size?
JVM throws ArrayIndexOutOfBoundsException to indicate that the array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of an array.
Below code shows what happens if we try to access elements outside the array size:

- Java

```
// Code for showing error "ArrayIndexOutOfBoundsException"

public class GFG {
    public static void main(String[] args)
    {
        int[] arr = new int[4];
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;

        System.out.println(
            "Trying to access element outside the size of array");
```

```
        System.out.println(arr[5]);
    }
}
```

Output
Trying to access element outside the size of array
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4
at GFG.main(GFG.java:13)
Example (Iterating the array):

- Java

```
public class GFG {
    public static void main(String[] args)
    {
        int[] arr = new int[2];
        arr[0] = 10;
        arr[1] = 20;

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

Output
10

20

Complexity of the above method:
Time Complexity: O(n),here n is size of array.
Auxiliary Space: O(1), since no extra space required.

Multidimensional Arrays in Java
Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other arrays. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.
Syntax of Java Multidimensional Array
There are 2 methods to declare Java Multidimensional Arrays as mentioned below:
-- datatype [][] arrayrefvariable;
-- datatype arrayrefvariable[][];
Example:

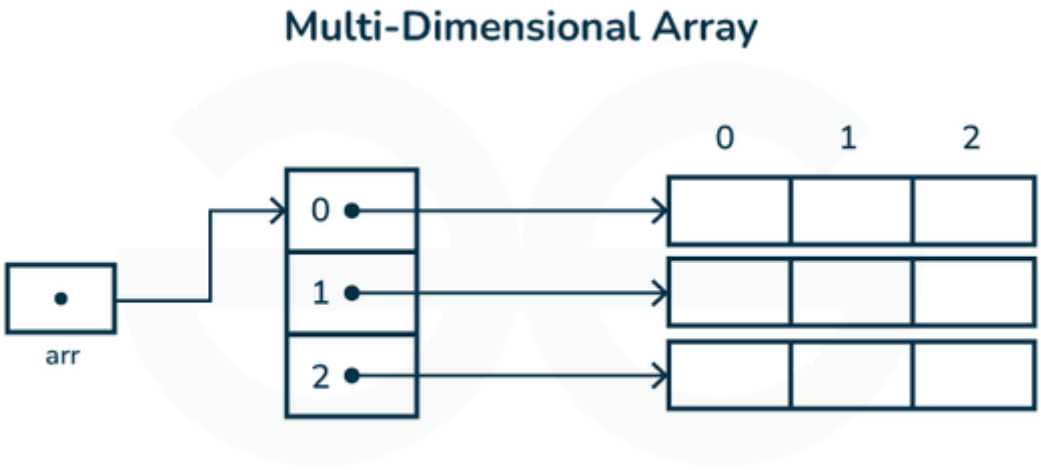
- Java

```
// Java Program to demonstrate
// Java Multidimensional Array
import java.io.*;
```

```java
// Driver class
class GFG {
    public static void main(String[] args)
    {
        // Syntax
        int[][] arr = new int[3][3];
        // 3 row and 3 column

          // Number of Rows
        System.out.println("Number of Rows:"+
                           arr.length);

          // Number of Columns
        System.out.println("Number of Columns:"+
                           arr[0].length);
    }
}
```

Output
Number of Rows:3

Number of Columns:3



Arrays in Java

Multidimensional Array Declaration
int[][] intArray = new int[10][20]; //a 2D array or matrix
int[][][] intArray = new int[10][20][10]; //a 3D array
Example of Muilti Dimensional Array in Java
Example 1:
Below is the implementation of the above method:

- Java

```java
// Java Program to Multidimensional Array

// Driver Class
public class multiDimensional {
    // main function
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][]
            = { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };

        // printing 2D array
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++)
                System.out.print(arr[i][j] + " ");

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

Output
2 7 9

3 6 1

7 4 2

Passing Arrays to Methods
Like variables, we can also pass arrays to methods. For example, the below program passes the array to method *sum* to calculate the sum of the array's values.

- Java

```java
// Java program to demonstrate
// passing of array to method

public class Test {
    // Driver method
    public static void main(String args[])
    {
        int arr[] = { 3, 1, 2, 5, 4 };

        // passing array to method m1
        sum(arr);
    }

    public static void sum(int[] arr)
    {
```

```
        // getting sum of array values
        int sum = 0;

        for (int i = 0; i < arr.length; i++)
            sum += arr[i];

        System.out.println("sum of array values : " + sum);
    }
}
```

Output
sum of array values : 15

Complexity of the above method:
Time Complexity: O(n)
Auxiliary Space : O(1)

Returning Arrays from Methods
As usual, a method can also return an array. For example, the below program returns an array from method *m1*.

- Java

```
// Java program to demonstrate
// return of array from method

class Test {
    // Driver method
    public static void main(String args[])
    {
        int arr[] = m1();

        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }

    public static int[] m1()
    {
        // returning  array
        return new int[] { 1, 2, 3 };
    }
}
```

Output
1 2 3

Complexity of the above method:

Time Complexity: O(n)
Auxiliary Space : O(1)

Class Objects for Arrays
Every array has an associated Class object, shared with all other arrays with the same component type.

- Java

```
// Java program to demonstrate
// Class Objects for Arrays

class Test {
   public static void main(String args[])
   {
      int intArray[] = new int[3];
      byte byteArray[] = new byte[3];
      short shortsArray[] = new short[3];

      // array of Strings
      String[] strArray = new String[3];

      System.out.println(intArray.getClass());
      System.out.println(
         intArray.getClass().getSuperclass());
      System.out.println(byteArray.getClass());
      System.out.println(shortsArray.getClass());
      System.out.println(strArray.getClass());
   }
}
```

Output
class [I

class java.lang.Object

class [B

class [S

class [Ljava.lang.String;

Explanation of the above method:

1. The string "[I" is the run-time type signature for the class object "array with component type *int*."
2. The only direct superclass of an array type is java.lang.Object.
3. The string "[B" is the run-time type signature for the class object "array with component type *byte*."
4. The string "[S" is the run-time type signature for the class object "array with component type *short*."
5. The string "[L" is the run-time type signature for the class object "array with component type of a Class." The Class name is then followed.

Java Array Members

Now, as you know that arrays are objects of a class, and a direct superclass of arrays is a class Object. The members of an array type are all of the following:

- The public final field *length* contains the number of components of the array. Length may be positive or zero.
- All the members are inherited from class Object; the only method of Object that is not inherited is its clone method.
- The public method *clone()* overrides the clone method in class Object and throws no checked exceptions.

Arrays Types and Their Allowed Element Types

Array Types	Allowed Element Types
Primitive Type Arrays	Any type which can be implicitly promoted to declared type.
Object Type Arrays	Either declared type objects or it's child class objects.
Abstract Class Type Arrays	Its child-class objects are allowed.
Interface Type Arrays	Its implementation class objects are allowed.

Cloning of Single-Dimensional Array in Java

When you clone a single-dimensional array, such as Object[], a "deep copy" is performed with the new array containing copies of the original array's elements as opposed to references.

Below is the implementation of the above method:

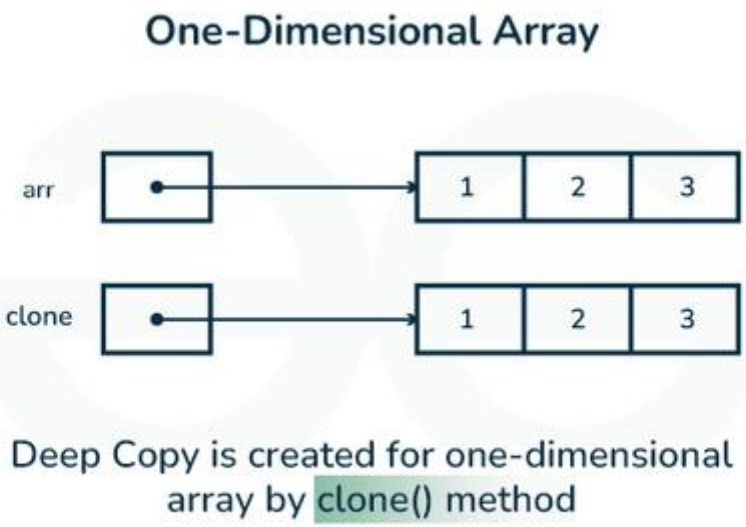
- Java

```
// Java program to demonstrate
// cloning of one-dimensional arrays

class Test {
   public static void main(String args[])
   {
      int intArray[] = { 1, 2, 3 };

      int cloneArray[] = intArray.clone();

      // will print false as deep copy is created
      // for one-dimensional array
      System.out.println(intArray == cloneArray);

      for (int i = 0; i < cloneArray.length; i++) {
         System.out.print(cloneArray[i] + " ");
      }
   }
}
```

Output

false

1 2 3



Arrays in Java

Cloning Multidimensional Array in Java
A clone of a multi-dimensional array (like Object[][]) is a “shallow copy,” however, which is to say that it creates only a single new array with each element array a reference to an original element array, but subarrays are shared.

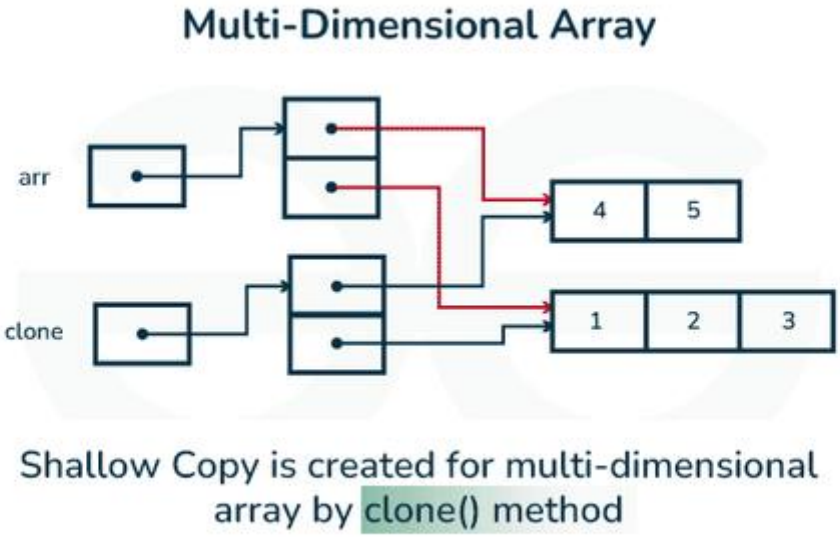
- Java

```
// Java program to demonstrate
// cloning of multi-dimensional arrays

class Test {
   public static void main(String args[])
   {
      int intArray[][] = { { 1, 2, 3 }, { 4, 5 } };

      int cloneArray[][] = intArray.clone();

      // will print false
      System.out.println(intArray == cloneArray);

      // will print true as shallow copy is created
      // i.e. sub-arrays are shared
      System.out.println(intArray[0] == cloneArray[0]);
      System.out.println(intArray[1] == cloneArray[1]);
   }
}
```

Output
false

true

true



Frequently Asked Questions in Java Arrays

1. Can we specify the size of array as long?

No we can't specify the size of array as long but we can specify it as int or short.

2. Which is the direct superclass of an array in Java?

An Object is direct superclass of an array in Java.

3. Which Interfaces are implemented by Arrays in Java?

Every array type implements the interfaces Cloneable and java.io.Serializable.

4. Can we alter the size of Array?

The size of the array cannot be altered(once initialized). However, an array reference can be made to point to another array.