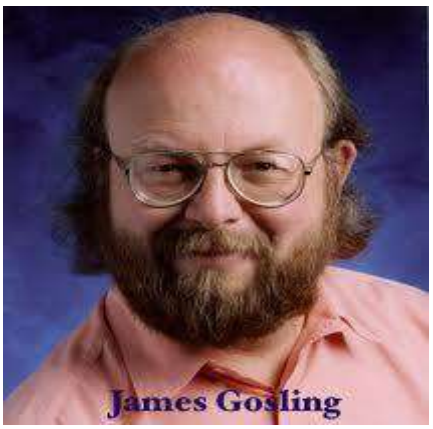
History of java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team.



Java team members (also known as **Green Team**), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. There are given significant points that describe the history of Java.

1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
2) Initially designed for small, embedded systems in electronic appliances like set-top boxes.
3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
4) 4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java named "Oak"?

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) **Why had they chosen java name for Java language?**
   The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

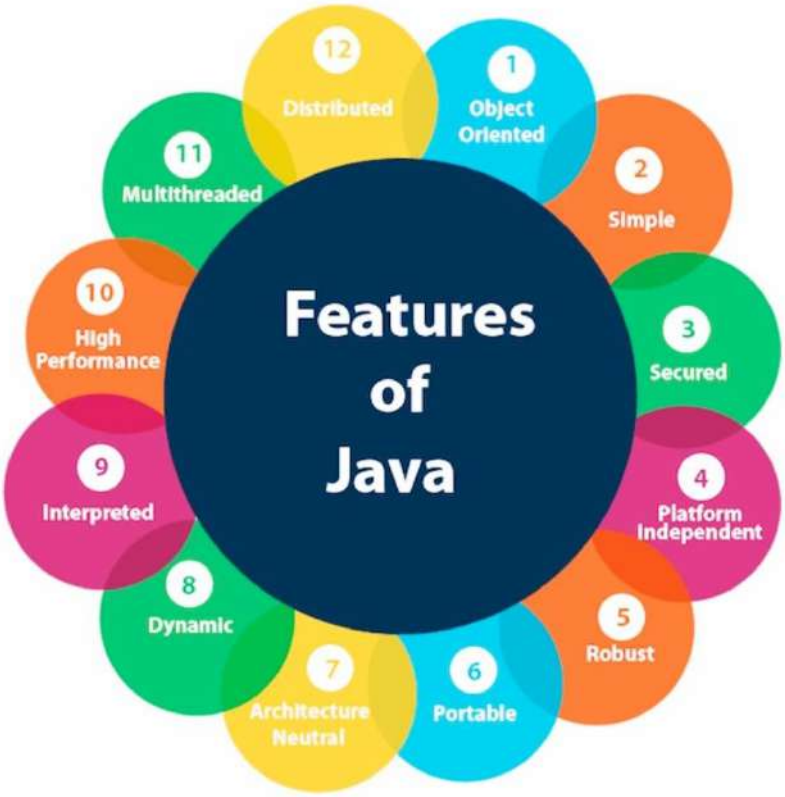
8) Java is an island of Indonesia where the first coffee was produced (called java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having coffee near his office.
9) Notice that Java is just a name, not an acronym.
10) developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
11) 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
12) JDK 1.0 released in(January 23, 1996). After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds the new features in Java.

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)

Features of Java



The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.
A list of most important features of Java language is given below.

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

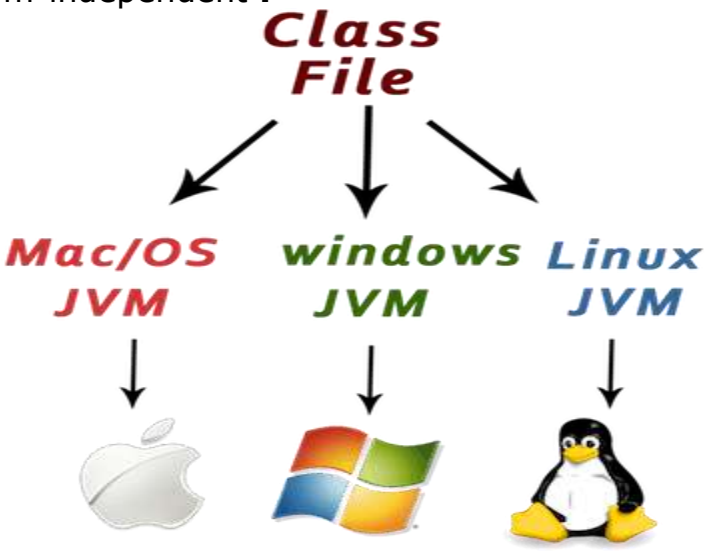
Simple:Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object - Oriented : Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.
Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.
Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform-independent **:**

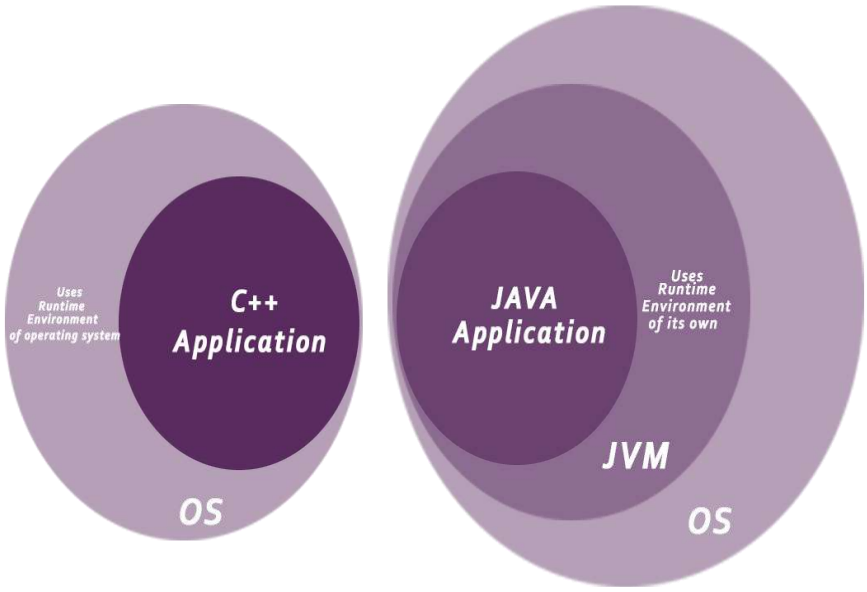


Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.
There are two types of platforms software-based and hardware-based. Java provides a software-based platform.
The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).
Secure : Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package

for the classes of the local file system from those that are imported from network sources.

- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust : Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture Neutral : Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable : Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High Performance : Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed : Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi Threaded: A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Dynamic : Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

C++ vs Java(Migration from C++ to JAVA)

There are many differences and similarities between the C++ programming language and Java. A list of top differences between C++ and Java are given below:

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.

Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent.

Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comment.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for

		the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in C language, single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

Note

- Java doesn't support default arguments like C++.

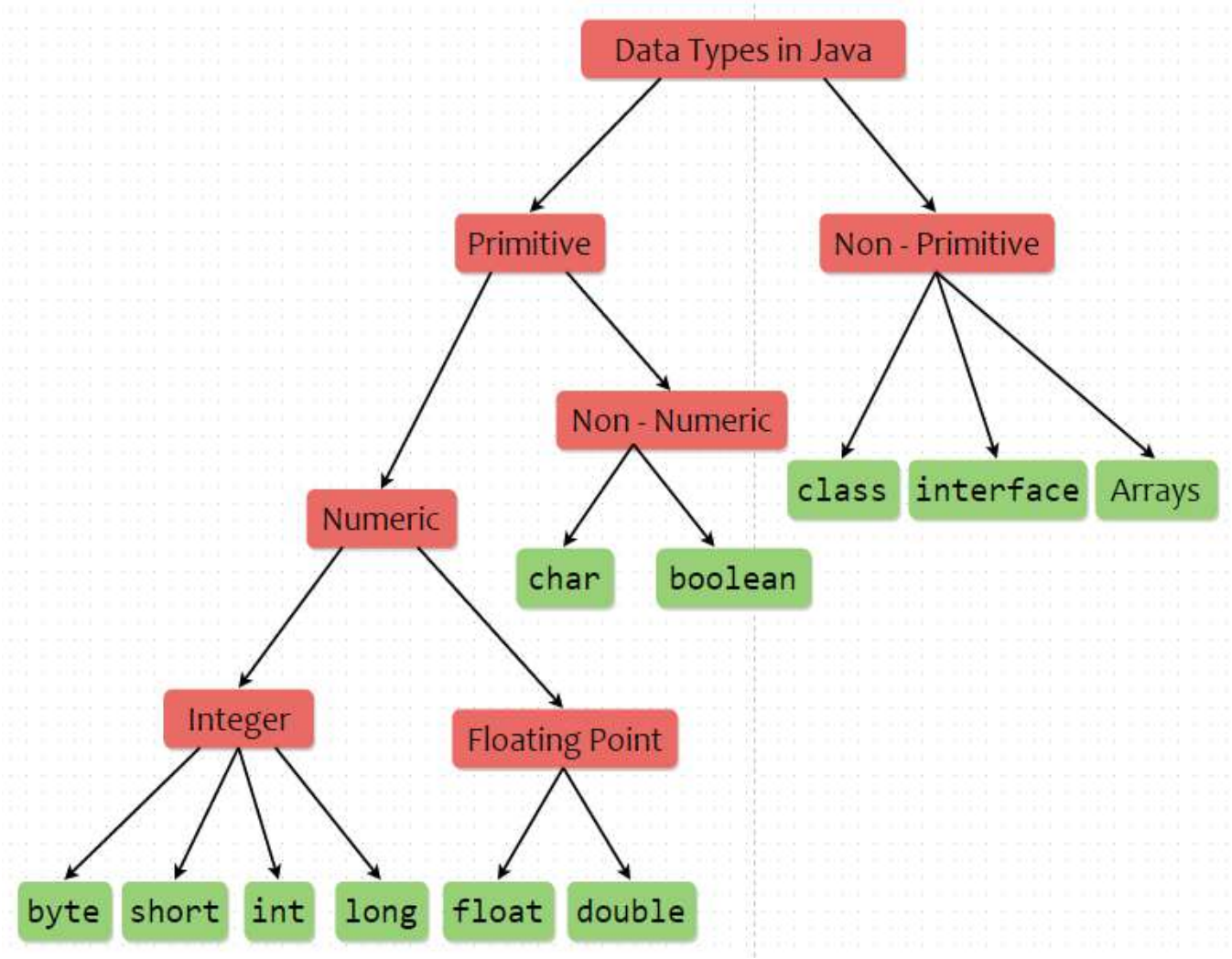
- Java does not support header files like C++. Java uses the import keyword to include different classes and methods.

Data Types in Java

Data type specifies the size and type of values that can be stored in an identifier. The Java language is rich in its data types. Different data types allow you to select the type appropriate to the needs of the application.

Data types in Java are classified into two types:

1. **Primitive—**which include Integer, Character, Boolean, and Floating Point. A primitive data type specifies the size and type of variable values, and it has no additional methods.
2. **Non-primitive—** These data types are special types of data which are user defined, i,e, the program contains their definition. Some examples are- Classes, Interfaces, and Arrays.



These are the data types in Java. The primitive data types are much like the C++ data types. But, unlike C++, **String is not a primitive data type. Strings in Java are objects**. The Java library has the String class and we create their objects when we deal with strings. When it comes to numeric data types, Java does not have any notion of unsigned integers. So, the numeric data types in Java can always store positive and negative values. We will talk about the non-primitive data types later. For now, focus on getting comfy with the primitive data types.

The sizes of the primitive data types are –

- byte – 1 Byte
- short – 2 Bytes

- int – 4 Bytes
- long – 8 Bytes
- float – 4 Bytes – up to 7 digits of precision
- double – 8 Bytes – up to 15 digits of precision
- char – 2 Bytes – But holds only up to 1 character
- boolean – JVM Dependant

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Operators

Operator in Java is a symbol which is used to perform operations. For example: +, -, *, / etc.There are many types of operators in Java which are given below:

Arithmetic Operators in JAVA

Operator	Meaning	Example
+	Addition	`int a = 10;` `int b = a + 10; // 20`
–	Subtraction	`int a = 10;` `int b = a - 5; // 5`
*	Multiplication	`int a = 10;` `int b = a * 7; // 70`
/	Division	`int a = 50;` `int b = a / 5; // 10`
%	Remainder / Modulo	`int a = 11;` `int b = a % 3; // 2`

++	Increment	`int a = 10;`
		`++a; // 11`
		`a++; // 12`

—	Decrement	`int a = 10;`
		`a--; // 9`
		`--a; // 8`

Relational Operators in Java

Operator	Meaning	Example

<	less than	`int a = 10, b = 20;`
		
		`if (a < b) {`
		`  // true`
		`}`

>	greater than	`int a = 10, b = 20;`
		
		`if (a > b) {`
		`  // false`
		`}`

<=	less than or equal to	`int a = 10;`
		
		`if (a <= 10) {`
		`  // true`
		`}`

>=	greater than or equal to	`int b = 20;`
		
		`if (b >= 20) {`
		`  // true`
		`}`

==	is it equal to	`int a = 10, b = 20;`
		
		`if (a == b) {`
		`  // false`
		`}`

!=	is not equal to	`int a = 10, b = 20;`
		
		`if (a != b) {`
		`  // true`
		`}`

Logical Operators in Java

Operator	Meaning	Example

&&	AND	`boolean a = true, b = false;`
		
		`if (a && b) {`
		`  // false`
		`}`

||	OR	`boolean a = true, b = false;`
		
		`if (a || b) {`
		`  // true`
		`}`

!	NOT	`boolean a = true;`
		
		`if (!a) {`
		`  // false`
		`}`

Bitwise Operators in Java

Operator	Meaning	Example

&	bitwise AND	`int a = 2;`
		
		`a = a & 1; // 0`

|	bitwise OR	`int a = 2;`
		
		`a = a | 1; // 3`

^	bitwise XOR	`int a = 3;`
		
		`a = a ^ 1; // 2`

~	bitwise NOT / complement	`int  a = 8;`
		
		`a = ~a; // -9`
		`// 2's complement`

<<	left shift	`int a = 8;`
		
		`a = a << 1; // 16`

>>	right shift	`int a = 8;`
		
		`a = a >> 1; // 4`

>>>	right shift with zero fill	`int a = 8;`
		
		`a = a >>> 1; // 4`

Shorthand Operators

Operator	Meaning	Example

x+=1;	x=x+1;	`int a = 2;`
		
		`a += 1; // 3`

x-=1;	x=x-1;	`int a = 8;`
		
		`a -= 1; // 7`

x*=2;	x=x*2;	`int a = 8;`
		
		`a *= 2; // 16`

x/=2;	x=x/2;	`int a = 8;`
		
		`a /= 2; // 4`

x%=2;	x=x%2;	`int a = 8;`
		
		`a %= 2; // 0`

Operator Precedence in Java

Operator	Meaning
.	Member Selection
function()	Function Call
arr[]	Array's Random Access
-var	Unary Minus
var++	Postfix Increment
var–	Postfix Decrement
++var	Prefix Increment
–var	Prefix Decrement
!	Logical Negation
~	Complement
(data_type)	Type Casting
*	Multiplication
/	Division
%	Modulo Division
+	Addition
-	Subtraction
<<	Left Shift
>>	Right Shift
>>>	Right Shift with Zero Fill
<	Less than
<=	Less than or equal to

<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
instanceof	Object-Class comparison
==	Is it equal to
!=	Is not equal to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
(condition) ? true : false	Ternary Operator

These are the precedence levels. We still haven't discussed a few operators, such as `instanceof`. We will discuss them soon when we talk more about classes. You need not mug up the whole hierarchy, but if you have an idea of the hierarchy between the arithmetic operators, relational operators and logical operators, it's enough, because those are the most frequently used ones. But if you ever need, you can always come here to lookup the precedence levels of various operators.

Java Variables

In Java, Variables are the data containers that save the data values during Java program execution. Every Variable in Java is assigned a data type that designates the type and quantity of value it can hold. A variable is a memory location name for the data.

Variables in Java

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.
- In Java, all variables must be declared before use.

How to Declare Variables in Java?

We can declare variables in Java as pictorially depicted below as a visual aid.

From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

1. **datatype**: Type of data that can be stored in this variable.
2. **data_name:** Name was given to the variable.

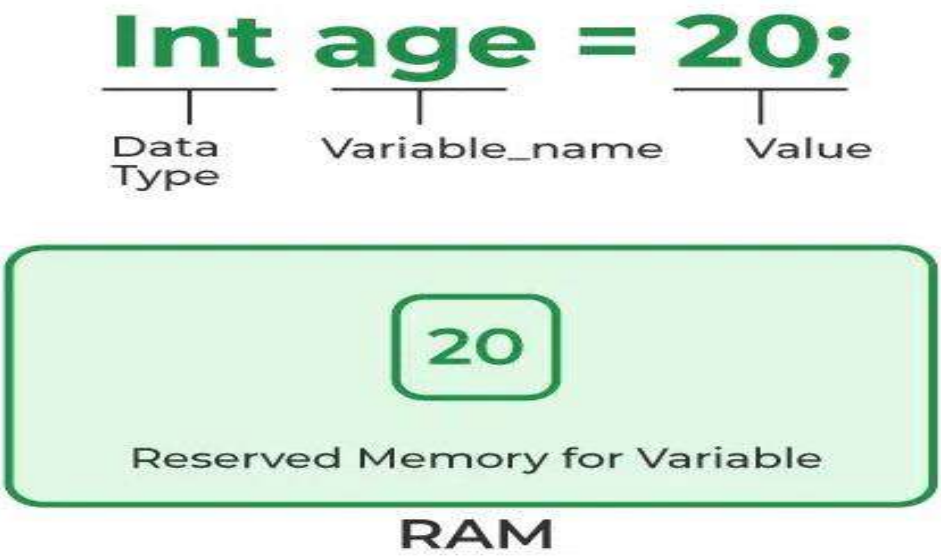
In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

How to Initialize Variables in Java?

It can be perceived with the help of 3 components that are as follows:

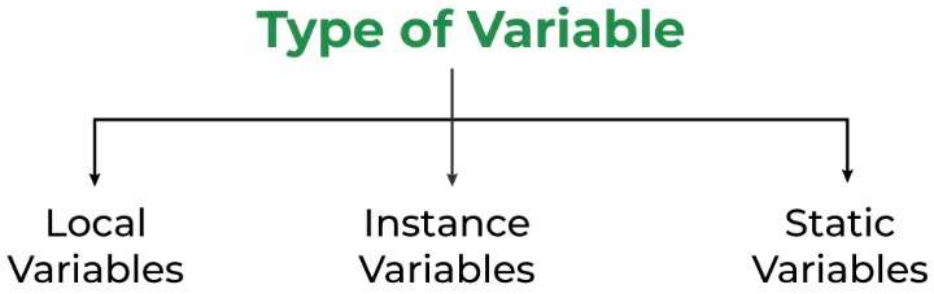
- **datatype**: Type of data that can be stored in this variable.
- **variable_name**: Name given to the variable.
- **value**: It is the initial value stored in the variable.



Types of Variables in Java

Now let us discuss different types of variables which are listed as follows:

1. Local Variables
2. Instance Variables
3. Static Variables



1) Local Variable

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

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

2) Instance Variable

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

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

3) Static variable

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

Example to understand the types of variables in java

```
1.  public class A
2.  {
3.     static int m=100;//static variable
4.     void method()
5.     {
6.        int n=90;//local variable
7.     }
8.     public static void main(String args[])
9.     {
10.       int data=50;//instance variable
11.    }
12. }//end of class
```

Final variables in Java

Final Keyword In Java

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
```

```
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. }
```

```
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. }
```

```
Output:

Compile Time Error
```

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. }
```

```
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. }
```

```
Output:

Compile Time Error
```

Q) Can we declare a constructor final?

No, because constructor is never inherited.

finalize() Method in Java and How to Override it?

The Java **finalize() method** of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize() method completes immediately, Garbage Collector destroys that object.
Finalization: Just before destroying any object, the garbage collector always calls finalize() method to perform clean-up activities on that object. This process is known as Finalization in Java.

Syntax:

```
protected void finalize throws Throwable{}
```

Since the Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class, Garbage Collector can call the finalize() method on any java object.

Why finalize() method is used?

finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.

How to override finalize() method?

The finalize method, which is present in the Object class, has an empty implementation. In our class, clean-up activities are there. Then we have to override this method to define our clean-up activities.

In order to Override this method, we have to define and call finalize within our code explicitly.

```
// Java code to show the
// overriding of finalize() method

import java.lang.*;

// Defining a class demo since every java class
// is a subclass of predefined Object class
// Therefore demo is a subclass of Object class
public class demo {

    protected void finalize() throws Throwable
    {
        try {

            System.out.println("inside demo's finalize()");
        }
        catch (Throwable e) {

            throw e;
        }
        finally {

            System.out.println("Calling finalize method"
                               + " of the Object class");

            // Calling finalize() of Object class
            super.finalize();
        }
    }

    // Driver code
    public static void main(String[] args) throws Throwable
    {

        // Creating demo's object
        demo d = new demo();

        // Calling finalize of demo
        d.finalize();
    }
}
```

Output:

```
inside demo's finalize()
Calling finalize method of the Object class
```

Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

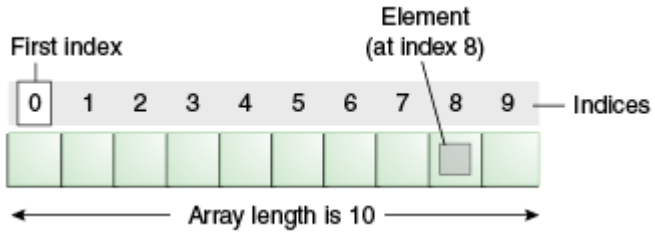
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array. Let's see the simple example to get the minimum number of an array using a method.

```
1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. class Testarray2{
4. //creating a method which receives an array as a parameter
5. static void min(int arr[]){
6. int min=arr[0];
7. for(int i=1;i<arr.length;i++)
8.  if(min>arr[i])
```

```
9.    min=arr[i];
10.
11. System.out.println(min);
12. }
13.
14. public static void main(String args[]){
15. int a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}
```

Output:

```
3
```

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. public class TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. static void printArray(int arr[]){
6. for(int i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
9.
10. public static void main(String args[]){
11. printArray(new int[]{10,22,44,66});//passing anonymous array to method
12. }}
```

Output:

```
10
22
44
66
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//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();
}
}}
```

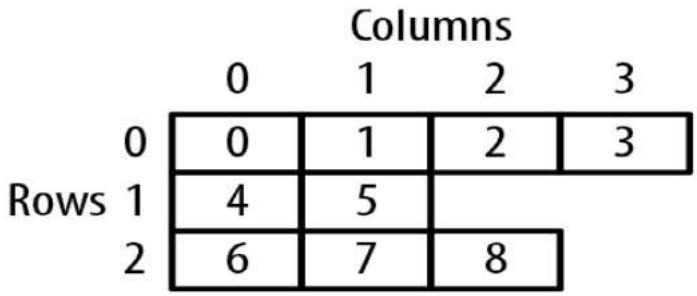
Output:

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

Jagged Arrays In Java

Jagged arrays are also known as ragged arrays. They are the arrays containing arrays of different length. Consider the below example in which each row consists

of different number of elements. First row contains 4 elements, second row contains 2 elements and third row contains 3 elements.



Example

```
import java.util.Arrays;
public class Test {
        public static void main(String args[]){
                int[][] jaggedArray = new int[3][];
                jaggedArray[0] = new int[]{0,1,2,3};
                jaggedArray[1] = new int[]{4,5};
                jaggedArray[2] = new int[]{6,7,8};

                for(int[] row : jaggedArray){
                        System.out.println(Arrays.toString(row));
                }
        }
}
```

Output

[0, 1, 2, 3]

[4, 5]

[6, 7, 8]

Class Definition in Java

In object-oriented programming, a **class** is a basic building block. It can be defined as template that describes the data and behaviour associated with the class instantiation. Instantiating is a class is to create an object (variable) of that class that can be used to access the member variables and methods of the class.

A class can also be called a logical template to create the objects that share common properties and methods.

For example, an Employee class may contain all the employee details in the form of variables and methods. If the class is instantiated i.e. if an object of the class is created (say e1), we can access all the methods or properties of the class.

Defining a Class in Java

Java provides a reserved keyword **class** to define a class. The keyword must be followed by the class name. Inside the class, we declare methods and variables.

In general, class declaration includes the following in the order as it appears:

1. **Modifiers:** A class can be public or has default access.
2. **class keyword:** The class keyword is used to create a class.
3. **Class name:** The name must begin with an initial letter (capitalized by convention).
4. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

Syntax:

```
1. <access specifier> class class_name
2. {
3. // member variables
4. // class methods
5. }
```

Java Class Example

Example 1:

Let's consider the following example to understand how to define a class in Java and implement it with the object of class.

Calculate.java

```
1.  // class definition
2.  public class Calculate {
3.
4.    // instance variables
5.    int a;
6.    int b;
7.
8.    // constructor to instantiate
9.    public Calculate (int x, int y) {
10.     this.a = x;
11.     this.b = y;
12.   }
13.
14.   // method to add numbers
15.   public int add () {
16.     int res = a + b;
17.     return res;
18.   }
19.
20.   // method to subtract numbers
21.   public int subtract () {
22.     int res = a - b;
23.     return res;
24.   }
25.
26.   // method to multiply numbers
27.   public int multiply () {
28.     int res = a * b;
29.     return res;
30.   }
31.
32.   // method to divide numbers
```

```
33.   public int divide () {
34.     int res = a / b;
35.     return res;
36.   }
37.
38.
39.   // main method
40.   public static void main(String[] args) {
41.     // creating object of Class
42.     Calculate c1 = new Calculate(45, 4);
43.
44.     // calling the methods of Calculate class
45.     System.out.println("Addition is :" + c1.add());
46.     System.out.println("Subtraction is :" + c1.subtract());
47.     System.out.println("Multiplication is :" + c1.multiply());
48.     System.out.println("Division is :" + c1.divide());
49.
50.
51.   }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac Calculate.java

C:\Users\Anurati\Desktop\abcDemo>java Calculate
Addition is :49
Subtraction is :41
Multiplication is :180
Division is :11
```

Access Modifiers in Java

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;
```

```
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

```
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

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
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.

Java - Constructors

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Syntax

Following is the syntax of a constructor −

```
class ClassName {
  ClassName() {
  }
}
```

What Are the Rules for Creating Constructors in Java?

There are a total of three rules defined for creating a constructor.

1. The constructor's and class's name must be identical

2. You cannot define an explicit value to a constructor

3. A constructor cannot be any of these: static, synchronized, abstract, or final

One thing to note here is that you can have a public, private, or protected constructor in Java by using access modifiers that control object creation.

Java allows two types of constructors namely −

- No argument Constructors
- Parameterized Constructors

No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

Example

```
Public class MyClass {
   Int num;
   MyClass() {
      num = 100;
   }
}
```

You would call constructor to initialize objects as follows

```
public class ConsDemo {
   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.num + " " + t2.num);
   }
}
```

This would produce the following result

```
100 100
```

Parameterized Constructors

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor −

```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass(int i ) {
      x = i;
   }
}
```

You would call constructor to initialize objects as follows −

```
public class ConsDemo {
   public static void main(String args[]) {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce the following result −

```
10 20
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

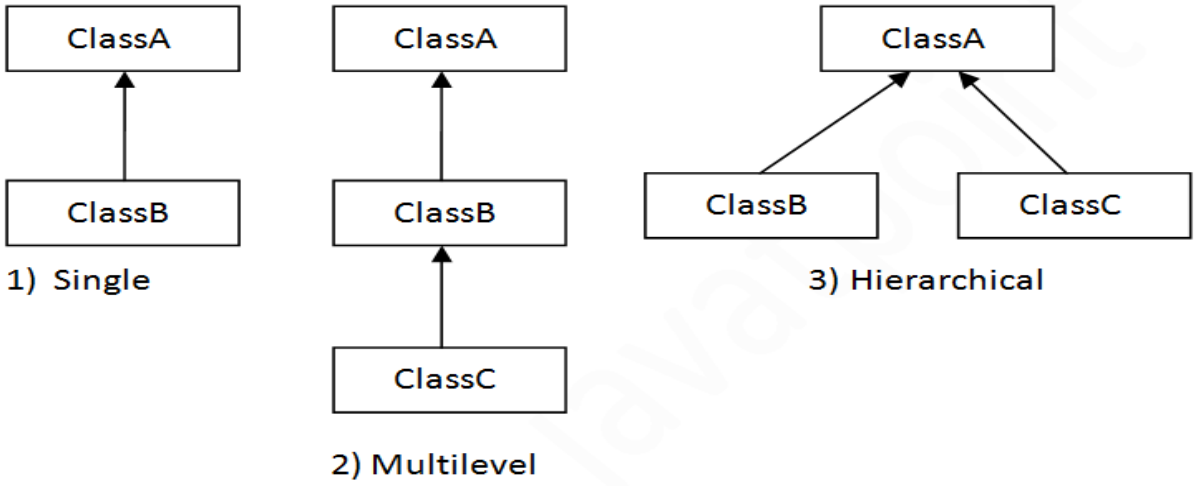
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in java

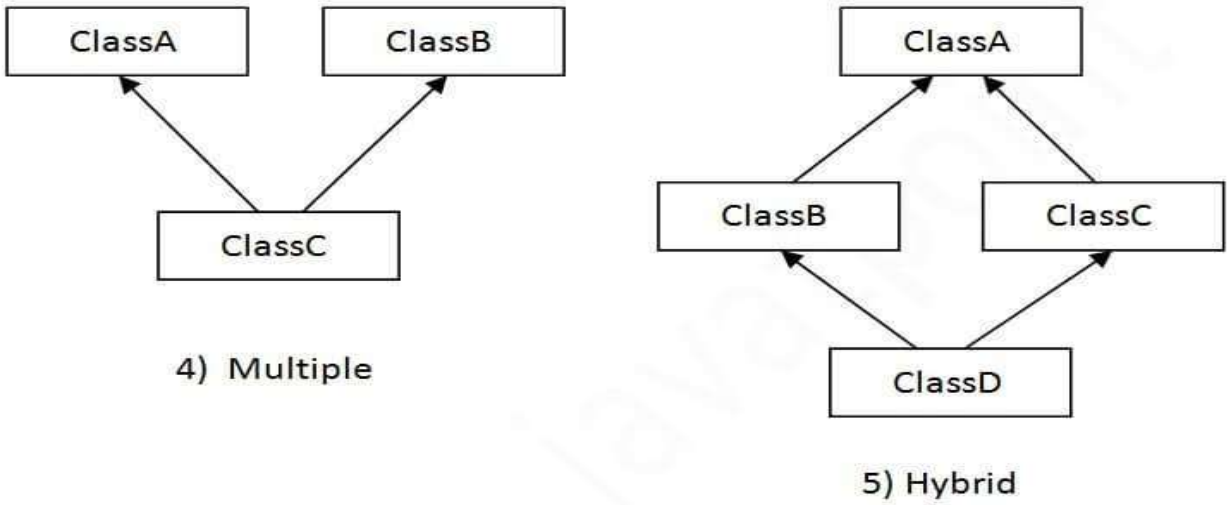
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
```

```
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Interface in Java

1. Interface
2. Example of Interface
3. Multiple inheritance by Interface
4. Why multiple inheritance is supported in Interface while it is not supported in case of class.
5. Marker Interface
6. Nested Interface

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
1. interface <interface_name>{
2.
3.     // declare constant fields
4.     // declare methods that abstract
5.     // by default.
6. }
```

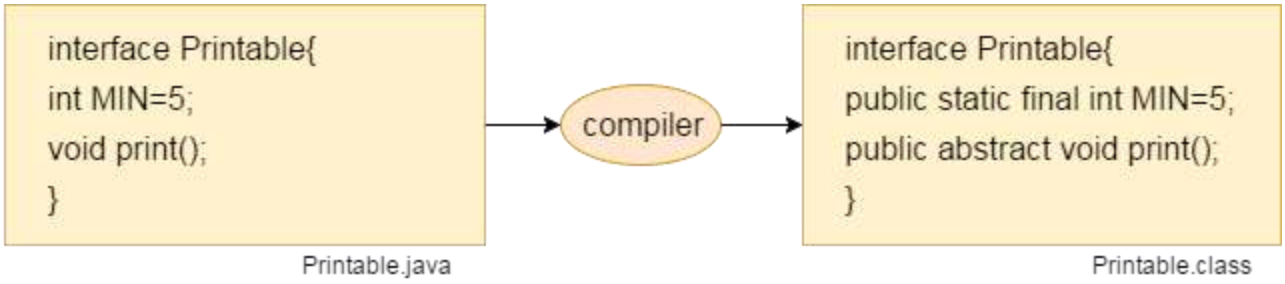
Java 8 Interface Improvement

Since Java 8, interface can have default and static methods which is discussed later.

Internal addition by the compiler

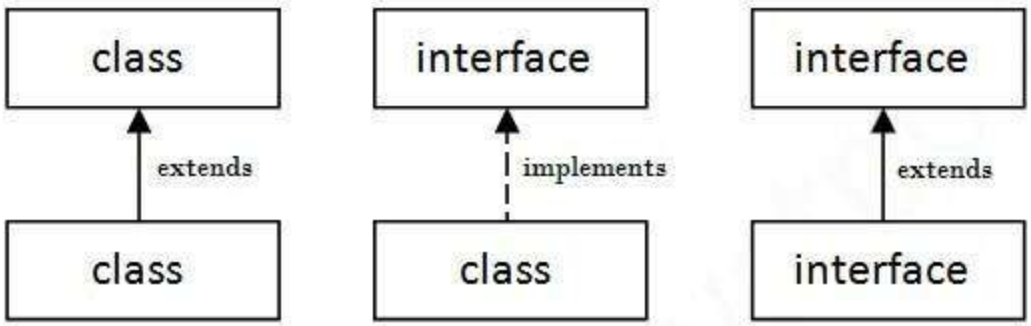
> ***The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.***

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
6.
7. public static void main(String args[]){
8. A6 obj = new A6();
9. obj.print();
```

```
10.  }
11. }
```

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
1.  //Interface declaration: by first user
2.  interface Drawable{
3.  void draw();
4.  }
5.  //Implementation: by second user
6.  class Rectangle implements Drawable{
7.  public void draw(){System.out.println("drawing rectangle");}
8.  }
9.  class Circle implements Drawable{
10. public void draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. class TestInterface1{
14. public static void main(String args[]){
15. Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}
```

Output:

```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

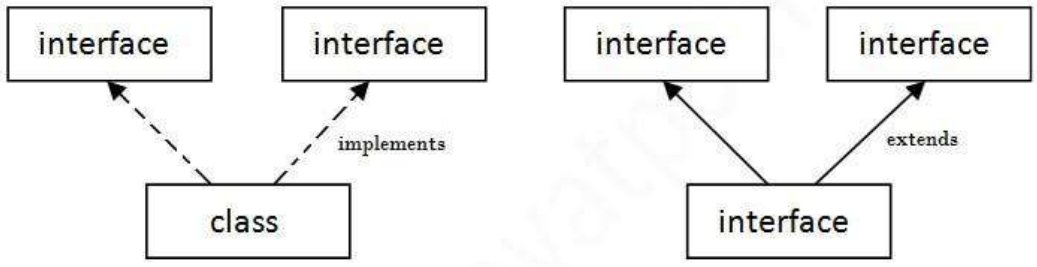
```
1. interface Bank{
2. float rateOfInterest();
3. }
4. class SBI implements Bank{
5. public float rateOfInterest(){return 9.15f;}
6. }
7. class PNB implements Bank{
8. public float rateOfInterest(){return 9.7f;}
9. }
10. class TestInterface2{
11. public static void main(String[] args){
12. Bank b=new SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}
```

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void show();
6. }
7. class A7 implements Printable,Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. A7 obj = new A7();
13. obj.print();
14. obj.show();
15.  }
16. }
```

```
Output:Hello
       Welcome
```

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void print();
6. }
7.
8. class TestInterface3 implements Printable, Showable{
9. public void print(){System.out.println("Hello");}
10. public static void main(String args[]){
11. TestInterface3 obj = new TestInterface3();
12. obj.print();
13.  }
```

```
14. }
```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable extends Printable{
5.  void show();
6.  }
7.  class TestInterface4 implements Showable{
8.  public void print(){System.out.println("Hello");}
9.  public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. TestInterface4 obj = new TestInterface4();
13. obj.print();
14. obj.show();
15.  }
16. }
```

Output:

```
Hello
Welcome
```

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Output:

```
drawing rectangle
default method
```

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Output:

```
drawing rectangle
27
```

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
1. //How Serializable interface is written?
2. public interface Serializable{
3. }
```

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the nested classes chapter. For example:

```
1. interface printable{
2.  void print();
3.  interface MessagePrintable{
4.    void msg();
5.  }
6. }
```

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

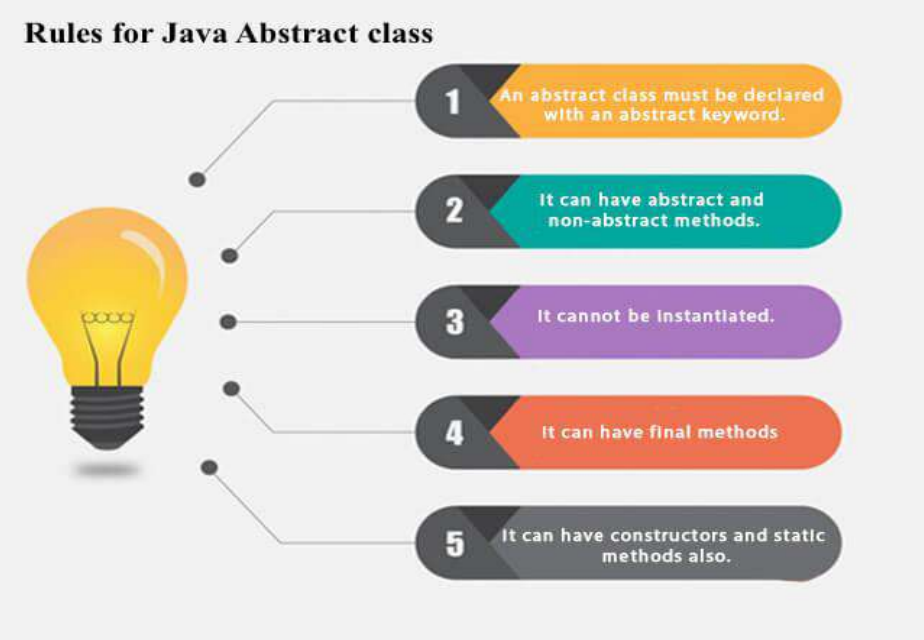
1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.



Syntax of abstract class

```
abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

```
running safely
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8. class Circle1 extends Shape{
9. void draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. class TestAbstraction1{
13. public static void main(String args[]){
14. Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
15. s.draw();
16. }
17. }
```

```
drawing circle
```

Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

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

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

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

What is Exception Handling?

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

Advantage of Exception Handling

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

```
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;
```

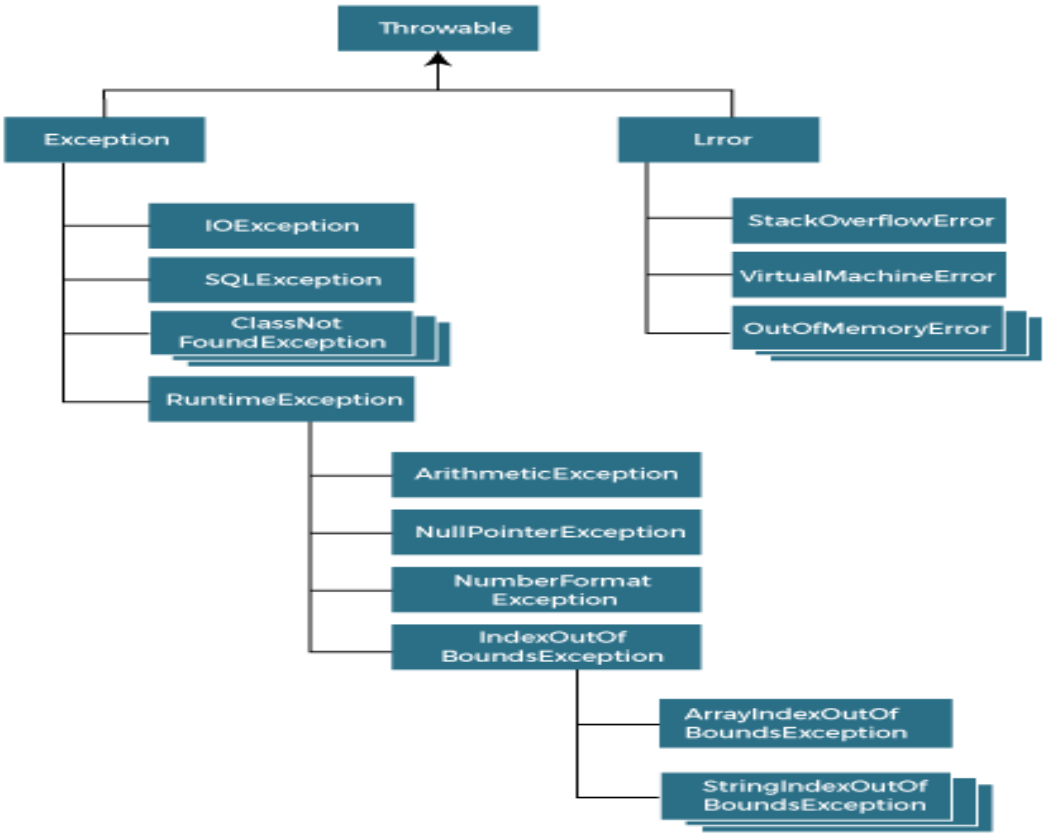
Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling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

Do You Know?

- What is the difference between checked and unchecked exceptions?
- What happens behind the code int data=50/0;?
- Why use multiple catch block?
- Is there any possibility when the finally block is not executed?
- What is exception propagation?
- What is the difference between the throw and throws keyword?
- What are the 4 rules for using exception handling with method overriding?

Hierarchy of Java Exception classes

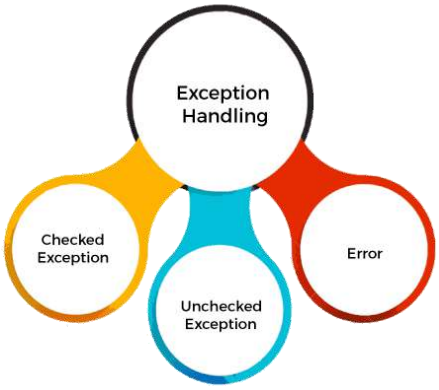
The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

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

1. Checked Exception
2. Unchecked Exception
3. Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Java Exception Keywords

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

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

Java Exception Handling Example

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

JavaExceptionExample.java

```
1. public class JavaExceptionExample{
2.   public static void main(String args[]){
3.    try{
4.       //code that may raise exception
5.       int data=100/0;
6.    }catch(ArithmeticException e){System.out.println(e);}
7.    //rest code of the program
8.    System.out.println("rest of the code...");
9.   }
10. }
```

Output:

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

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

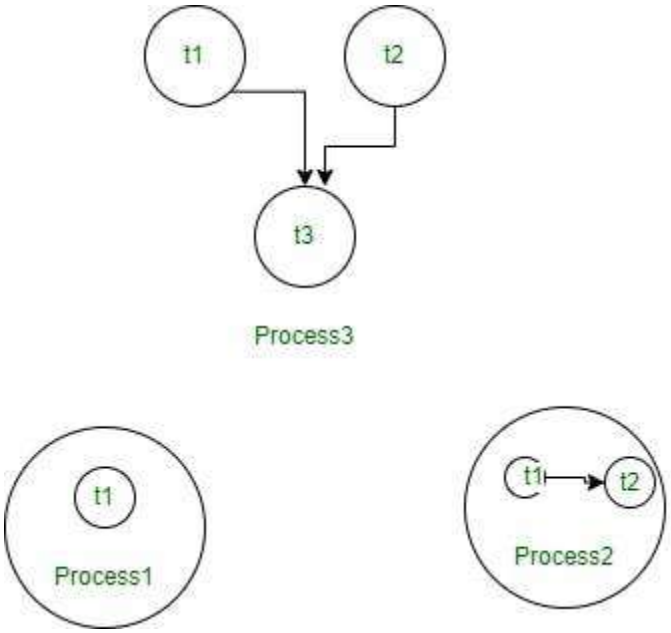
4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Threads

Typically, we can define threads as a subprocess with lightweight with the smallest unit of processes and also has separate paths of execution. These threads use shared memory but they act independently hence if there is an exception in threads that do not affect the working of other threads despite them sharing the same memory.



Threads in a Shared Memory Environment in OS

As we can observe in, the above diagram a thread runs inside the process and there will be context-based switching between threads there can be multiple processes running in OS, and each process again can have multiple threads running simultaneously. The Multithreading concept is popularly applied in games, animation…etc.

The Concept Of Multitasking

To help users Operating System accommodates users the privilege of multitasking, where users can perform multiple actions simultaneously on the machine. This Multitasking can be enabled in two ways:

1. **Process-Based Multitasking**
2. **Thread-Based Multitasking**

1. Process-Based Multitasking (Multiprocessing)

In this type of Multitasking, processes are heavyweight and each process was allocated by a separate memory area. And as the process is heavyweight the cost of communication between processes is high and it takes a long time for switching between processes as it involves actions such as loading, saving in registers, updating maps, lists, etc.

2. Thread-Based Multitasking

As we discussed above Threads are provided with lightweight nature and share the same address space, and the cost of communication between threads is also low.

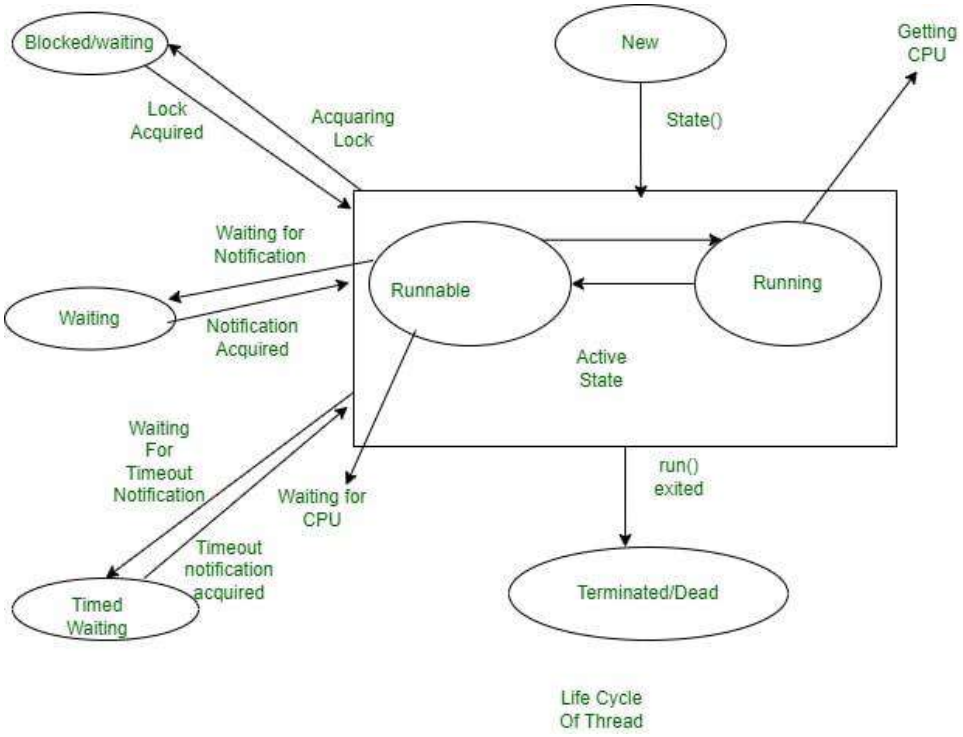
Why Threads are used?

Now, we can understand why threads are being used as they had the advantage of being lightweight and can provide communication between multiple threads at a Low Cost contributing to effective multi-tasking within a shared memory environment.

Life Cycle Of Thread

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State



We can see the working of different states in a Thread in the above Diagram, let us know in detail each and every state:

1. New State

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is not yet initiated.

2. Active State

A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states namely:

- **Runnable State:** In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.
- **Running State:** When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the "Runnable" state to the "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

3. Waiting/Blocked State

If a Thread is inactive but on a temporary time, then either it is a waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 needs to communicate to the camera and the other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

4. Timed Waiting State

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing a Critical Coding operation and if it does not exist the CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

5. Terminated State

A thread will be in Terminated State, due to the below reasons:

- Termination is achieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions…etc. and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longer available.

What is Main Thread?

As we are familiar, we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM, Similarly in this Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM, hence whenever a program is being created in java, JVM provides the Main Thread for its Execution.

How to Create Threads using Java Programming Language?

We can create Threads in java using two ways, namely:

1. Extending Thread Class
2. Implementing a Runnable interface

1. By Extending Thread Class

We can run Threads in Java by using Thread Class, which provides constructors and methods for creating and performing operations on a Thread, which extends a Thread class that can implement Runnable Interface. We use the following constructors for creating the Thread:

- Thread
- Thread(Runnable r)
- Thread(String name)
- Thread(Runnable r, String name)

Sample code to create Threads by Extending Thread Class:

Java

```
import java.io.*;
import java.util.*;
public class GFG extends Thread {
    // initiated run method for Thread
    public void run()
    {
        System.out.println("Thread Started Running...");
    }
    public static void main(String[] args)
    {
        GFG g1 = new GFG();
         // Invoking Thread using start() method
        g1.start();
    }
}
```

Output

```
Thread Started Running...
```

Runnable interface

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

public class GFG implements Runnable {
    // method to start Thread
    public void run()
    {
        System.out.println(
            "Thread is Running Successfully");
```

```
    }

    public static void main(String[] args)
    {
        GFG g1 = new GFG();
        // initializing Thread Object
        Thread t1 = new Thread(g1);
        t1.start();
    }
}
```

Output

```
Thread is Running Successfully
```

Sample Code to create Thread in Java using Thread(String name):

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

public class GFG {
    public static void main(String args[])
    {
        // Thread object created
        // and initiated with data
        Thread t = new Thread("Hello Geeks!");

        // Thread gets started
        t.start();

        // getting data of
        // Thread through String
        String s = t.getName();
        System.out.println(s);
    }
}
```

Output

```
Hello Geeks!
```

What is Multithreading in Java?

Multithreading is a feature in Java that **concurrently executes two or more parts of the program for utilizing the CPU at its maximum**. The part of each program is called Thread which is a lightweight process.

According to the definition, it can be deduced that it expands the concept of multitasking in the program by allowing certain operations to be divided into smaller units using a single application.

Each Thread operates concurrently and permits the execution of multiple tasks inside the same application.

Multithreading vs. Multiprocessing in Java

Multithreading	Multiprocessing
In this, **multiple threads are created** for increasing computational power using a single process.	In this, **CPUs are added** in order to increase computational power.
Many **threads** of a process are executed simultaneously.	Many **processes** are executed simultaneously.
It is **not classified** into any categories.	**Classified** into two categories, symmetric and asymmetric.
The creation of a process is **economical**.	Creation of a process is **time-consuming**.
In this, a **common space of address** is shared by all threads	Every process in this owns a **separate space of address**.

What is the Use of Multi-Thread in Java?

1. Because each Thread is managed individually and several operations can be carried out at once, **the user is not blocked**.
2. It is used to **save time** as multiple operations are performed concurrently.
3. Since threads are **independent**, other threads don't get affected even if an exception occurs in a single thread.

Examples of Multithreading in java

Implementation of Multithreading using java:

Example 1: To facilitate thread programming, Java offers the Thread class. To generate and manage threads, the Thread class offers constructors and methods. The Runnable interface is implemented by the Thread class, which extends the Object class.

```
class MultithreadingTest implements Runnable {

 // run method to execute the thread

 public void run()

 {

  try {

   // Displaying the running Thread

   System.out.println(

    "Thread " + Thread.currentThread().getId()

    + " is running");
```

```
    }
    catch (Exception e) {
      // exception is caught if occurred
      System.out.println("Exception has occurred and is caught");
    }
  }
}
class MultithreadMain {
  public static void main(String[] args)
  {
    int n = 6;        // Number of threads
    // Creating and starting n number of threads
    for (int i = 0; i < n; i++) {
      Thread obj
        = new Thread(new MultithreadingTest());
      obj.start();
    }
  }
}
```

Output:

```
Thread 13 is running
Thread 14 is running
Thread 11 is running
Thread 10 is running
Thread 12 is running
Thread 15 is running
```

References:

1. https://www.javatpoint.com/history-of-java
2. https://www.javatpoint.com/features-of-java
3. https://www.javatpoint.com/cpp-vs-java
4. https://www.geeksforgeeks.org/variables-in-java/
5. https://www.javatpoint.com/final-keyword
6. https://www.javatpoint.com/array-in-java
7. https://www.javatpoint.com/class-definition-in-java
8. https://www.javatpoint.com/class-definition-in-java
9. https://www.tutorialspoint.com/java/java_constructors.htm
10. https://www.javatpoint.com/inheritance-in-java
11. https://www.javatpoint.com/abstract-class-in-java
12. https://www.javatpoint.com/exception-handling-in-java
13. https://www.geeksforgeeks.org/java-threads/
14. https://www.scaler.com/topics/multithreading-in-java/