Unit 1. Introduction to Java

What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Java Example

Let's have a quick look at Java programming example. A detailed description of Hello Java example is available in next page.

Simple.java

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

Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

Unit 1. Introduction to Java

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3) Java ME (Java Micro Edition)

It is a micro platform that is dedicated to mobile applications.

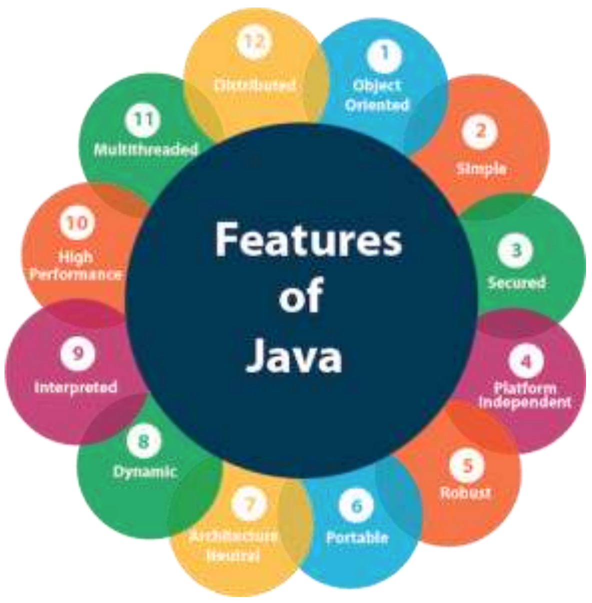
4) JavaFX

It is used to develop rich internet applications. It uses a lightweight user interface API.

1.1 Properties or Features of Java OR Java buzzwords

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 the most important features of the 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

Unit 1. Introduction to Java

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, 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.

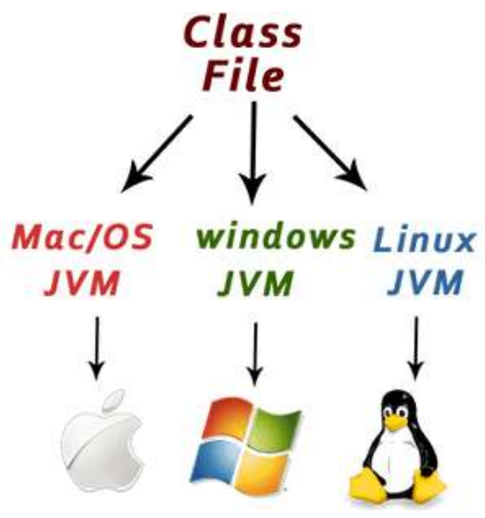
Object-oriented

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 incorporate both data and behavior. 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



Unit 1. Introduction to Java

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 top of other hardware-based platforms. It has two components:

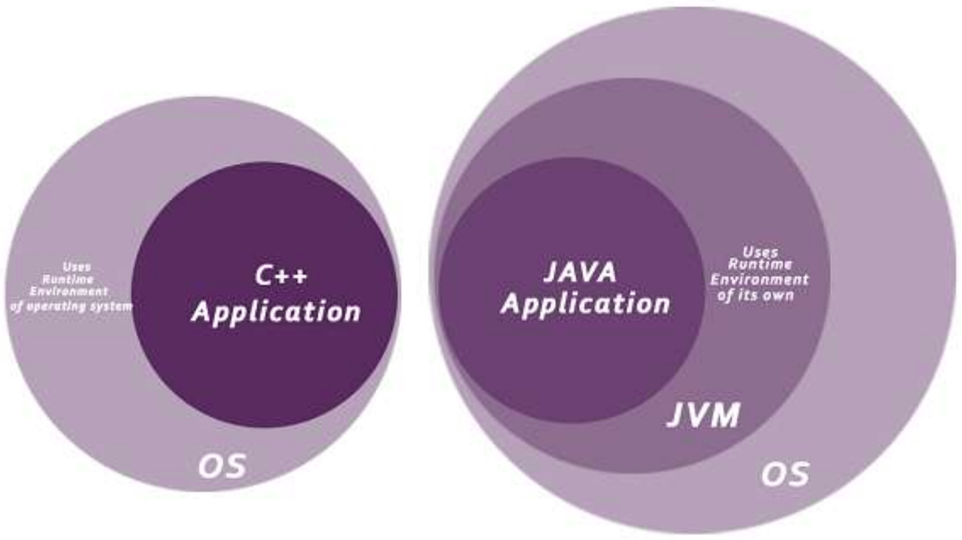
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed 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).

Secured

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**


Uses Runtime Environment of operating system
C++ Application
OS
JAVA Application
Uses Runtime Environment of its own
JVM
OS

- **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 rights to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Unit 1. Introduction to Java

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

Robust

The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection 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 the 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).

1.2 Comparison of java with C++

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 Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the 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 to be 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 using interfaces in java.

Unit 1. Introduction to Java

Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a 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 both compiler and interpreter. 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 comments.	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 to 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++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the 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.

Unit 1. Introduction to Java

Object-oriented	C++ is an object-oriented language. However, in the C language, a 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.

C++ Program Example

File: main.cpp

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello C++ Programming";
  return 0;
}
```

Output:

```
Hello C++ Programming
```

Java Program Example

File: Simple.java

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

Output:

```
Hello Java
```

Unit 1. Introduction to Java

First Java Program | Hello World Example

In this section, we will learn how to write the simple program of Java. We can write a simple hello Java program easily after installing the JDK.

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

The requirement for Java Hello World Example

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. http://www.javatpoint.com/how-to-set-path-in-java
- Create the Java program
- Compile and run the Java program

Creating Hello World Example

Let's create the hello java program:

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

Save the above file as Simple.java.

To compile:	javac Simple.java
To execute:	java Simple

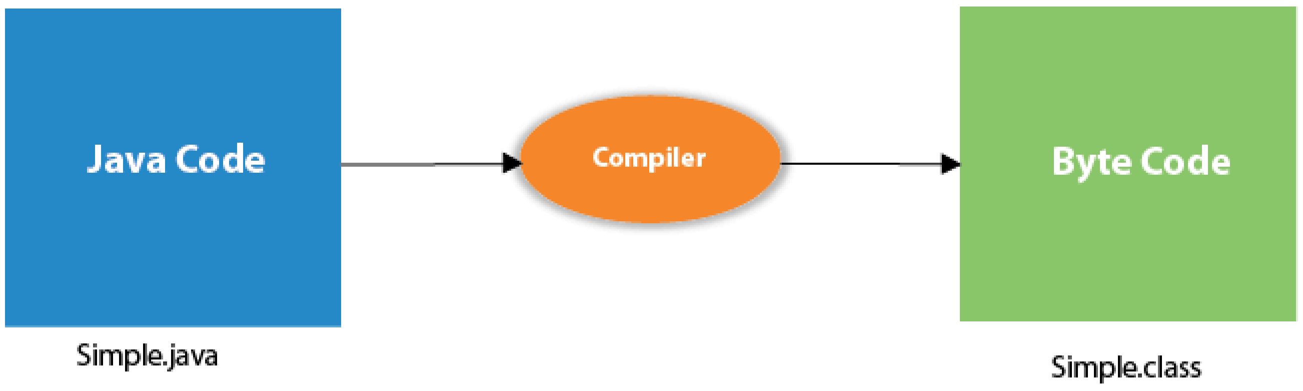
Output:

```
Hello Java
```

Unit 1. Introduction to Java

1.3 Java Compiler, Java Interpreter :

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



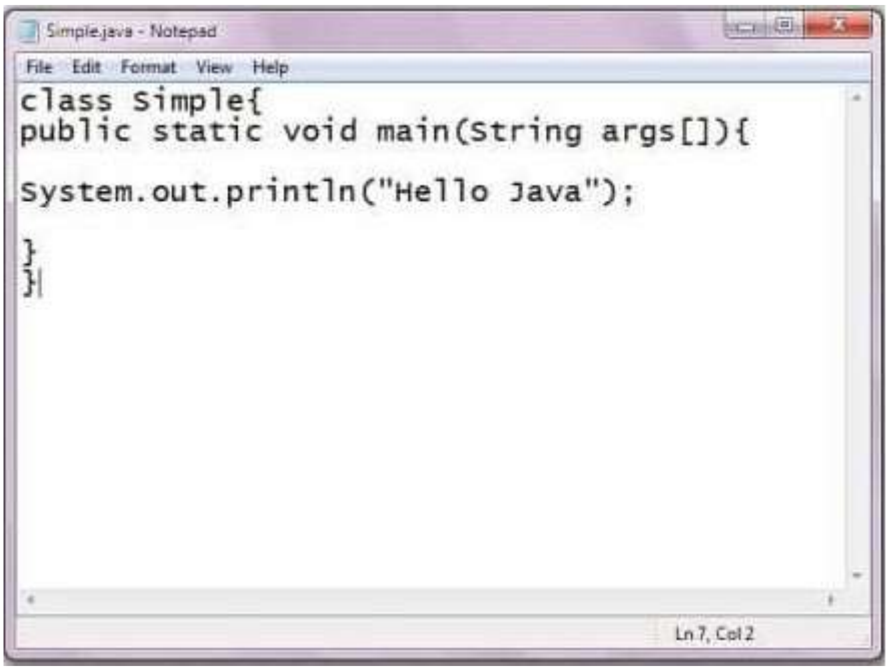
Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

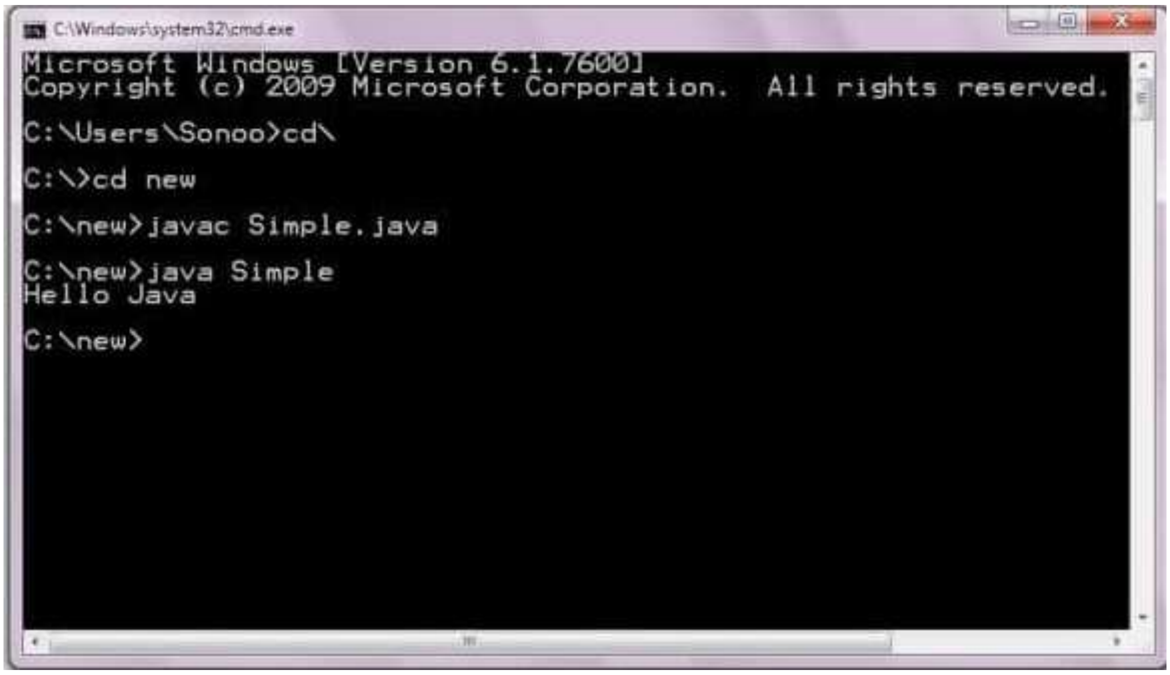
To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> Notepad** and write a simple program as we have shownbelow:

Unit 1. Introduction to Java



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

System.out.println("Hello Java");

}
}
```

As displayed in the above diagram, write the simple program of Java in notepad and saved it as Simple.java. In order to compile and run the above program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**. When we have done with all the steps properly, it shows the following output:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

To compile and run the above program, go to your current directory first; my current directory is c:\new. Write here:

To compile:	javac Simple.java
To execute:	java Simple

Unit 1. Introduction to Java

In how many ways we can write a Java program?

There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

1) By changing the sequence of the modifiers, method prototype is not changed in Java.

Let's see the simple code of the main method.

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

2) The subscript notation in the Java array can be used after type, before the variable or after the variable.

Let's see the different codes to write the main method.

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

3) You can provide var-args support to the main() method by passing 3 ellipses (dots)

Let's see the simple code of using var-args in the main() method. We will learn about var-args later in the Java New Features chapter.

```
public static void main(String... args)
```

4) Having a semicolon at the end of class is optional in Java.

Let's see the simple code.

```
class A{
static public void main(String... args){
System.out.println("hello java4");
}
};
```

Valid Java main() method signature

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

Unit 1. Introduction to Java

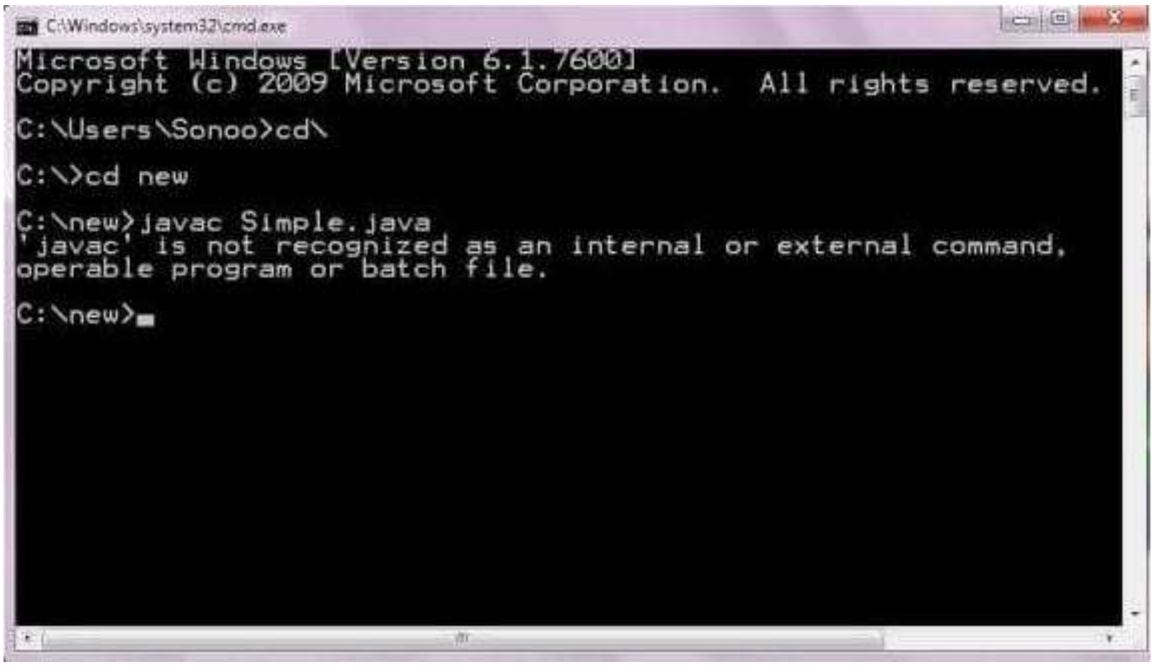
5. **static public void** main(String[] args)
6. **public static final void** main(String[] args)
7. **final public static void** main(String[] args)
8. **final strictfp public static void** main(String[] args)

Invalid Java main() method signature

1. **public void** main(String[] args)
2. **static void** main(String[] args)
3. **public void static** main(String[] args)
4. **abstract public static void** main(String[] args)

Resolving an error "javac is not recognized as an internal or external command"?

If there occurs a problem like displayed in the below figure, you need to set a path. Since DOS doesn't recognize javac and java as internal or external command. To overcome this problem, we need to set a path. The path is not required in a case where you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path. Click here for How to set path in java.



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\new>
```

Unit 1. Introduction to Java

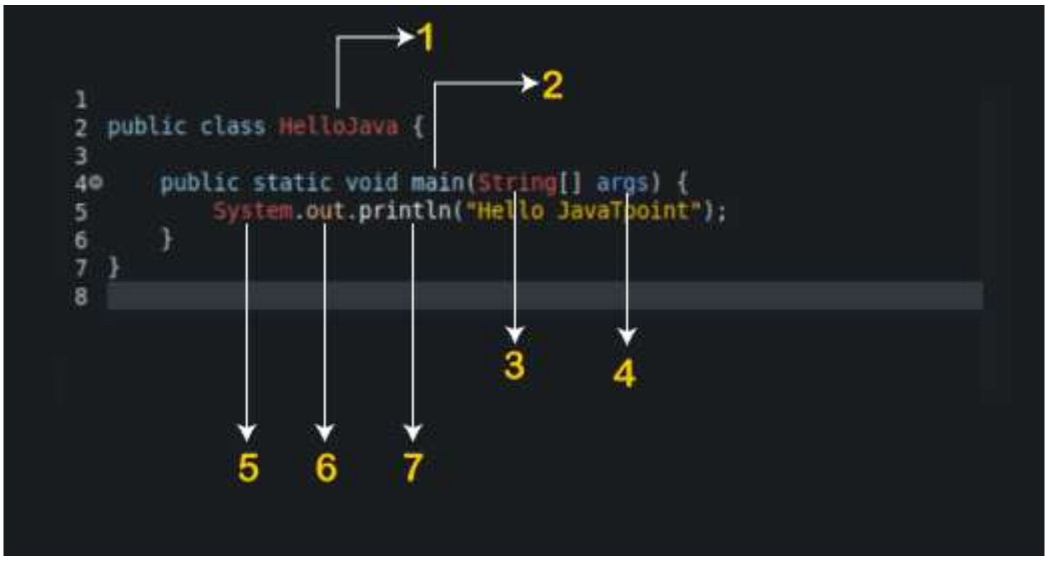
Identifiers in Java

Identifiers in Java are symbolic names used for identification. They can be a class name, variable name, method name, package name, constant name, and more. However,

In Java, There are some reserved words that can not be used as an identifier.

For every identifier there are some conventions that should be used before declaring them. Let's understand it with a simple Java program:

```
public class HelloJava {

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



```

public class HelloJava {

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

```

From the above example, we have the following Java identifiers:

1. HelloJava (Class name)
2. main (main method)
3. String (Predefined Class name)
4. args (String variables)
5. System (Predefined class)
6. out (Variable name)
7. println (method)

Unit 1. Introduction to Java

let's understand the rules for Java identifier:

Rules for Identifiers in Java

There are some rules and conventions for declaring the identifiers in Java. If the identifiers are not properly declared, we may get a compile-time error. Following are some rules and conventions for declaring identifiers:

- A valid identifier must have characters [A-Z] or [a-z] or numbers [0-9], and underscore(_) or a dollar sign ($). for example, @javatpoint is not a valid identifier because it contains a special character which is @.
- There should not be any space in an identifier. For example, java tpoint is an invalid identifier.
- An identifier should not contain a number at the starting. For example, 123javatpoint is an invalid identifier.
- An identifier should be of length 4-15 letters only. However, there is no limit on its length. But, it is good to follow the standard conventions.
- We can't use the Java reserved keywords as an identifier such as int, float, double, char, etc. For example, int double is an invalid identifier in Java.
- An identifier should not be any query language keywords such as SELECT, FROM, COUNT, DELETE, etc.

Literals in Java

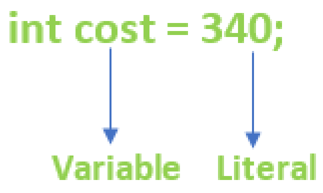
In Java

, **literal** is a notation that represents a fixed value in the source code. In lexical analysis, literals of a given type are generally known as **tokens**

. In this section, we will discuss the term **literals in Java**.

Literals

In Java, **literals** are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals. The following figure represents a literal.

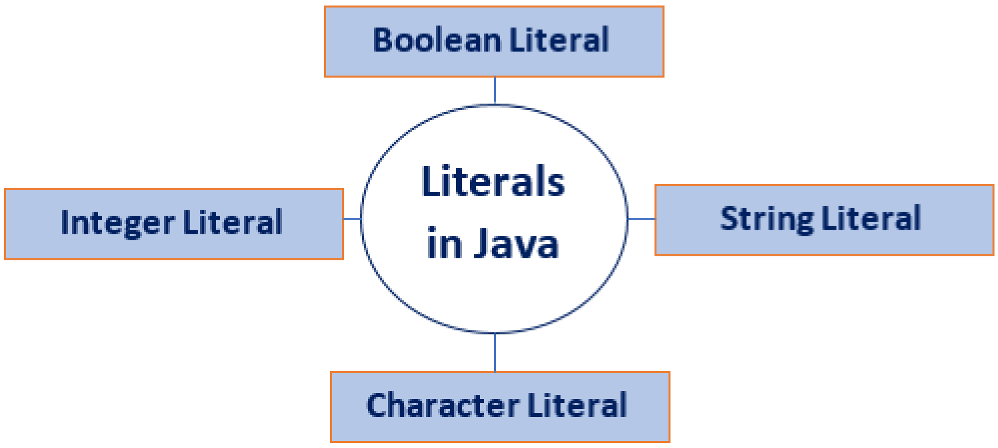


Types of Literals in Java

There are the majorly **four** types of literals in Java:

Unit 1. Introduction to Java

1. Integer Literal
2. Character Literal
3. Boolean Literal
4. String Literal



Integer Literals

Integer literals are sequences of digits. There are three types of integer literals:

- **Decimal Integer:** These are the set of numbers that consist of digits from 0 to 9. It may have a positive (**+)** or negative (**-**) Note that between numbers commas and non-digit characters are not permitted. For example, **5678, +657, -89,** etc.

  ```
  int decVal = 26;
  ```

- **Octal Integer:** It is a combination of number have digits from 0 to 7 with a leading 0. For example, **045, 026,**

  ```
  int octVal = 067;
  ```

- **Hexa-Decimal:** The sequence of digits preceded by **0x** or **0X** is considered as hexadecimal integers. It may also include a character from **a** to **f** or **A** to **F** that represents numbers from **10** to **15**, respectively. For example, **0xd, 0xf,**

  ```
  int hexVal = 0x1a;
  ```

- **Binary Integer:** Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later). Prefix 0b represents the Binary system. For example, 0b11010.

  ```
  int binVal = 0b11010;
  ```

Unit 1. Introduction to Java

Real Literals

The numbers that contain fractional parts are known as real literals. We can also represent real literals in exponent form. For example, **879.90, 99E-3,** etc.

Backslash Literals

Java supports some special backslash character literals known as backslash literals. They are used in formatted output. For example:

\n: It is used for a new line

\t: It is used for horizontal tab

\b: It is used for blank space

\v: It is used for vertical tab

\a: It is used for a small beep

\r: It is used for carriage return

\': It is used for a single quote

\": It is used for double quotes

Character Literals

A character literal is expressed as a character or an escape sequence, enclosed in a **single** quote ('') mark. It is always a type of char. For example, **'a', '%', '\u000d',** etc.

String Literals

String literal is a sequence of characters that is enclosed between **double** quotes ("") marks. It may be alphabet, numbers, special characters, blank space, etc. For example, "**Jack", "12345", "\n",** etc.

Floating Point Literals

The vales that contain decimal are floating literals. In Java, float and double primitive types fall into floating-point literals. Keep in mind while dealing with floating-point literals.

- Floating-point literals for float type end with F or f. For example, **6f, 8.354F**, etc. It is a **32**-bit float literal.
- Floating-point literals for double type end with D or d. It is optional to write D or d. For example, **6d, 8.354D,** etc. It is a **64**-bit double literal.
- It can also be represented in the form of the **exponent**.

Floating:

```
float length = 155.4f;
```

Decimal:

```
double interest = 99658.445;
```

Decimal in Exponent form:

```
double val= 1.234e2;
```

Boolean Literals

Boolean literals are the value that is either true or false. It may also have values 0 and 1. For example, **true, 0,** etc.

```
boolean isEven = true;
```

Null Literals

Null literal is often used in programs as a marker to indicate that reference type object is unavailable. The value **null** may be assigned to any variable, except variables of primitive types.

```
String stuName = null;

Student age = null;
```

Class Literals

Class literal formed by taking a type name and appending **.class** extension. For example, **Scanner.class**. It refers to the object (of type Class) that represents the type itself.

```
class classType = Scanner.class;
```

Invalid Literals

There is some invalid declaration of literals.

```
float g = 6_.674f;

float g = 6._674F;
long phoneNumber = 99_00_99_00_99_L;
int x = 77_;
int y = 0_x76;
int z = 0X_12;
int z = 0X12_;
```

Unit 1. Introduction to Java

Restrictions to Use Underscore (_)

- It can be used at the beginning, at the end, and in-between of a number.
- It can be adjacent to a decimal point in a floating-point literal.
- Also, can be used prior to an F or L suffix.
- In positions where a string of digits is expected.

Why use literals?

To avoid defining the constant somewhere and making up a label for it. Instead, to write the value of a constant operand as a part of the instruction.

How to use literals?

A literal in Java can be identified with the prefix **=**, followed by a specific value.

Let's create a Java program and use above discussed literals.

LiteralsExample.java

```
public class LiteralsExample

{
public static void main(String args[])
{
int count = 987;
float floatVal = 4534.99f;
double cost = 19765.567;
int hexaVal = 0x7e4;
int binary = 0b11010;
char alpha = 'p';
String str = "Java";
boolean boolVal = true;
int octalVal = 067;
String stuName = null;
char ch1 = '\u0021';
char ch2 = 1456;
System.out.println(count);
System.out.println(floatVal);
System.out.println(cost);
System.out.println(hexaVal);
System.out.println(binary);
System.out.println(alpha);
System.out.println(str);
System.out.println(boolVal);
System.out.println(octalVal);
System.out.println(stuName);
System.out.println(ch1);
System.out.println("\t" +"backslash literal");
System.out.println(ch2);
}
}
```

Output:

```
987
4534.99
19765.567
2020
26
p
Java
true
55
null
!
        backslash literal
?
```

Operators in Java

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

Unit 1. Introduction to Java

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.: rementing/decrementing a value by one

- negating an expression
- inverting the value of a boolean

Unit 1. Introduction to Java

Java Unary Operator Example: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21


}}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

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

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

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

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

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

Output:

```
21
```

Unit 1. Introduction to Java

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

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

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

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

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{
public static void main(String args[]){
  //For positive number, >> and >>> works same
  System.out.println(20>>2);
  System.out.println(20>>>2);
  //For negative number, >>> changes parity bit (MSB) to 0
```

```
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}}
```

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.

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

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

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

Unit 1. Introduction to Java

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.

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

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

```
public class OperatorExample{

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

Unit 1. Introduction to Java

```
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Another Example:

```
public class OperatorExample{

public static void main(String args[]){
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

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

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
```

Unit 1. Introduction to Java

```
16
```

Java Assignment Operator Example

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

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

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

Output:

```
Compile time error
```

After type cast:

```
public class OperatorExample{

public static void main(String args[]){
short a=10;
```

Unit 1. Introduction to Java

```
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

```
20
```

Java Variables

A variable is a container which holds the value while the Java program

is executed. A variable is assigned with a data type.

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

Variable

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

```
int data=50;//Here data is variable
```

Types of Variables

There are three types of variables in Java

- local variable
- instance variable
- static variable

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.

Unit 1. Introduction to Java

2) Instance Variable

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

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

3) Static variable

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

Example to understand the types of variables in java

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

Java Variable Example: Add Two Numbers

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

Output:

```
20
```

Unit 1. Introduction to Java

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords

A list of Java keywords or reserved words are given below:

1. **abstract:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:**Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case:** Java case keyword is used with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

Unit 1. Introduction to Java

19. **for**: Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if**: Java if keyword tests the condition. It executes the if block if the condition is true.
21. **implements**: Java implements keyword is used to implement an interface.
22. **import**: Java import keyword makes classes and interfaces available and accessible to the current source code.
23. **instanceof**: Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int**: Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface**: Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long**: Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native**: Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new**: Java new keyword is used to create new objects.
29. **null**: Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package**: Java package keyword is used to declare a Java package that includes the classes.
31. **private**: Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected**: Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. **public**: Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return**: Java return keyword is used to return from a method when its execution is complete.
35. **short**: Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. **static**: Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. **strictfp**: Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super**: Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. **switch**: The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized**: Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this**: Java this keyword can be used to refer the current object in a method or constructor.

Unit 1. Introduction to Java

42. **throw**: The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. **throws**: The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. **transient**: Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try**: Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. void**:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile**: Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while**: Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

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.

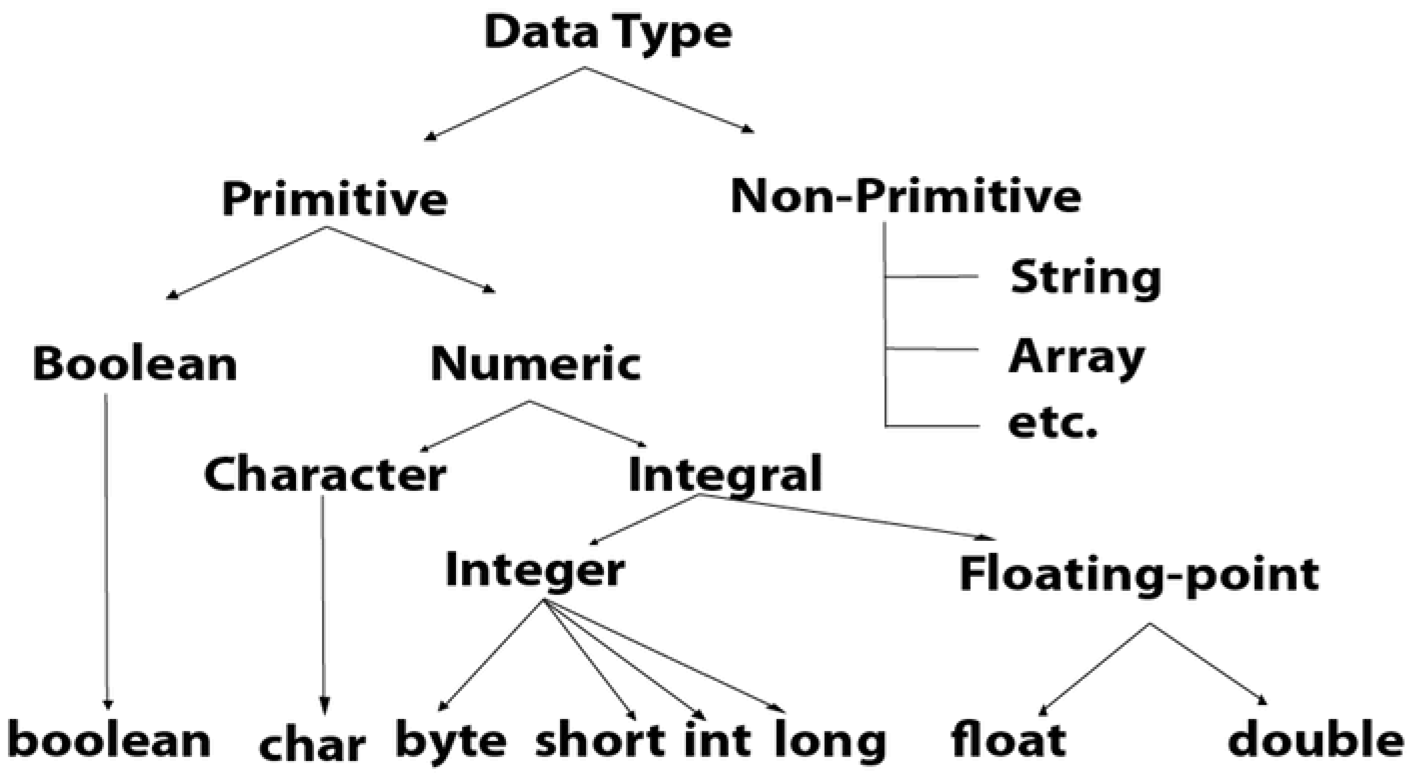
Java Primitive Data Types

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

Unit 1. Introduction to Java



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

Boolean Data Type

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

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

Example:

Boolean one = **false**

Byte Data Type

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

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

Example:

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

Short Data Type

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

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

Example:

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

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:

int a = 100000, **int** b = -200000

Long Data Type

Unit 1. Introduction to Java

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

Example:

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

Float Data Type

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

Example:

```
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:

```
double d1 = 12.3
```

Char Data Type

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

Example:

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

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Unit 1. Introduction to Java

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

Unit 1. Introduction to Java

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:

```
if(condition) {

statement 1; //executes when condition is true


}
```

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

Student.java

Student.java

```
public class Student {

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

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement

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

Syntax:

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

Unit 1. Introduction to Java

```
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

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

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:

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

Unit 1. Introduction to Java

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
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:

```
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.

Unit 1. Introduction to Java

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.
-

Unit 1. Introduction to Java

syntax:

```
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
```

Unit 1. Introduction to Java

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

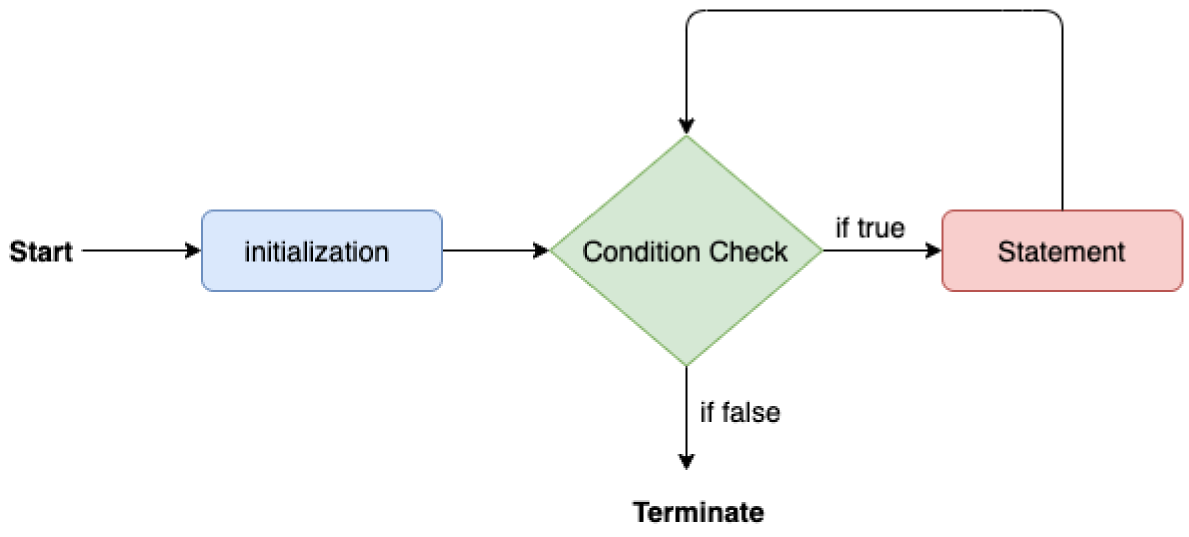
In Java, for loop is similar to C and C++

. 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.

Syntax:

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

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



Unit 1. Introduction to Java

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) {
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.

Syntax:

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

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

Calculation.java

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

Output:

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

Java while loop

The while loop

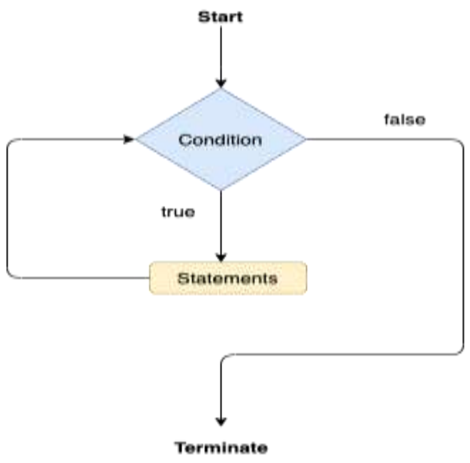
is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

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.

Syntax:

```
while(condition){
//looping statements
}
```

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



Consider the following example.

Calculation .java

```
public class Calculation {
public static void main(String[] args) {
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
```

```
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

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

Java do-while loop

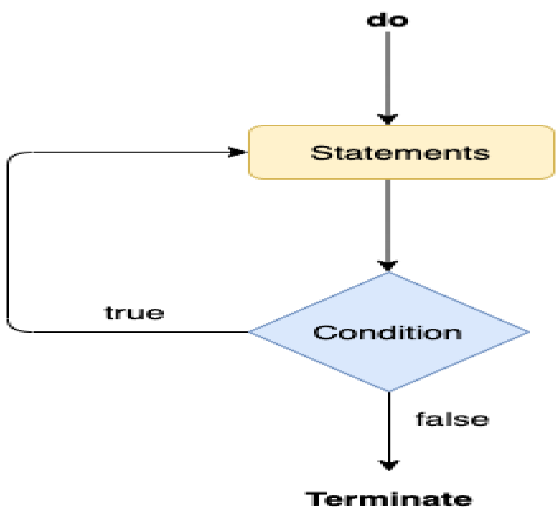
The do-while loop

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.

Syntax:

```
do
{
//statements
} 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.

Unit 1. Introduction to Java

Calculation.java

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

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.

Unit 1. Introduction to Java

BreakExample.java

```
public class BreakExample {
public static void main(String[] args) {
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

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

break statement example with labeled for loop

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
a:
for(int i = 0; i<= 10; i++) {
b:
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}
}
}
}
}
```

Unit 1. Introduction to Java

```
 }
```

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.

```
public class ContinueExample {

public static void main(String[] args) {
for(int i = 0; i<= 2; i++) {
for (int j = i; j<=5; j++) {
if(j == 4) {
continue;
}
System.out.println(j);
}
}
}
  }
```

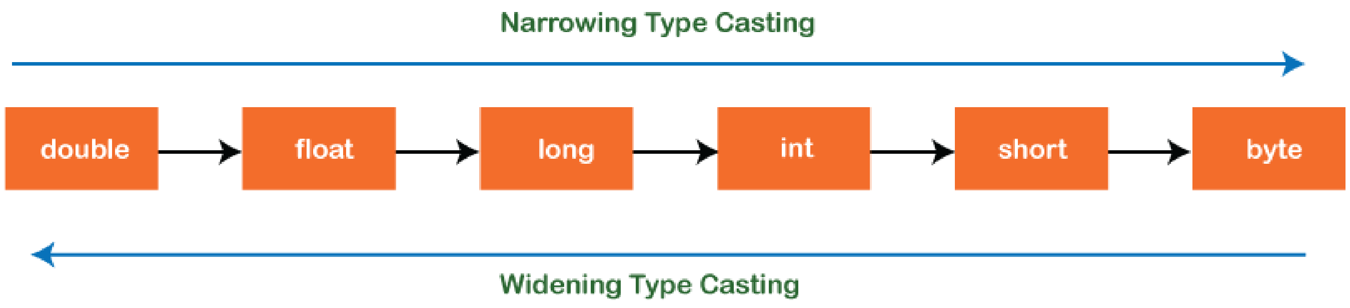
Output:

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

Unit 1. Introduction to Java

Type Casting in Java

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type Casting in Java

Type casting

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- The target type must be larger than the source type.

byte -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

Unit 1. Introduction to Java

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

Unit 1. Introduction to Java

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

Unit 2. Classes and Objects

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

Unit 2. Classes and Objects

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

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

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
```

Unit 2. Classes and Objects

```
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

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

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Unit 2. Classes and Objects

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

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

```
101 Sonoo
```

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

File: TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
```

Unit 2. Classes and Objects

```
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

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

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation()
{
System.out.println(rollno+" "+name);
}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
```

Unit 2. Classes and Objects

```
 Student s2=new Student();
 s1.insertRecord(111,"Karan");
 s2.insertRecord(222,"Aryan");
 s1.displayInformation();
 s2.displayInformation();
}
}
```

Output:

```
111 Karan
222 Aryan
```

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

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
```

Unit 2. Classes and Objects

```
    e1.display();
    e2.display();
    e3.display();
}
}
```

Output:

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

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;
```

Unit 2. Classes and Objects

```
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.depssosit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}
```

Output:

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

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.

Unit 2. Classes and Objects

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.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
```

Unit 2. Classes and Objects

```
 public class Simple{
 public static void main(String args[]){
 A obj=new A();
 System.out.println(obj.data);//Compile Time Error
 obj.msg();//Compile Time Error
 }
 }
```

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:

```
 class A{
 private A(){}//private constructor
 void msg(){System.out.println("Hello java");}
 }
 public class Simple{
  public static void main(String args[]){
    A obj=new A();//Compile Time Error
  }
 }
```

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.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

Unit 2. Classes and Objects

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

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.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
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.

```
class A{
protected void msg(){System.out.println("Hello java");}
}
```

Unit 2. Classes and Objects

```
public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
public static void main(String args[]){
Simple obj=new Simple();
obj.msg();
}
}
```

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

Constructors in Java

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.

Rules for creating Java constructor

There are following 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-argument constructor)
2. Parameterized constructor

Unit 2. Classes and Objects

1. Java Default Constructor

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

Syntax of default constructor:

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

Example of default constructor

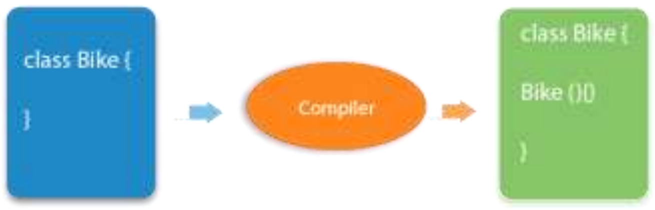
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

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

Output:

```
Bike is created
```

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



Example of default constructor that displays the default values

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
int id;
String name;
//method to display the value of id and name
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}
```

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.

2. 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.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
int id;
String name;
//creating a parameterized constructor
Student4(int i,String n){
id = i;
```

Unit 2. Classes and Objects

```
name = n;
}
//method to display the values
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
//creating objects and passing values
Student4 s1 = new Student4(111,"Karan");
Student4 s2 = new Student4(222,"Aryan");
//calling method to display the values of object
s1.display();
s2.display();
}
}
```

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

```
//Java program to overload constructors
class Student5{
int id;
String name;
int age;
//creating two arg constructor
Student5(int i,String n){
id = i;
name = n;
```

Unit 2. Classes and Objects

```
}
//creating three arg constructor
Student5(int i,String n,int a){
id = i;
name = n;
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();
}
}
```

Output:

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

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Unit 2. Classes and Objects

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
//Java program to initialize the values from one object to another object.
class Student6{
int id;
String name;
//constructor to initialize integer and string
Student6(int i,String n){
id = i;
name = n;
}
//constructor to initialize another object
Student6(Student6 s){
id = s.id;
name =s.name;
}
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student6 s1 = new Student6(111,"Karan");
Student6 s2 = new Student6(s1);
s1.display();
s2.display();
}
}
```

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{
int id;
String name;
Student7(int i,String n){
id = i;
name = n;
}
Student7(){}
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student7 s1 = new Student7(111,"Karan");
Student7 s2 = new Student7();
s2.id=s1.id;
s2.name=s1.name;
s1.display();
s2.display();
}
}
```

Output:

```
111 Karan
111 Karan
```

Unit 2. Classes and Objects

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.

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.

Java Inheritance Example

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

```
class Employee{
float salary=40000;
```

Unit 2. Classes and Objects

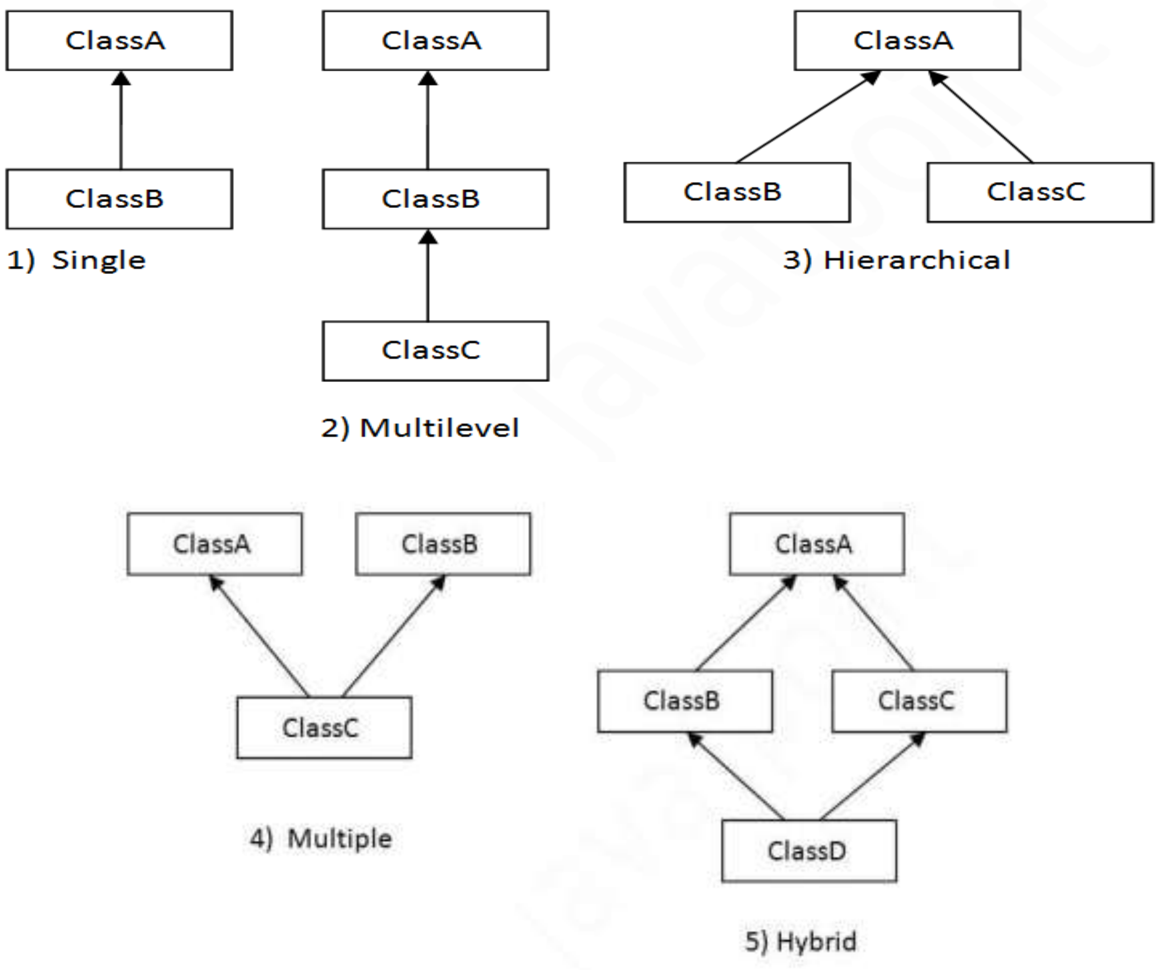
```
}
class Programmer extends Employee{
int bonus=10000;
public static void main(String args[]){
Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary);
System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

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.



Unit 2. Classes and Objects

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

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

```
    class Dog {
    void bark(){System.out.println("barking...");}
    }
    class TestInheritance extends Dog{
    public static void main(String args[]){
    Dog d=new Dog();
    d.bark();
}}
```

Output:

```
barking...
```

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

```
    class Animal{
    void eat(){System.out.println("eating...");}
    }
    class Dog extends Animal{
    void bark(){System.out.println("barking...");}
    }
    class BabyDog extends Dog{
    void weep(){System.out.println("weeping...");}
    }
    class TestInheritance2{
    public static void main(String args[]){
    BabyDog d=new BabyDog();
```

```
d.weep();
d.bark();
d.eat();
}}
```

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

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

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

Unit 2. Classes and Objects

Q) Why multiple inheritance is not supported in java?

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

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

public static void main(String args[]){
C obj=new C();
obj.msg();//Now which msg() method would be invoked?
}
}
```

```
Compile Time Error
```

Polymorphism in Java

- **Polymorphism** is the greek word whose meaning is “same object having different behaviour”.
- Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- We can perform polymorphism in java by method overloading and method overriding.
- **There are two types of polymorphism in Java:**
  1. compile-time polymorphism
  2. runtime polymorphism.

1. Compile time Polymorphism (or Static polymorphism)

Unit 2. Classes and Objects

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.

Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

Example of static Polymorphism:

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class SimpleCalculator
{
   int add(int a, int b)
   {
       return a+b;
   }
   int  add(int a, int b, int c)
   {
       return a+b+c;
   }
}
public class Demo
{
  public static void main(String args[])
  {
      SimpleCalculator obj = new SimpleCalculator();
     System.out.println(obj.add(10, 20));
     System.out.println(obj.add(10, 20, 30));
  }
}
```

Output:

```
30
60
```

2. Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism.

Unit 2. Classes and Objects

Example

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

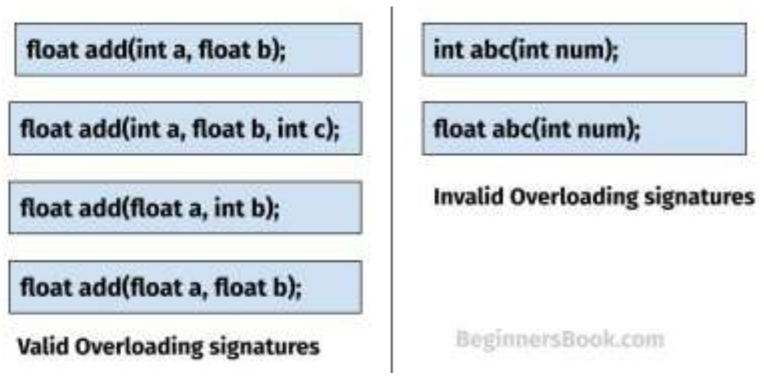
```
class ABC{
  public void myMethod(){
    System.out.println("Overridden Method");
  }
}
public class XYZ extends ABC{

  public void myMethod(){
    System.out.println("Overriding Method");
  }
  public static void main(String args[]){
    ABC obj = new XYZ();
    obj.myMethod();
  }
}
```

Output:

```
Overriding Method
```

Method Overloading

Method Overloading is a feature that allows a class to have multiple methods with the same name but with different number, sequence or type of parameters. In short **multiple methods with same name but with different signatures**. For example the signature of method add(int a, int b) having two int parameters is different from signature of method add(int a, int b, int c) having three int parameters.


float add(int a, float b);
float add(int a, float b, int c);
float add(float a, int b);
float add(float a, float b);
Valid Overloading signatures
int abc(int num);
float abc(int num);
Invalid Overloading signatures
BeginnersBook.com

Unit 2. Classes and Objects

This is one of the most popular OOP feature in java, there are several cases where we need more than one methods with same name. **For example** let's say we are writing a java program to find the sum of input numbers, we need different variants of add method based on the user inputs such as add(int, int), add(float, float) etc.

Three ways to overload a method

In order to overload a method, the parameter list of the methods must differ in either of these:
1. Number of parameters.
For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.
For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.
For example:

```
add(int, float)
add(float, int)
```

Invalid case of method overloading:
Parameters list doesn't mean the return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw a compilation error.

```
int add(int, int)
float add(int, int)
```

Method overloading is an example of Static Polymorphism.

Example: Overloading – Data type of parameters are different

In this example, we are overloading the method add() based on the **data type of parameters**. We have two methods with the name add() but the type of parameters are different. The first variation of add() has two int params while the second variation of method add() has two float params.

```
class DisplayOverloading2
{
  //two int parameters
```

```
    public int add(int a, int b)
    {
      int sum = a+b;
      return sum;
    }
    //two float parameters
    public float add(float a, float b)
    {
      float sum = a+b;
      return sum;
    }
  }

  class JavaExample
  {
    public static void main(String args[])
    {
      DisplayOverloading2 obj = new DisplayOverloading2();
      //This will call the method add with two int params
      System.out.println(obj.add(5, 15));

      //This will call the method add with two float params
      System.out.println(obj.add(5.5f, 2.5f));
    }
  }
```

Output:

```
20
8.0
```

Method overriding

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
  //Overridden method
  public void eat()
  {
    System.out.println("Human is eating");
  }
}
class Boy extends Human{
  //Overriding method
  public void eat(){
    System.out.println("Boy is eating");
  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
  }
}
```

Output:

```
Boy is eating
```

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.
This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Abstract method

Method which has no body has to be declare as abstract method.

Abstract class

- When there is one or more abstract method in a class then that class should be declared as abstract class.
- To declare a class as abstract, use abstract keyword in front of class keyword.
- **We cannot create an object of abstract class, it is only used for inheritance purpose.**

- We cannot declare abstract constructor or abstract static method.

Syntax:

```
abstract class classname
 {
    //variable declarations;
    //method declarations;
 }
```

Example

```
class A
    {
        int x;
        abstract void show();
        void getx(int a)
    {
        x=a;
    }
    }
    class B extends A
    {
        void show()
    {
        System.out.println(x);
    }
    }
    class AbsDemo
    {
    public static void main(String a[])
    {
        B ob=new B();
        //A ob1=new A();
        ob.getx(10);
        ob.show();
    }
    }
```

Final keyword (important)

There are three use of final keyword;

1. To declare constant variable
2. To prevent method overriding
3. To prevent inheritance

Unit 2. Classes and Objects

1. To declare constant variable

- If a variable is declared as final then it will become constant.
- Its value cannot be change throughout the program.
- To declare a final variable, write final keyword in front of datatype.

Example;

```
final int a=10;
```

2. To prevent method overriding.

- To prevent method from being overridden write final keyword in front of the method name.
- Method declared as final cannot be overridden. i.e. one cannot make same method name and type signature in subclass.

Example;

```
class A
{
int a;
A()
{
a=10;
}
final void show()
{
System.out.println("a is:"+a);
}
}
class B extends A
{
B()
{ }
void show()
{ }
void show(int x)
{
a=x;
System.out.println("a is: "+a);
}
}
class FinalMethod
{
    public static void main(String a[])
    {
    B ob=new B();
    ob.show();
    ob.show(20);
    }
}
```

3. To prevent inheritance.

- Sometimes you want to preserve a class from being inherited, use final keyword before the class keyword.
- Declaring class as final implicitly declares all its method as final.

Example;

```
    final class A
    {
    //variable declarations;
    //method;
    }
    Class B extends A //here cannot inherit class A as it is declared final.
    {
    //code;
    }
```

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.

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?

Unit 2. Classes and Objects

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:

```
interface <interface_name>{
// declare constant fields
// declare methods that abstract
// by default.
}
```

Java Interface Example

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

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

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
```

Unit 2. Classes and Objects

```
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

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

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

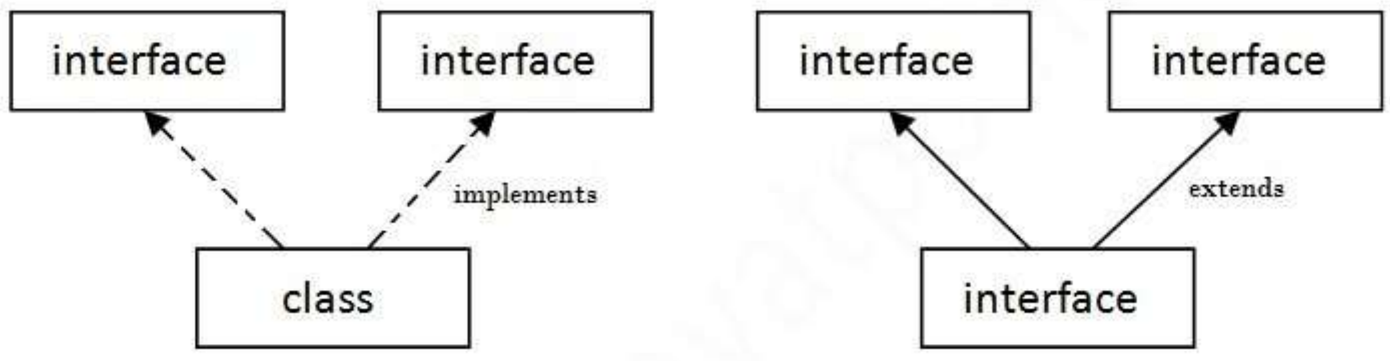
Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

Unit 2. Classes and Objects

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



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

```
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:

Unit 2. Classes and Objects

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
}
}
```

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.

```
    interface Printable{
    void print();
    }
    interface Showable extends Printable{
    void show();
    }
    class TestInterface4 implements Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
    TestInterface4 obj = new TestInterface4();
    obj.print();
```

```
obj.show();
}
 }
```

Output:

```
Hello
Welcome
```

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:

```
interface printable{
void print();
interface MessagePrintable{
void msg();
}
}
```

Difference between Class and Interface

CLASS	INTERFACE
Variables declared in class may be constant.	Variables declared in interface are always constant.
methods declared in a class may have body or may not have body.	methods declared in an interface have no body.
we can make abstract methods using abstract keyword.	Methods declared within an interface will be abstract by default
We can use public, private, protected or default access sprcifier in class	We can use only public or default access specifier in an interface
We can create object of class	We can only declared reference variable of an interface
We have to use extends keyword to inherit the class into another class	We have to use implements keyword to implement interface into class
We cannot inherit more than one class in a class	We can implement more than one interface into a class.

Unit 2. Classes and Objects

(Ques: Explain This and Super keyword)

This keyword

- Sometimes a method needs to refer to the object that invokes it, to allow. Java defines 'This'
  keyword.
- 'This' can be used inside any method to refer to the current object.
- 'This' keyword is always referenced to the object on which method was invoked.
- We can use 'This' keyword anywhere. It is one type of self referential structure.
- A reference to an object of current class is always permitted.

Example:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

super keyword:

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

Unit 2. Classes and Objects

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){System.out.println("eating...");}
```

Unit 2. Classes and Objects

```
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:

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

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
```

```
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Java Arrays

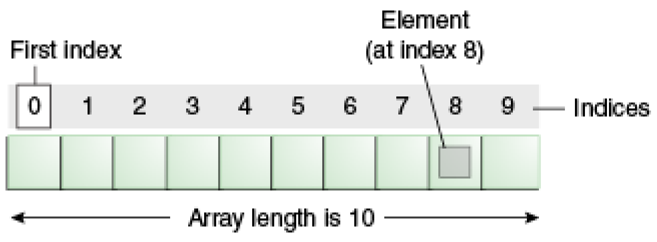
Normally, an array is a collection of similar type of elements which has contiguous memory location.

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.

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.

Unit 2. Classes and Objects

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

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
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.

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

Unit 2. Classes and Objects

Declaration, Instantiation and Initialization of Java Array

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

```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
33
3
4
5
```

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.

```
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];
System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

Output:

Unit 2. Classes and Objects

3

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

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
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.

```
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]+" ");
}
```

Unit 2. Classes and Objects

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

Output:

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

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
public static void main(String[] args){
//declaring a 2D array with odd columns
int arr[][] = new int[3][];
arr[0] = new int[3];
arr[1] = new int[4];
arr[2] = new int[2];
//initializing a jagged array
int count = 0;
for (int i=0; i<arr.length; i++)
for(int j=0; j<arr[i].length; j++)
    arr[i][j] = count++;

//printing the data of a jagged array
for (int i=0; i<arr.length; i++){
for (int j=0; j<arr[i].length; j++){
    System.out.print(arr[i][j]+" ");
}
System.out.println();//new line
}
}
}
```

Output:

```
0 1 2
3 4 5 6
7 8
```

Addition of 2 Matrices in Java

```
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];
//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}}
```

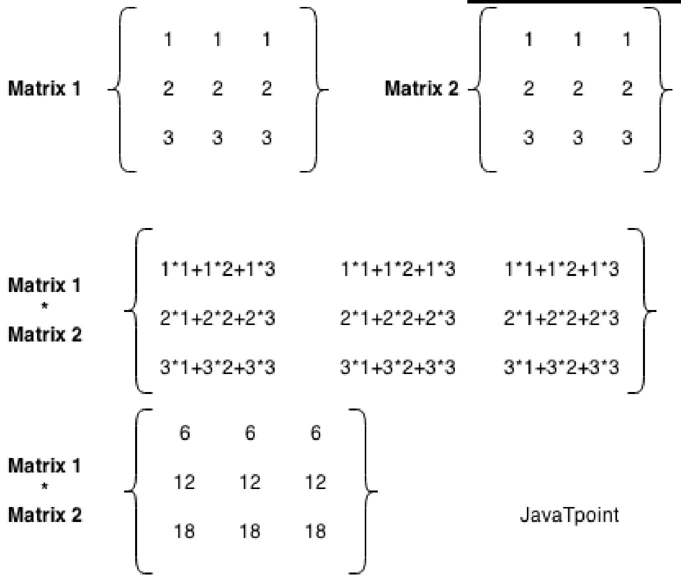
Output:

```
2 6 8
6 8 10
```

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.

Unit 2. Classes and Objects



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3];  //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}}
```

Output:

Unit 2. Classes and Objects

```
6 6 6
12 12 12
18 18 18
```

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

---

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

Unit 2. Classes and Objects

```
protected void finalize(){}
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{
public void finalize(){System.out.println("object is garbage collected");}
public static void main(String args[]){
TestGarbage1 s1=new TestGarbage1();
TestGarbage1 s2=new TestGarbage1();
s1=null;
s2=null;
System.gc();
}
}
```

```
object is garbage collected
object is garbage collected
```

Java String

In java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

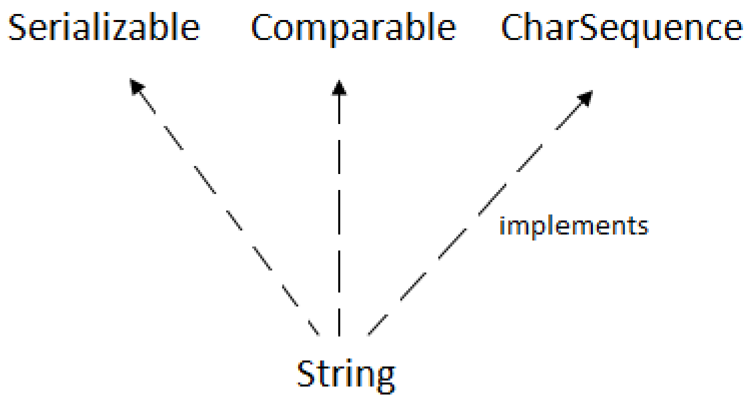
```
char[] ch={'k','i','r','t','a','n'};
String s=new String(ch);
```

is same as:

```
String s="kirtan";
```

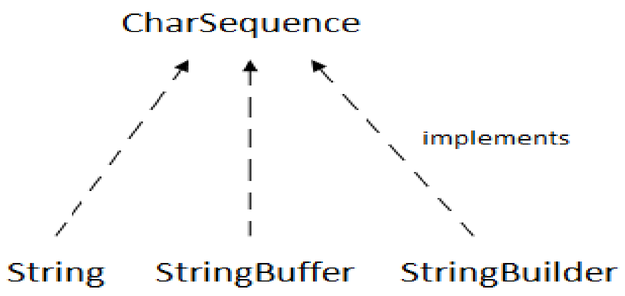
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interface.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder, classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, jvm will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

```
public class StringExample
{
public static void main(String args[])
{
String s1="java";//creating string by Java string literal

char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string

String s3=new String("example");//creating Java string by new keyword

System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

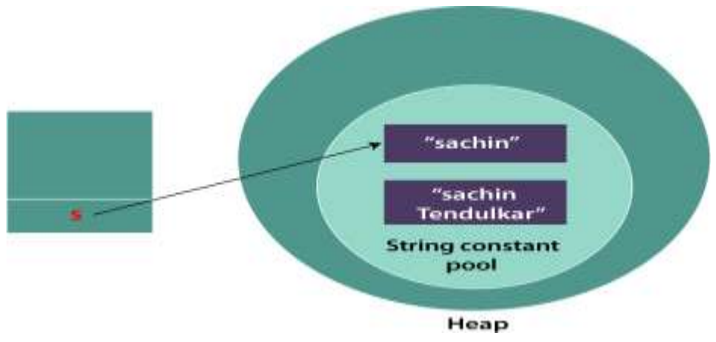
Let's try to understand the concept of immutability by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }  }
```

Output:

```
Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but ***s*** reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

```
String s="Sachin";
s=s.concat(" Tendulkar");
System.out.println(s);
```

```
Sachin Tendulkar
```

In such a case, s points to the "Sachin Tendulkar". Please notice that still Sachin object is not modified.

Why String objects are immutable in Java?

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Why String class is Final in Java?

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

1. charAt()

The **Java String class charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is the length of the string. It returns **StringIndexOutOfBoundsException,** if the given index number is greater than or equal to this string length or a negative number.

Syntax

```
public char charAt(int index)
```

The method accepts **index** as a parameter. The starting index is 0. It returns a character at a specific index position in a string. It throws **StringIndexOutOfBoundsException** if the index is a negative value or greater than this string length.

Specified by CharSequence interface, located inside java.lang package.

```
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
```

```
S
h
```

2. concat()

The **Java String class concat()** method *combines specified string at the end of this string*. It returns a combined string. It is like appending another string.

Signature

The signature of the string concat() method is given below:

```
public String concat(String anotherString)
```

Parameter

anotherString : another string i.e., to be combined at the end of this string.

Returns

combined string

```
String s1="Sachin ";
String s2="Tendulkar";
String s3=s1.concat(s2);
System.out.println(s3);//Sachin Tendulkar
```

```
Sachin Tendulkar
```

The above Java program, concatenates two String objects ***s1*** and ***s2*** using ***concat()*** method and stores the result into ***s3*** object.

3. equals()

The **Java String class equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of the Object class.

Signature public boolean equals(Object anotherObject)

Parameter anotherObject **: another object, i.e., compared with this string.**

Returns

true if characters of both strings are equal otherwise **false**.

```
String s1="kirtan";
String s2="kirtan";
String s3="KIRTAN";
String s4="MIHIKA";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
```

```
true
false
false
```

4. indexOf()

The **Java String class indexOf()** method returns the position of the first occurrence of the specified character or string in a specified string.

Signature

There are four overloaded indexOf() method in Java. The signature of indexOf() methods are given below:

No.	Method	Description
1	int indexOf(int ch)	It returns the index position for the given char value
2	int indexOf(int ch, int fromIndex)	It returns the index position for the given char value and from index
3	int indexOf(String substring)	It returns the index position for the given substring
4	int indexOf(String substring, int fromIndex)	It returns the index position for the given substring and from index

Parameters

ch: It is a character value, e.g. 'a'

fromIndex: The index position from where the index of the char value or substring is returned.

substring: A substring to be searched in this string.

Returns

Index of the searched string or character.

```
String s1="this is index of example";

//passing substring
int index1=s1.indexOf("is");//returns the index of is substring
int index2=s1.indexOf("index");//returns the index of index substring
System.out.println(index1+"  "+index2);//2 8

//passing substring with from index
int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
System.out.println(index3);//5 i.e. the index of another is

//passing char value
int index4=s1.indexOf('s');//returns the index of s char value
System.out.println(index4);//3
```

```
2 8
5
3
```

We observe that when a searched string or character is found, the method returns a non-negative value. If the string or character is not found, -1 is returned. We can use this property to find the total count of a character present in the given string.

5. lastIndexOf()

The **Java String class lastIndexOf()** method returns the last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Signature

There are four types of lastIndexOf() method in Java. The signature of the methods are given below:

No.	Method	Description

1	int lastIndexOf(int ch)	It returns last index position for the given char value
2	int lastIndexOf(int ch, int fromIndex)	It returns last index position for the given char value and from index
3	int lastIndexOf(String substring)	It returns last index position for the given substring
4	int lastIndexOf(String substring, int fromIndex)	It returns last index position for the given substring and from index

Parameters

ch: char value i.e. a single character e.g. 'a'

fromIndex: index position from where index of the char value or substring is retured

substring: substring to be searched in this string

Returns

last index of the string

Java String lastIndexOf() method example

```
String s1="this is index of example";//there are 2 's' characters in this sentence
int index1=s1.lastIndexOf('s');//returns last index of 's' char value
System.out.println(index1);//6
```

```
6
```

Java String lastIndexOf(int ch, int fromIndex) Method Example

Here, we are finding the last index from the string by specifying *fromIndex*.

```
String str = "This is index of example";
int index = str.lastIndexOf('s',5);
System.out.println(index);
```

```
3
```

Java String lastIndexOf(String substring) Method Example

It returns the last index of the substring.

```
String str = "This is last index of example";
int index = str.lastIndexOf("of");
System.out.println(index);
```

```
19
```

Java String lastIndexOf(String substring, int fromIndex) Method Example

It returns the last index of the substring from the *fromIndex*.

```
String str = "This is last index of example";
int index = str.lastIndexOf("of", 25);
System.out.println(index);
index = str.lastIndexOf("of", 10);
System.out.println(index); // -1, if not found
```

```
19
-1
```

6. isEmpty()

The **Java String class isEmpty()** method checks if the input string is empty or not. Note that here empty means the number of characters contained in a string is zero.

Signature The signature or syntax of string isEmpty() method is given below:

```
public boolean isEmpty()
```

Returns

true if length is 0 otherwise false.

```
String  s1="";
String s2="kirtan";

System.out.println(s1.isEmpty());
System.out.println(s2.isEmpty());
```

```
true
false
```

7. join()

The Java String class join() method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element. The join() method is included in the Java string since JDK 1.8.

Signature

The signature or syntax of the join() method is given below:

```
public static String join(CharSequence delimiter, CharSequence... elements)
```

Parameters

delimiter : char value to be added with each element

elements : char value to be attached with delimiter

Returns

joined string with delimiter

Exception Throws

NullPointerException if element or delimiter is null.

```
String joinString1=String.join("-","welcome","to","javaworld");
System.out.println(joinString1);
```

```
welcome-to-javaworld
```

8. length()

The **Java String class length()** method finds the length of a string. The length of the Java string is the same as the Unicode code units of the string.

Signature

The signature of the string length() method is given below:

```
public int length()
```

Specified by

CharSequence interface

Returns

Length of characters. In other words, the total number of characters present in the string.

```
String s1="kirtanpatel";
String s2="python";
System.out.println("string length is: "+s1.length());//11 is the length of kirtanpatel
stringSystem.out.println("string length is: "+s2.length());//6 is the length of python
string
```

```
string length is: 11
string length is: 6
```

9. split()

The **java string split()** method splits this string against given regular expression and returns a char array.

Signature

There are two signature for split() method in java string.

```
public String split(String regex)
and,
public String split(String regex, int limit)
```

Parameter

regex : regular expression to be applied on string.

limit : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

Returns
array of strings

```
String s1="java string split method by kirtan";
String[] words=s1.split("\\s");//splits the string based on whitespace
//using java foreach loop to print elements of string array
for(String w:words){
    System.out.println(w);
    }
```

```
java
string
split
method
by
kirtan
```

Split With Regular Expression

```
String Str = new String("Welcome-to-java");
    System.out.println("Return Value :" );

    for (String retval: Str.split("-")) {
      System.out.println(retval);
    }
  }
}
```

Output

```
Return Value :
Welcome
To
java
```

10.substring()

The **Java String class substring()** method returns a part of the string.

We pass beginIndex and endIndex number position in the Java substring method where beginIndex is inclusive, and endIndex is exclusive. In other words, the beginIndex starts from 0, whereas the endIndex starts from 1.

There are two types of substring methods in Java string.

Signature

```
public String substring(int startIndex) // type - 1
and
public String substring(int startIndex, int endIndex)  // type - 2
```

If we don't specify endIndex, the method will return all the characters from startIndex.

Parameters

startIndex : starting index is inclusive

endIndex : ending index is exclusive

Returns

specified string

Exception Throws

StringIndexOutOfBoundsException is thrown when any one of the following conditions is met.

- if the start index is negative value
- end index is lower than starting index.
- Either starting or ending index is greater than the total number of characters present in the string.

```
String s1="kirtan";
String substr = s1.substring(0); // Starts with 0 and goes to end
System.out.println(substr);
String substr2 = s1.substring(2,5); // Starts from 2 and goes to 5
System.out.println(substr2);
String substr3 = s1.substring(5,15); // Returns Exception
```

```
kirtan
rtan
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: begin 5, end 15,
length 6
```

11.trim()

The **Java String class trim()** method eliminates leading and trailing spaces. The Unicode value of space character is '\u0020'. The trim() method in Java string checks this Unicode value before and after the string, if it exists then the method removes the spaces and returns the omitted string.

The string trim() method doesn't omit middle spaces.

Signature

The signature or syntax of the String class trim() method is given below:

```
public String trim()
```

Returns

string with omitted leading and trailing spaces

```
String s1="  hello string   ";
System.out.println(s1+"kirtan");//without trim()
System.out.println(s1.trim()+"kirtan");//with trim()
```

```
hello string   kirtan
hello stringkirtan
```

12.toUpperCase() and toLowerCase()

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

```
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
```

```
SACHIN
sachin
Sachin
```

13. startsWith() and endsWith()

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

```
String s="Sachin";
System.out.println(s.startsWith("Sa"));//true
System.out.println(s.endsWith("n"));//true
```

```
true
true
```

14. replace()

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
String s1="Java is a programming language. Java is a platform. Java is an Island.";
String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava
"
System.out.println(replaceString);
```

```
Kava is a programming language. Kava is a platform. Kava is an Island.
```

String comparision

We can compare String in Java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

1) By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

```
String s1="Sachin";
String s2="Sachin";
String s3=new  String("Sachin");
String s4="Saurav";
System.out.println(s1.equals(s2));//true
System.out.println(s1.equals(s3));//true
System.out.println(s1.equals(s4));//false
```

```
true
true
false
```

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

```
String s1="Sachin";
String s2="SACHIN";
System.out.println(s1.equals(s2));//false
System.out.println(s1.equalsIgnoreCase(s2));//true
```

```
false
true
```

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings.

2) By Using == operator

The == operator compares references not values.

```
String s1="Sachin";
String s2="Sachin";
String s3=new String("Sachin");
System.out.println(s1==s2);//true (because both refer to same instance)
System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
```

```
true
false
```

3) By Using compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

- **s1 == s2** : The method returns 0.
- **s1 > s2** : The method returns a positive value.
- **s1 < s2** : The method returns a negative value.

```
String s1="Sachin";
String s2="Sachin";
String s3="Ratan";
```

```
        System.out.println(s1.compareTo(s2));//0
        System.out.println(s1.compareTo(s3));//1(because s1>s3)
        System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
```

```
0
1
-1
```

String Concatenation in Java

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator
2. By concat() method

1) String Concatenation by + (String concatenation) operator

```
String s="Sachin"+"Tendulkar";
System.out.println(s);//Sachin Tendulkar
```

Output:

The **Java compiler transforms** above code to this:

String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();

In Java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and it's append method. String concatenation operator produces a new String by appending the second operand onto the end of the first operand. The String concatenation operator can concatenate not only String but primitive values also. For Example:

```
 String s=50+30+"Sachin"+40+40;
System.out.println(s);//80Sachin4040
```

1.

```
80Sachin4040
```

Note: After a string literal, all the + will be treated as string concatenation operator.

2) String Concatenation by concat() method

The String concatenation concat() method concatenates the specified string to the end of current string. Syntax:

```
public String concat(String another)
```

```
String s1="Sachin ";
String s2="Tendulkar";
String s3=s1.concat(s2);
System.out.println(s3);//Sachin Tendulkar
```

```
Sachin Tendulkar
```

The above Java program, concatenates two String objects ***s1*** and ***s2*** using ***concat()*** method and stores the result into ***s3*** object.

There are some other possible ways to concatenate Strings in Java,

1. String concatenation using StringBuilder class

```
StringBuilder s1 = new StringBuilder("Hello");    //String 1
StringBuilder s2 = new StringBuilder(" World");    //String 2
StringBuilder s = s1.append(s2);   //String 3 to store the result
   System.out.println(s.toString());  //Displays result
```

```
Hello World
```

In the above code snippet, **s1, s2** and **s** are declared as objects of **StringBuilder** class. **s** stores the result of concatenation of **s1** and **s2** using **append**() method.

2. String concatenation using format() method

String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.

```
String s1 = new String("Hello");    //String 1
String s2 = new String(" World");    //String 2
String s = String.format("%s%s",s1,s2);   //String 3 to store the result
   System.out.println(s.toString()); //Displays result
```

```
Hello World
```

Here, the String objects **s** is assigned the concatenated result of Strings **s1** and **s2** using **String.format()** method. format() accepts parameters as format specifier followed by String objects or values.

3. String concatenation using String.join() method (Java Version 8+)

The String.join() method is available in Java version 8 and all the above versions. String.join()

method accepts arguments first a separator and an array of String objects.

```
String s1 = new String("Hello");    //String 1
String s2 = new String(" World");    //String 2
String s = String.join("",s1,s2);   //String 3 to store the result
   System.out.println(s.toString()); //Displays result
```

```
Hello World
```

In the above code snippet, the String object **s** stores the result of **String.join("",s1,s2)** method. A separator is specified inside quotation marks followed by the String objects or array of String objects.

toString() Method

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the String representation of the object.

If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

How to use the toString() method

The toString() method in Java has two implementations;

- The first implementation is when it is called as a method of an object instance. The example below shows this implementation

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

```
      //Creating an integer of value 10
      Integer number=10;
      // Calling the toString() method as a function of the Integer variable
      System.out.println( number.toString() );
   }
```

Output

```
10
```

- The second implementation is when you call the member method of the relevant class by passing the value as an *argument*. The example below shows how to do this

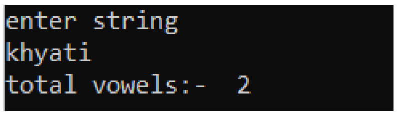
```
class HelloWorld {
   public static void main( String args[] ) {

      // The method is called on datatype Double
      // It is passed the double value as an argument
      System.out.println(Double.toString(11.0));
      // Implementing this on other datatypes

      //Integer
      System.out.println(Integer.toString(12));

      // Long
      System.out.println(Long.toString(123213123));

      // Booleam
      System.out.println(Boolean.toString(false));
   }
}
```

Output

```
11.0

12

123213123

false
```

Example 1: (write a program to count number of vowels in inputted string)

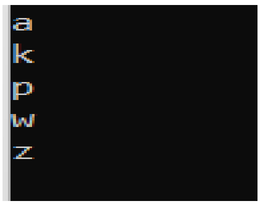
```
import java.io.*;
class vowel
{
        public static void main(String args[])throws IOException
        {
        DataInputStream dis=new DataInputStream(System.in);
        int cnt=0;
        System.out.println("enter string");
        String st=dis.readLine();
        for(int i=0;i<st.length();i++)
        {
                char t=st.charAt(i);
                if(t=='a' || t=='e' || t=='o' || t=='i' || t=='u' ||
                   t=='A' || t=='E' || t=='O' || t=='I' || t=='U')
                        cnt++;
        }
        System.out.println("total vowels:-  "+cnt);
        }
}
```



```
enter string
khyati
total vowels:-  2
```

Example 2: (write a program to enter 5 string and sort it)

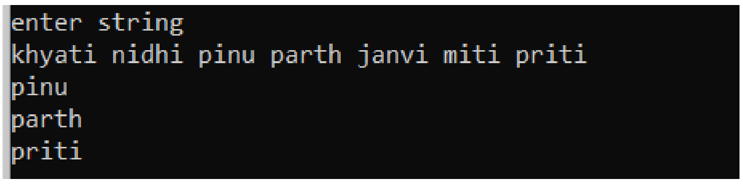
```
class sort1
{
        public static void main(String args[])
        {
        String st[]={"w","p","a","k","z"};
                for(int i=0;i<st.length;i++)
                {
                for(int j=i+1;j<st.length;j++)
                {
                        if(st[i].compareTo(st[j])>0)
                        {
                                String t=st[i];
                                st[i]=st[j];
                                st[j]=t;
                        }
                }
                }
                for(String t:st)
                {
                        System.out.println(t);
                }
        }
}
```



```
a
k
p
w
z
```

Example 3: (write a program to find words beginning with ‘p’ chatacter in inputted string)

```
import java.io.*;
class search1
{
	public static void main(String args[])throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter string");
	String st=dis.readLine();
	String t[]=st.split(" ");
	for(int i=0;i<t.length;i++)
	{
		if(t[i].startsWith("p"))
			System.out.println(t[i]);
	}
	}
}
```



```
enter string
khyati nidhi pinu parth janvi miti priti
pinu
parth
priti
```

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.

public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

1) StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

StringBufferExample.java

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Output:

```
Hello Java
```

2) StringBuffer insert() Method

The insert() method inserts the given String with this string at the given position.

StringBufferExample2.java

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

Output:

```
HJavaello
```

3) StringBuffer replace() Method

The replace() method replaces the given String from the specified beginIndex and endIndex.

StringBufferExample3.java

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

Output:

```
HJavalo
```

4) StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

StringBufferExample4.java

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

Output:

```
Hlo
```

5) StringBuffer reverse() Method

The reverse() method of the StringBuilder class reverses the current String.

StringBufferExample5.java

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

Output:

```
olleH
```

6) StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBufferExample6.java

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

Output:

```
16
16
34
```

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBufferExample7.java

```
class StringBufferExample7{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10);//now no change
System.out.println(sb.capacity());//now 34
sb.ensureCapacity(50);//now (34*2)+2
System.out.println(sb.capacity());//now 70
}
}
```

Output:

```
16
16
34
34
70
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.

3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

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.

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

Advantage of Exception Handling

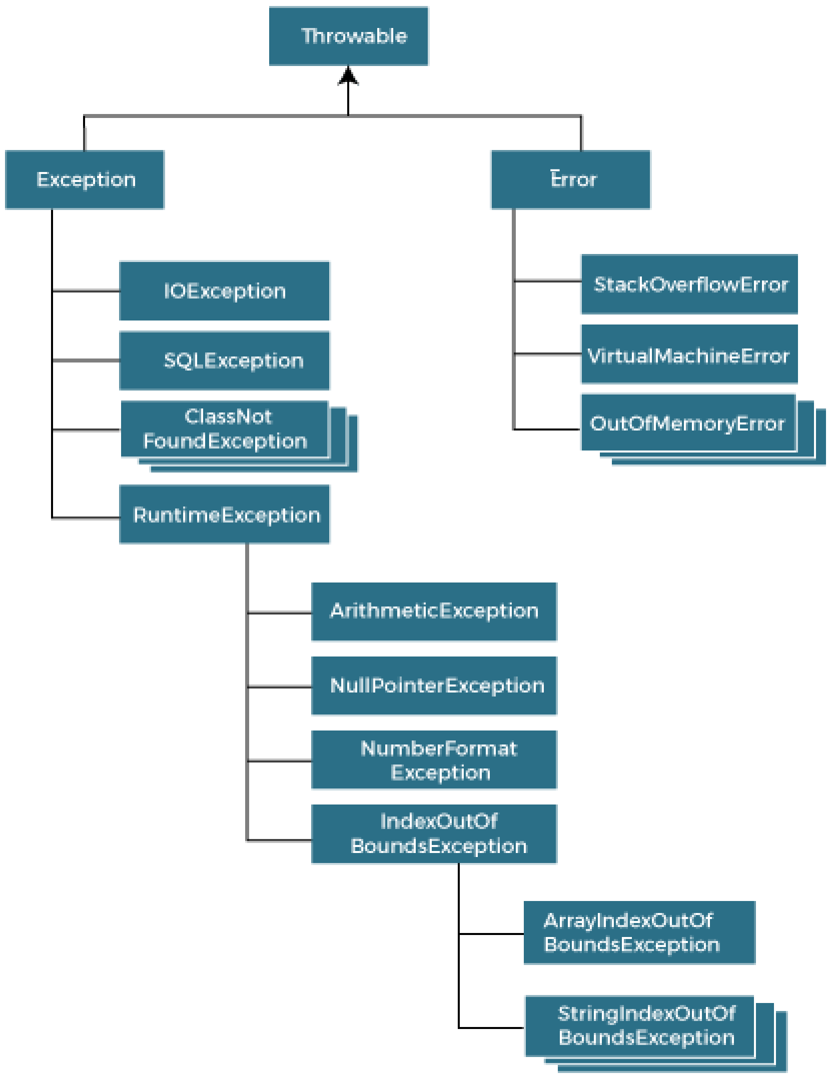
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

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

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

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

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

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.

```
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.

```
String s=null;
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.

```
String s="abc";
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.

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

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Example 1

TryCatchExample1.java

```
public class TryCatchExample1 {

public static void main(String[] args) {

int data=50/0; //may throw exception

System.out.println("rest of the code");

}
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```
public class TryCatchExample2 {

public static void main(String[] args) {
try
{
int data=50/0; //may throw exception
}
    //handling the exception
catch(ArithmeticException e)
{
    System.out.println(e);
}
System.out.println("rest of the code");
}
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

User-Defined Exception (Java Custom Exception)

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as **custom exception or user-defined exception**. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

In this section, we will learn how custom exceptions are implemented and used in Java programs.

Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

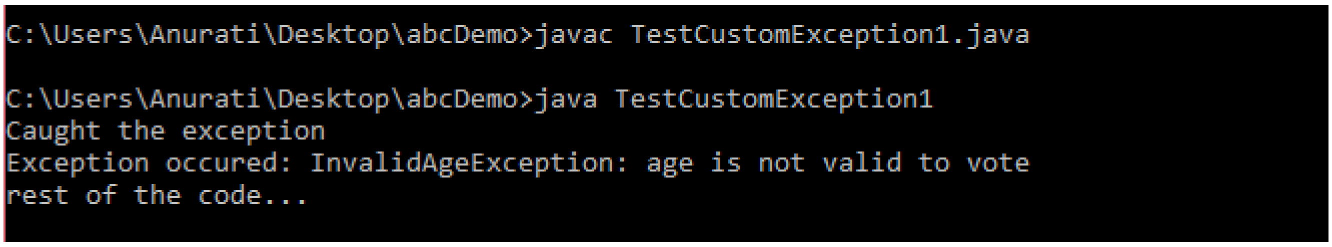
Consider the following example, where we create a custom exception named WrongFileNameException:

```
public class WrongFileNameException extends Exception {
public WrongFileNameException(String errorMessage) {
super(errorMessage);
}
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

Example 1:

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.



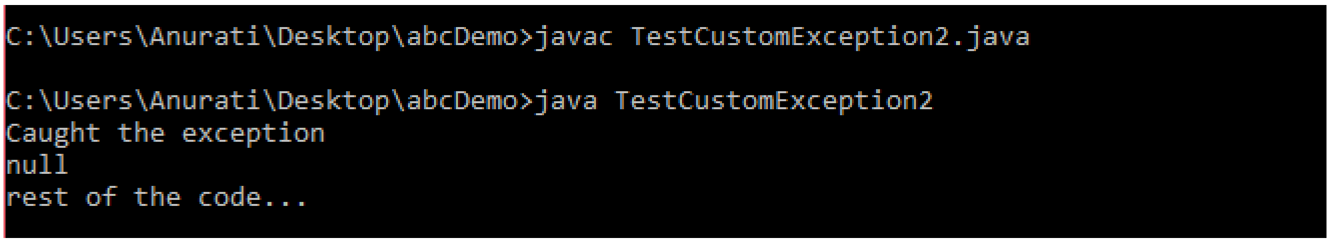
```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Example 2:

TestCustomException.java

```
// class representing custom exception
class MyCustomException extends Exception
{
}
// class that uses custom exception MyCustomException
public class TestCustomException2
{
// main method
public static void main(String args[])
{
try
{
    // throw an object of user defined exception
    throw new MyCustomException();
}
catch (MyCustomException ex)
{
    System.out.println("Caught the exception");
    System.out.println(ex.getMessage());
}

System.out.println("rest of the code...");
}
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
```

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

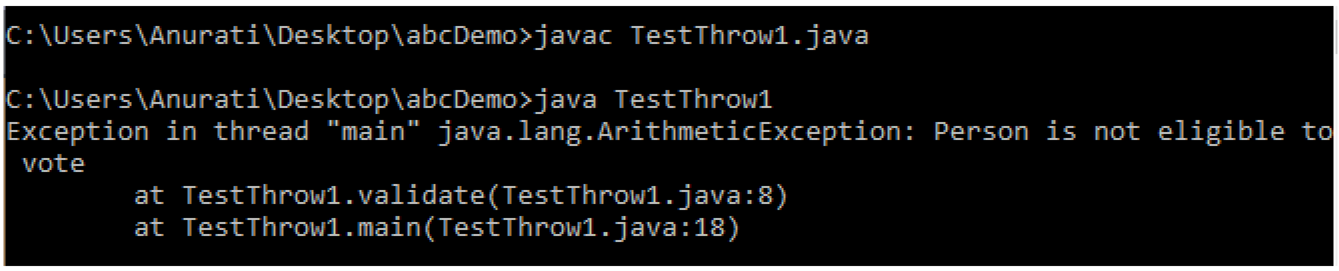
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {
//function to check if person is eligible to vote or not
public static void validate(int age) {
if(age<18) {
    //throw Arithmetic exception if not eligible to vote
    throw new ArithmeticException("Person is not eligible to vote");
}
else {
    System.out.println("Person is eligible to vote!!");
}
}
//main method
public static void main(String args[]){
//calling the function
validate(13);
System.out.println("rest of the code...");
}
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

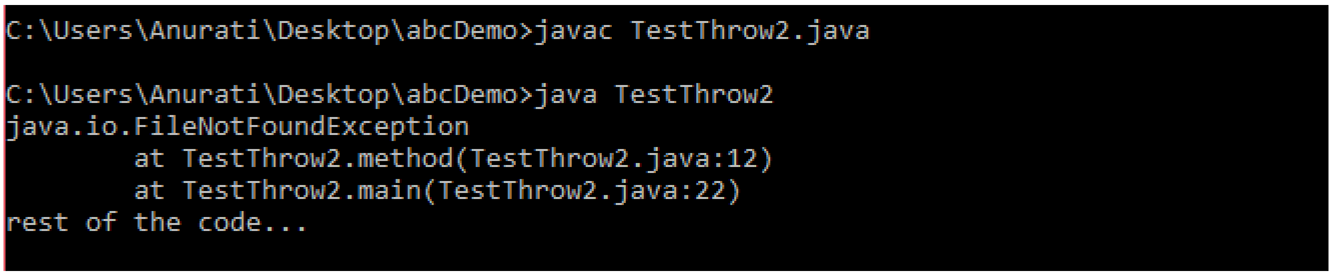
Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
import java.io.*;

public class TestThrow2 {

//function to check if person is eligible to vote or not
public static void method() throws FileNotFoundException {

FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
BufferedReader fileInput = new BufferedReader(file);



throw new FileNotFoundException();

}
//main method
public static void main(String args[]){
try
{
    method();
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
System.out.println("rest of the code...");
}
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

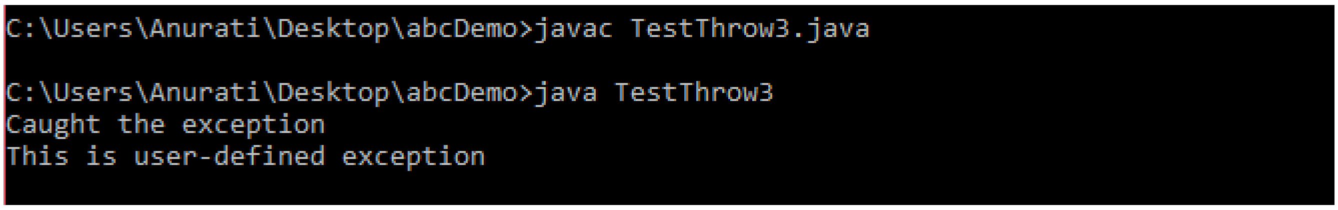
Example 3: Throwing User-defined Exception
exception is everything else under the Throwable class.

TestThrow3.java

```
// class represents user-defined exception
class UserDefinedException extends Exception
{
public UserDefinedException(String str)
{
// Calling constructor of parent Exception
super(str);
}
}
// Class that uses above MyException
public class TestThrow3
{
public static void main(String args[])
{
try
{
    // throw an object of user defined exception
    throw new UserDefinedException("This is user-defined exception");
}
catch (UserDefinedException ude)
{
    System.out.println("Caught the exception");
    // Print the message from MyException object
    System.out.println(ude.getMessage());
}
}
```

}

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow3
Caught the exception
This is user-defined exception
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws Example

Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

Testthrows1.java

```
import java.io.IOException;
class Testthrows1{
void m()throws IOException{
throw new IOException("device error");//checked exception
}
void n()throws IOException{
m();
}
void p(){
try{
n();
}catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[]){
Testthrows1 obj=new Testthrows1();
obj.p();
System.out.println("normal flow...");
}
}
```

Output:

```
exception handled
normal flow...
```

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

There are two cases:

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

Testthrows2.java

```
import java.io.*;
class M{
```

```
void method()throws IOException{
throw new IOException("device error");
}
}
public class Testthrows2{
public static void main(String args[]){
try{
M m=new M();
m.method();
}catch(Exception e){System.out.println("exception handled");}

System.out.println("normal flow...");
}
}
```

Output:

```
exception handled
     normal flow...
```

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

Let's see examples for both the scenario.

A) If exception does not occur

Testthrows3.java

```
import java.io.*;
class M{
void method()throws IOException{
System.out.println("device operation performed");
}
}
class Testthrows3{
public static void main(String args[])throws IOException{//declare exception
M m=new M();
m.method();

System.out.println("normal flow...");
}
}
```

Output:

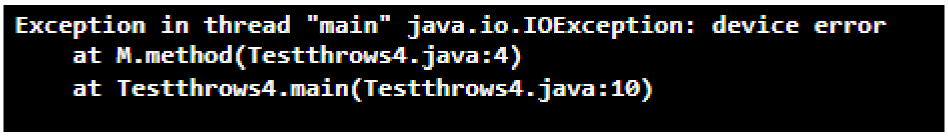
```
device operation performed
      normal flow...
```

B) If exception occurs

Testthrows4.java

```
import java.io.*;
class M{
void method()throws IOException{
throw new IOException("device error");
}
}
class Testthrows4{
public static void main(String args[])throws IOException{//declare exception
M m=new M();
m.method();

System.out.println("normal flow...");
}
}
```

Output:



```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

Difference between throw and throws in Java

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.

2.	Type of exception	Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Difference between final, finally and finalize

OR

Explain final keyword,finally block and finalized method

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

no	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.

3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

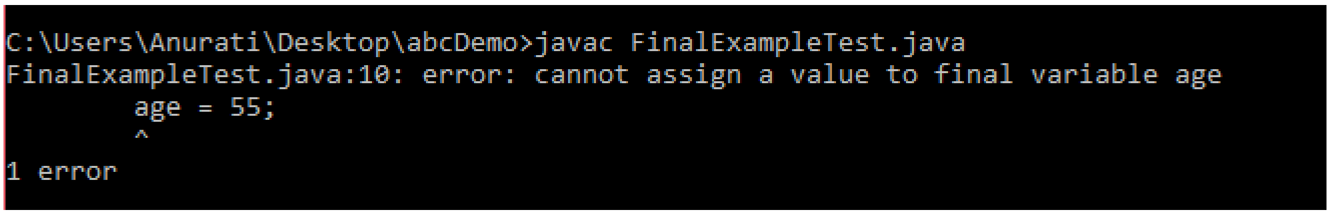
Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

FinalExampleTest.java

```
public class FinalExampleTest {

    //declaring final variable
    final int age = 18;
    void display() {

    // reassigning value to age variable
    // gives compile time error
    age = 55;
    }

    public static void main(String[] args) {

    FinalExampleTest obj = new FinalExampleTest();
    // gives compile time error
    obj.display();
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
        age = 55;
        ^
1 error
```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

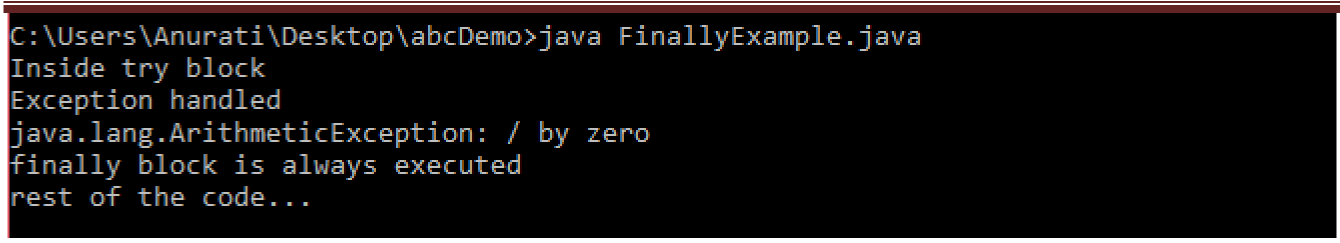
Java finally Example

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

FinallyExample.java

```java
public class FinallyExample {
public static void main(String args[]){
try {
System.out.println("Inside try block");
// below code throws divide by zero exception
int data=25/0;
System.out.println(data);
}
// handles the Arithmetic Exception / Divide by zero exception
catch (ArithmeticException e){
System.out.println("Exception handled");
System.out.println(e);
}
// executes regardless of exception occurred or not
finally {
System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
}
```

Output:



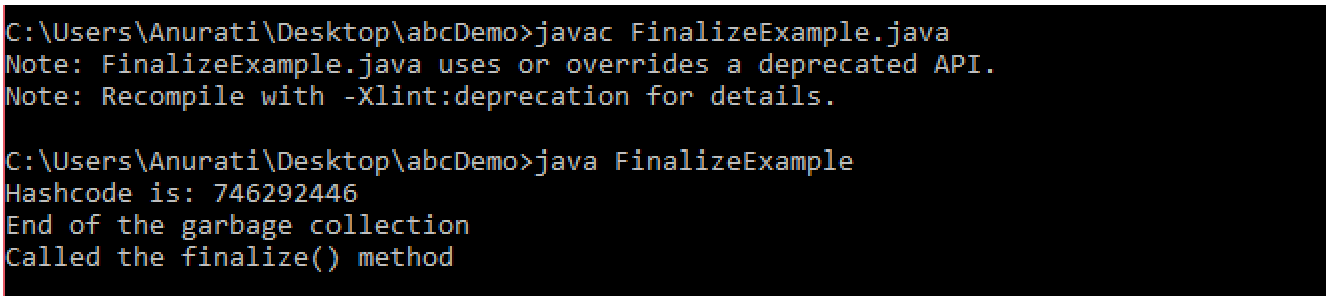
```
C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Java finalize Example

FinalizeExample.java

```
public class FinalizeExample {
public static void main(String[] args)
{
FinalizeExample obj = new FinalizeExample();
// printing the hashcode
System.out.println("Hashcode is: " + obj.hashCode());
obj = null;
// calling the garbage collector using gc()
System.gc();
System.out.println("End of the garbage collection");
}
// defining the finalize method
protected void finalize()
{
System.out.println("Called the finalize() method");
}
}
```

Output:



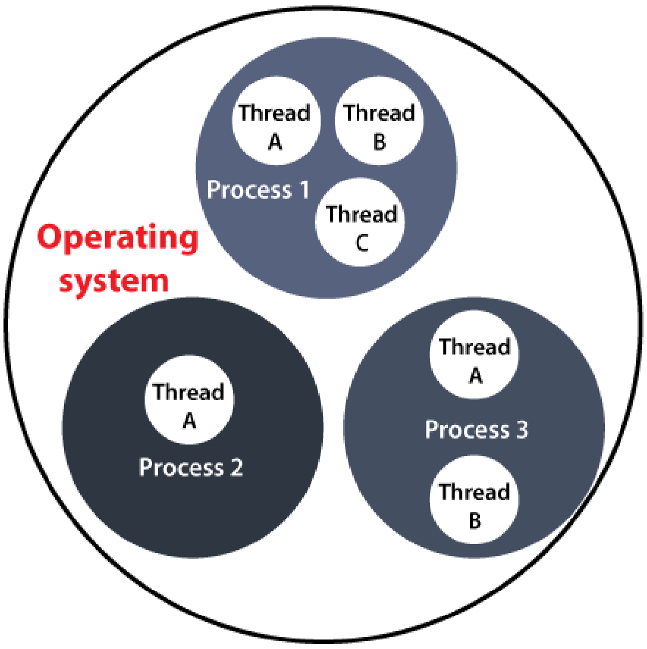
```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
Hashcode is: 746292446
End of the garbage collection
Called the finalize() method
```

Thread Concept in Java

Before introducing the **thread concept**, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, **Thread Concept** was introduced.

A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the **Thread concept in Java**. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as **Multithreading**.

In a simple way, a Thread is a:

- Feature through which we can perform multiple activities within a single process.
- Lightweight process.
- Series of executed statements.
- Nested sequence of method calls.

Note: At a time one thread is executed only.

Difference between Thread and Process

Process	Thread
A process is an instance of a program thatis being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process bythe operating system.	The operating system takes all the user-level threadsas a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fastbecause they are very lightweight.
The data segment and code segment ofeach process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time takingas each new process takes all the resources.	A thread needs less time for creation.

Life cycle of a Thread (Thread States)

OR

Thread Model

Life Cycle of Thread in Java is basically state transitions of a thread that starts from itsbirth and ends on its death.

When an instance of a thread is created and is executed by calling start() method of Threadclass, the thread goes into runnable state.

When sleep() or wait() method is called by Thread class, the thread enters into non-runnablestate.

From non-runnable state, thread comes back to runnable state and continues execution of statements. When the thread comes out of run() method, it dies. These state transitions of athread are called **Thread life cycle in Java**.

Thread States in Java

A thread is a path of execution in a program that enters in any one of the following five statesduring its life cycle. The five states are as follows:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead

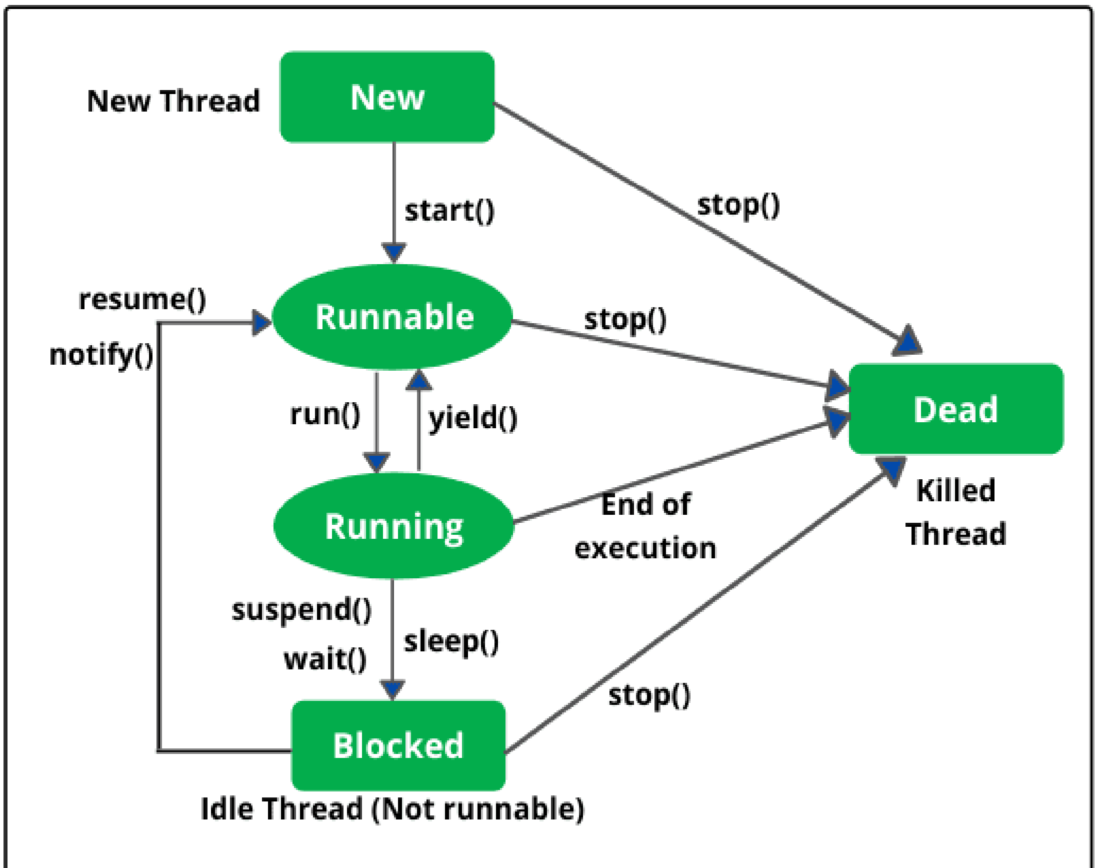


Fig: State Transition Diagram of a Thread

1. New (Newborn State): When we create a thread object using Thread class, thread is bornand is known to be in Newborn state. That is, when a thread is born, it enters into new state the start() method has not been called yet on the instance.

In other words, Thread object exists but it cannot execute any statement because it is not anexecution of thread. Only start() method can be called on a new thread; otherwise, an **IllegalThreadStateException** will be thrown.

2. Runnable state: Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.

In runnable state, thread is ready for execution and is waiting for availability of the processor(CPU time). That is, thread has joined queue (line) of threads that are waiting for execution. If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.

3. Running state: Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.

In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come intorunning state only from runnable state.

A running thread may give up its control in any one of the following situations and can enterinto the blocked state.

- When sleep() method is invoked on a thread to sleep for specified time period, the threadis out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

- When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

- When wait() method is called on a thread to wait for some time. The thread in wait statecan be run again using notify() or notifyAll() method.

4. Blocked state: A thread is considered to be in the blocked state when it is suspended,sleeping, or waiting for some time in order to satisfy some condition.

5. Dead state: A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a threadcomes out of run() method. A thread can also be dead when the stop() method is called.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads areexecuting, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting forsatisfying some conditions. Therefore, a thread is always in any of the five states.

How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

1. Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

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

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on thethread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep(temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specifiedmiliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pauseand allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

2. Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended tobe executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs thefollowing tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);           // Using the constructor Thread(Runnable r)
t1.start();
 }
 }
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors definedabove.

```
public class MyThread1
{
public static void main(String argvs[])
{
Thread t= new Thread("My first thread");
t.start();
String str = t.getName();
System.out.println(str);
}
}
```

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}
public static void main(String argvs[])
{
Runnable r1 = new MyThread2();

Thread th1 = new Thread(r1, "My new thread");

th1.start();
String str = th1.getName();
System.out.println(str);
```

```
    }
    }
```

Output:

```
My new thread

Now the thread is running ...
```

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is upto the thread scheduler to pick one of the threads and ignore the other ones.

There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread.sleep()

The Java Thread class provides the two variant of the sleep() method. First one accepts only an one arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

1. **public static void** sleep(**long** mls) **throws** InterruptedException
2. **public static void** sleep(**long** mls, **int** n) **throws** InterruptedException

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is

accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non- native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters:

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the currentthread.

Whenever another thread does interruption while the current thread is already in the sleepmode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almostequal to the time passed in the argument.

Example of the sleep() method in Java : on the custom thread

```java
class TestSleepMethod1 extends Thread{
public void run(){
for(int i=1;i<5;i++){
// the thread will sleep for the 500 milli seconds
try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
System.out.println(i);
}
}
public static void main(String args[]){
TestSleepMethod1 t1=new TestSleepMethod1();
TestSleepMethod1 t2=new TestSleepMethod1();

t1.start();
t2.start();
}
}
```

Output:

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

As you know well that at a time only one thread is executed. If you sleep a thread for thespecified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

```
// important import statements
import java.lang.Thread;
 import java.io.*;

public class TestSleepMethod2
{
    // main method
public static void main(String argvs[])
{

try
 {
for (int j = 0; j < 5; j++)
{
// The main thread sleeps for the 1000 milliseconds, which is 1 sec
// whenever the loop runs
Thread.sleep(1000);


// displaying the value of the variable
System.out.println(j);
}
}
catch (Exception expn)
{
// catching the exception System.out.println(expn);
} }  }
```

Output:

```
0
1
2
3
4
```

Example of the sleep() Method in Java: When the sleeping time is -ive

when the time for sleeping is negative it throws IllegalArguementException

```
Thread.sleep(-100);
```

Output : java.lang.IllegalArgumentException: timeout value is negative

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

```
public class TestThreadTwice1 extends Thread
{
 public void run()
{
   System.out.println("running...");
 }
 public static void main(String args[])
{
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

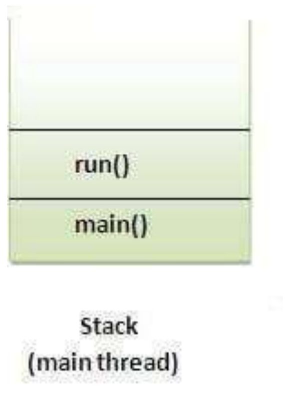
What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the currentcall stack rather than at the beginning of a new call stack.

```
class TestCallRun1 extends Thread
{
 public void run()
{
   System.out.println("running...");
 }
 public static void main(String args[])
{
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

Output:

```
running...
```



Problem if you direct call run() method

```
class TestCallRun2 extends Thread
{
 public void run()
{
  for(int i=1;i<5;i++)
{
    try
```

```
      {
       Thread.sleep(500);
       }
       catch(InterruptedException e)
       {
       System.out.println(e);
       }
       System.out.println(i);
     }
    }
   public static void main(String args[])
  {
    TestCallRun2 t1=new TestCallRun2();
    TestCallRun2 t2=new TestCallRun2();
    t1.run();
    t2.run();
    }
  }
```

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

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

Priority of a Thread (Thread Priority)

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

Setter & Getter Method of Thread Priority

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

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

3 constants defined in Thread class:

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

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

Example of priority of a Thread:

```
public class ThreadPriorityExample extends Thread
{
public void run()
{
System.out.println("Inside the run() method");
}
public static void main(String argvs[])
{
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

Thread.currentThread().setPriority(10);

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

Output:

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

We know that a thread with high priority will get preference over lower priority threads whenit comes to the execution of threads. However, there can be other scenarios where two threadscan have the same priority. All of the processing, in order to look after the threads, is done bythe Java thread scheduler.

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

How to perform single task by multiple threads in Java?

If you have to perform a single task by many threads, have only one run() method. For example:

Program of performing single task by multiple threads

FileName: TestMultitasking1.java

```
class TestMultitasking1 extends Thread{
public void run(){
System.out.println("task one");
}
public static void main(String args[]){
TestMultitasking1 t1=new TestMultitasking1();
TestMultitasking1 t2=new TestMultitasking1();
TestMultitasking1 t3=new TestMultitasking1();
t1.start();
t2.start();
t3.start();
}
}
```

Output:

```
task one
task one
task one
```

Program of performing single task by multiple threads

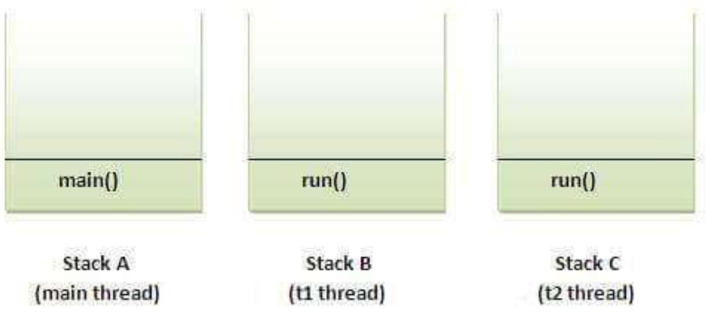
FileName: TestMultitasking2.java

```
class TestMultitasking2 implements Runnable{
public void run(){
System.out.println("task one");
}
public static void main(String args[]){
Thread t1 =new Thread(new TestMultitasking2());//passing annonymous object of TestMultitasking2 class
Thread t2 =new Thread(new TestMultitasking2());
t1.start();
t2.start();
}
}
```

Output:

```
task one
task one
```

Note: Each thread run in a separate callstack.



How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads,have multiple run() methods.For example:

Program of performing two tasks by two threads

FileName: TestMultitasking3.java

```
class Simple1 extends Thread{
public void run(){
System.out.println("task one");
}
}
class Simple2 extends Thread{
public void run(){
System.out.println("task two");
}
}
class TestMultitasking3{
public static void main(String args[]){
Simple1 t1=new Simple1();
Simple2 t2=new Simple2();
t1.start();
t2.start();
}
}
```

Output:

```
task one
task two
```

Same example as above by anonymous class that extends Thread class:

Program of performing two tasks by two threads

FileName: TestMultitasking4.java

```
class TestMultitasking4{
public static void main(String args[]){
```

```
Thread t1=new Thread(){
public void run(){
System.out.println("task one");
}
};
Thread t2=new Thread(){
public void run(){
System.out.println("task two");
}
};
t1.start();
t2.start();
}
}
```

Output:

```
task one
task two
```

Same example as above by anonymous class that implements Runnable interface:

Program of performing two tasks by two threads

FileName: TestMultitasking5.java

```
class TestMultitasking5{
public static void main(String args[]){
Runnable r1=new Runnable(){
public void run(){
System.out.println("task one");
}
};
Runnable r2=new Runnable(){
public void run(){
System.out.println("task two");
}
};
Thread t1=new Thread(r1);
Thread t2=new Thread(r2);
t1.start();
t2.start();
}
}
```

Output:

```
task one
task two
```

Printing even and odd numbers using two threads

To print the even and odd numbers using the two threads, we will use the synchronized block and the notify() method. Observe the following program.

FileName: OddEvenExample.java

```
public class OddEvenExample
{
int contr = 1;
static int NUM;
public void displayOddNumber()
{
synchronized (this)
{
while (contr < NUM)
{
while (contr % 2 == 0)
{
try
{
wait();
}
catch (InterruptedException ex)
{
ex.printStackTrace();
}
}
System.out.print(contr + " ");
contr = contr + 1;
notify();
}
}
}
public void displayEvenNumber()
{
synchronized (this)
{
while (contr < NUM)
{
while (contr % 2 == 1)
{
try
{
wait();
}
catch (InterruptedException ex)
{
ex.printStackTrace();
}
}
System.out.print(contr + " ");
contr = contr +1;
notify();
}
}
}
public static void main(String[] argvs)
{
NUM = 20;
OddEvenExample oe = new OddEvenExample();
Thread th1 = new Thread(new Runnable()
{
```

```
public void run()
{
oe.displayEvenNumber();
}
});
Thread th2 = new Thread(new Runnable()
{
public void run()
{
oe.displayOddNumber();
}
});
th1.start();
th2.start();
}
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Java Package

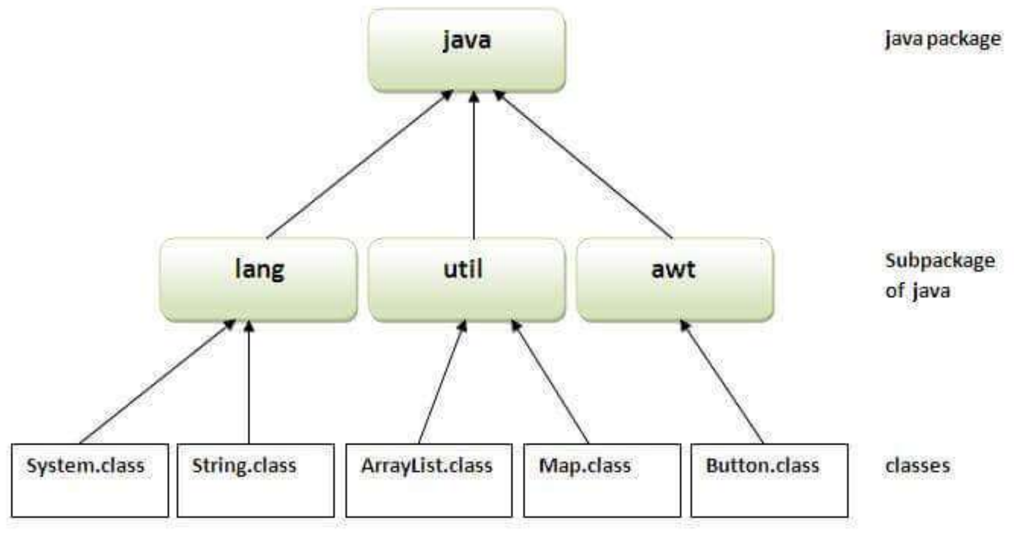
- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
public static void main(String args[]){
System.out.println("Welcome to package");
}
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folde

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//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
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java
  package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
  class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

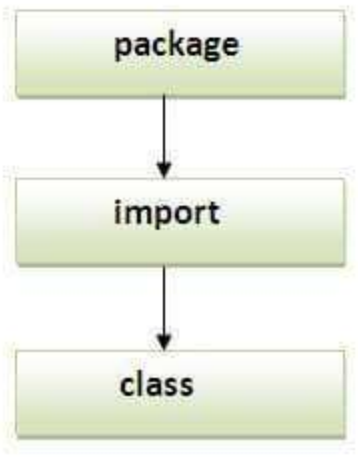
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

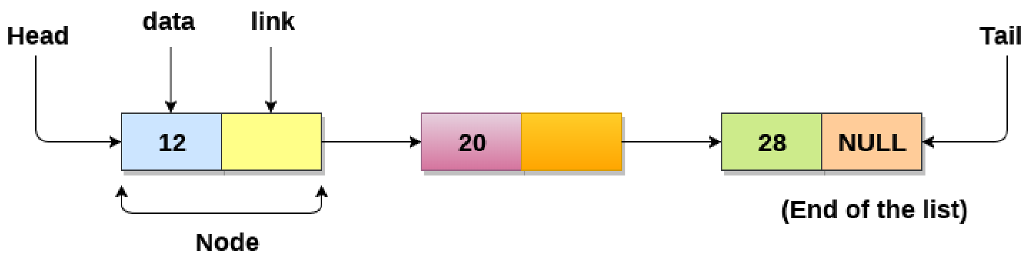
If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Linked List

- Linked List can be defined as collection of objects called **nodes** that arerandomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and thepointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The nodecan reside any where in the memory and linked together to make a list. Thisachieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

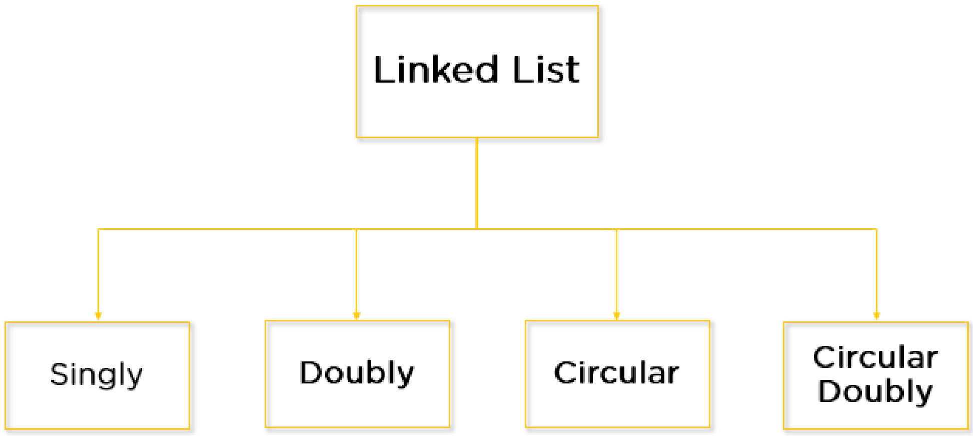
Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory.Inserting any element in the array needs shifting of all its predecessors.

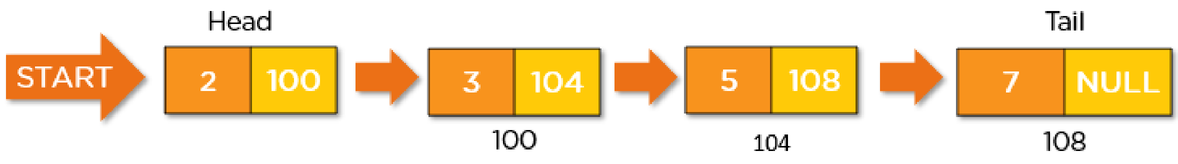
Linked list is the data structure which can overcome all the limitations of an array.Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at thetime of declaration. List grows as per the program's demand and limited to the available memory space.

Types Of Link List

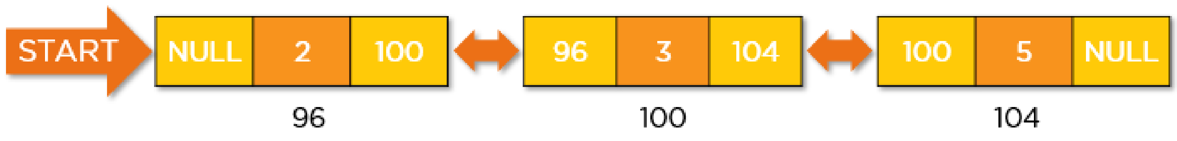


What is a Singly Linked List?



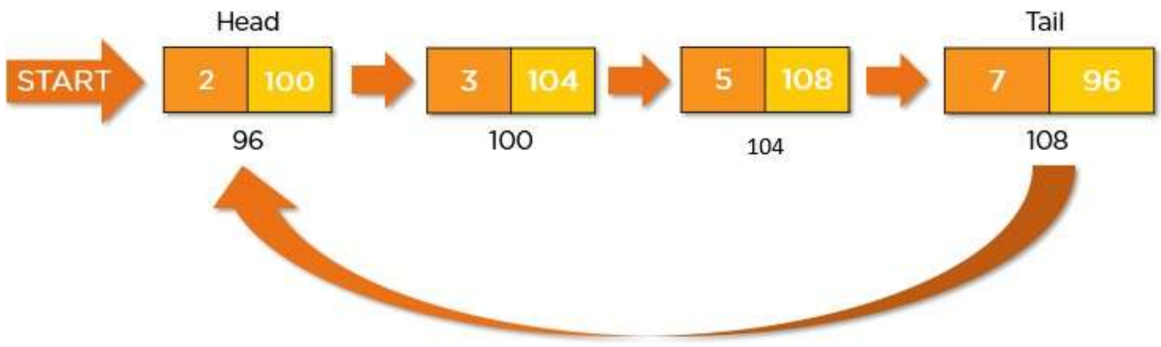
A singly linked list is a unidirectional linked list. So, you can only traverse it inone direction, i.e., from head node to tail node.

What is a Doubly Linked List?



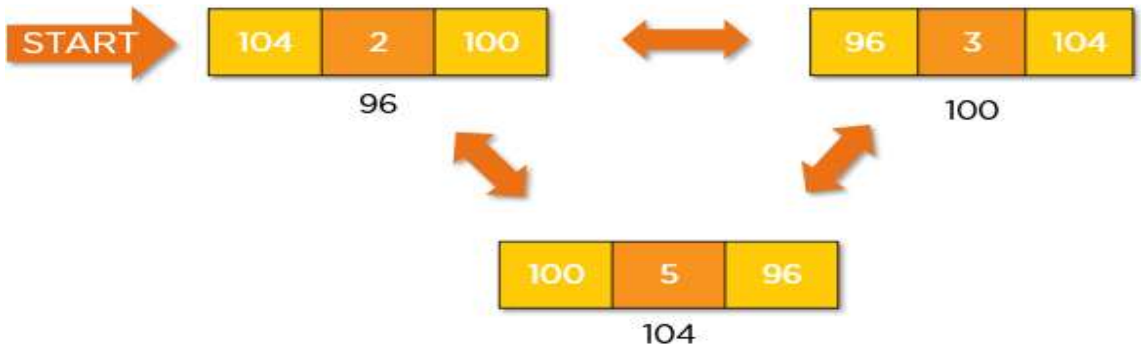
A doubly linked list is a bi-directional linked list. So, you can traverse it in bothdirections. Unlike singly linked lists, its nodes contain one extra pointer called the previous pointer. This pointer points to the previous node.

What is a Circular Linked List?



A circular Linked list is a unidirectional linked list. So, you can traverse it in only one direction. But this type of linked list has its last node pointing to the head node. So while traversing, you need to be careful and stop traversing whenyou revisit the head node.

What is a Circular Doubly Linked List?



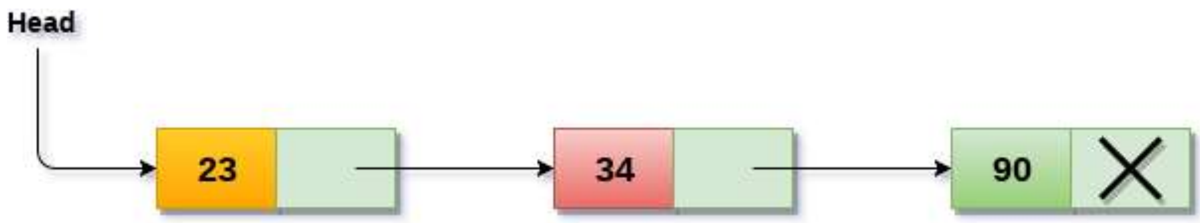
A circular doubly linked list is a mixture of a doubly linked list and a circular linked list. Like the doubly linked list, it has an extra pointer called the previouspointer, and similar to the circular linked list, its last node points at the head node. This type of linked list is the bi-directional list. So, you can traverse it in both directions.

1. Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. Thenumber of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the nodestores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In otherwords, we can say that each node contains only next pointer, therefore we can nottraverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last nodein the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked ListInsertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list.We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at endof the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

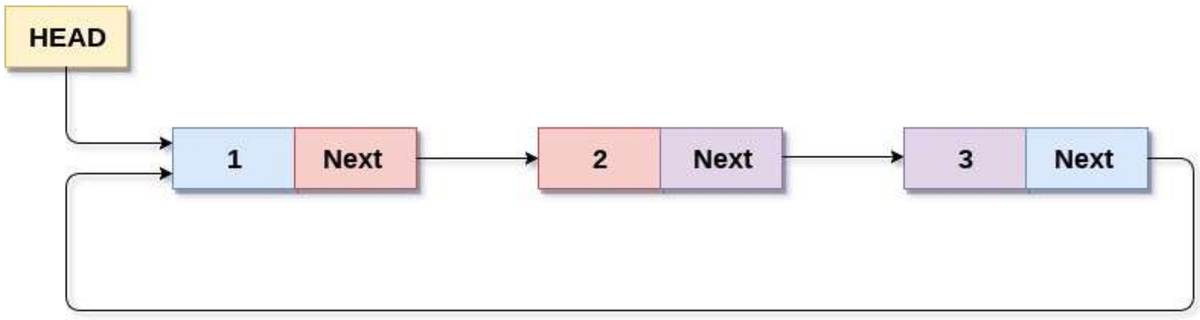
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

2. <u>Circular Singly Linked List</u>

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is nonull value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



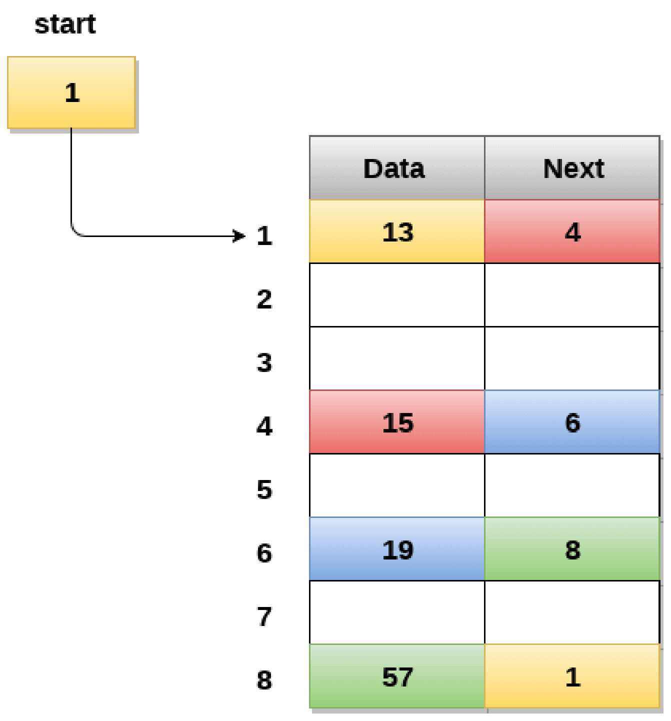
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containingmarks of a student in 4 subjects. However, the image shows a glimpse of how thecircular list is being stored in the memory. The start or head of the list is pointingto the element with the index 1 and containing 13 marks in the data part and 4 inthe next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can

find out the number of iterations which need to be performed while traversing thelist.

<u>Operations on Circular Singly linked list:</u>Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list atthe beginning.
2	Insertion at the end	Adding a node into circular singly linked list atthe end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list atthe beginning.
2	Deletion at the end	Removing the node from circular singly linked list atthe end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in orderto perform some specific operation.

Array vs. Link List

Array	Linked list
An array is a collection of elements of a similar data type.	A linked list is a collection of objects knownas a node where node consists of two parts, i.e., data and address.
Array elements store in a contiguous memory location.	Linked list elements can be stored anywherein the memory or randomly stored.
Array works with a static memory. Here static memory means that the memorysize is fixed and cannot be changed at therun time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address ofthe next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performingany operation like insertion, deletion, etc.
Accessing any element in an array is fasteras the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case ofa linked list as the memory can be allocated or deallocated at the run time according to our requirement.

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

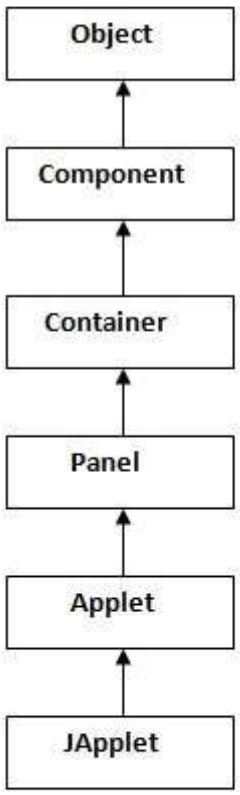
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

How to run an Applet?

There are two ways to run an applet

1. By html file.
2. By appletViewer tool (for testing purpose).

Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome",150,150);
}
}
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
c:\>appletviewer myapplet.html
```

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet{
public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);
g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer): is used draw the specified image.

How to get the object of Image:

The java.applet.Applet class provides getImage() method that returns the object of Image. Syntax:

```
public Image getImage(URL u, String image){}
```

Other required methods of Applet class to display image:

public URL getDocumentBase(): is used to return the URL of the document in which applet is embedded.

public URL getCodeBase(): is used to return the base URL.

Example of displaying image in applet:

```
import java.awt.*;
import java.applet.*;
public class DisplayImage extends Applet {
Image picture;
public void init() {
picture = getImage(getDocumentBase(),"sonoo.jpg");
}
public void paint(Graphics g) {
g.drawImage(picture, 30,30, this);
}
}
```

In the above example, drawImage() method of Graphics class is used to display the image. The 4th argume ImageObserver object. The Component class implements ImageObserver interface. So current class ob ImageObserver because Applet class indirectly extends the Component class.

myapplet.html

```
<html>
<body>
<applet code="DisplayImage.class" width="300" height="300">
</applet>
</body>
</html>
```

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

```
public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet{
public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```