MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: JAVA AND DATA STRUCTURES

SEMESTER: III

PREPARED BY: PROF W.SHARMILA

CORE–III JAVA AND DATA STRUCTURES

OBJECTIVES:

- To enable the students to learn the basic concepts of Javaprogramming
- To use class and objects to createapplications
- To have an overview of interfaces, packages, multithreading andexceptions.
- To familiarize students with basic data structures and their use inalgorithms.

OUTCOMES:

- Students will be able to develop Java Standalone applications andApplets.
- Choose the appropriate data structure for modeling a givenproblem.

UNIT - I

History and Evolution of Java - Features of Java - Object Oriented Concepts – Bytecode - Lexical Issues - Data Types – Variables- Type Conversion and Casting- Operators - Arithmetic Operators - Bitwise - Relational Operators - Assignment Operator - The conditional Operator - Operator Precedence- Control Statements – Arrays.

UNIT - II

Classes - Objects - Constructors - Overloading method - Static and fixed methods - Inner Classes - String Class- Overriding methods - Using super-Abstract class - this keyword – finalize() method – Garbage Collection.

UNIT - III

Packages - Access Protection - Importing Packages - Interfaces - Exception Handling - Throw and Throws-The Java Thread Model- Creating a Thread and Multiple Threads - Thread Priorities Synchronization-Inter thread Communication - Deadlock

- Suspending, Resuming and stopping threads – Multithreading-I/O Streams - File Streams - Applets .

UNIT - IV

Abstract Data Types(ADTs)-List ADT-Array based implementation-linked list implementation-singly linked list-doubly linked list-circular linked list-Stack ADT operations-Applications-Evaluating arithmetic expressions-Conversion of infix to postfix expression-Queue ADT-operations-Applications of Queues.

UNIT - V

Trees-Binary Trees- representation - Operations on Binary Trees- Traversal of a Binary Tree -Binary Search Trees, Graphs- Representation of Graphs - Traversal in Graph -Dijkstra's Algorithm, Depth-First vs Breadth-First Search.

TEXT BOOKS:

1. E.Balagurusamy," *Programming with Java: A Primer*", Tata McGraw Hill 2014, 5thEdition.

2. Mark Allen Weiss, "*Data Structures and Algorithms Analysis in C++*", Person Education 2014, 4thEdition.

REFERENCES:

. 1. Herbert Schildt, "*JAVA 2: The Complete Reference*", McGraw Hill 2018, 11thEdition.

2. Aho, Hopcroft and Ullman, "*Data Structures and Algorithms* ", Pearson Education2003.

3. S. Sahni, "*Data Structures, Algorithms and Applications in JAVA*", Universities Press 2005, 2ndEdition

WEB REFERENCES:

- NPTEL & MOOC courses titled Java and DataStructures
- https://nptel.ac.in/courses/106106127/
- https://nptel.ac.in/courses/106105191/

UNIT I

Java is an object-oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. Unlike conventional languages which are generally designed either to be compiled to native (machine) code, or to be interpreted from source code at runtime, Java is intended to be compiled to a bytecode, which is then run (generally using JIT compilation) by a Java Virtual Machine

The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java is only distantly related to JavaScript, though they have similar names and share a C-like syntax.

History

Java was started as a project called "Oak" by James Gosling in June 1991. Gosling's goals were to implement a virtual machine and a language that had a familiar C-like notation but with greater uniformity and simplicity than C/C++. The first public implementation was Java 1.0 in 1995. It made the promise of "Write Once, Run Anywhere", with free runtimes on popular platforms. It was fairly secure and its security was configurable, allowing for network and file access to be limited. The major web browsers soon incorporated it into their standard configurations in a secure "applet" configuration. popular quickly. New versions for large and small platforms (J2EE and J2ME) soon were designed with the advent of "Java 2". Sun has not announced any plans for a "Java 3".

In 1997, Sun approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a proprietary de facto standard that is controlled through the Java Community Process. Sun makes most of its Java implementations available without charge, with revenue being generated by specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) which is a subset of the SDK, the primary distinction being that in the JRE the compiler is not present.

Philosophy

There were five primary goals in the creation of the Java language:

1. It should use the object-oriented programming methodology

2. It should allow the same program to be executed onmultiple operating systems.

3. It should contain built-in support for using computer networks.

4. It should be designed to execute code from remote sources securely.
5. It should be easy to use by selecting what was considered the good parts of other object-orientedlanguages.

To achieve the goals of networking support and remote code execution, Java programmers sometimes find it necessary to use extensions such as CORBA, Internet Communications Engine, or OSGi.

Object orientation

The first characteristic, object orientation ("OO"), refers to a method of programming and language design. Although there are many interpretations of OO, one primary distinguishing idea is to design software so that the various types of data it manipulates are combined together with their relevant operations. Thus, data and code are combined into entities called objects. An object can be thought of as a self-contained bundle of behavior (code) and state (data). The principle is to separate the things that change from the things that stay the same; often, a change to some data structure requires a corresponding change to the code that operates on that data, or vice versa. This separation into coherent objects provides a more stable foundation for a software system's design. The intent is to make large software projects easier to manage, thus improving quality and reducing the number of failed projects.

Another primary goal of OO programming is to develop more generic objects so that software can become more reusable between projects. A generic "customer" object, for example, should have roughly the same basic set of behaviors between different software projects, especially when these projects overlap on some fundamental level as they often do in large organizations. In this sense, software objects can hopefully be seen more as pluggable components, helping the software industry build projects largely from existing and well-tested pieces, thus leading to a massive reduction in development times. Software reusability has met with mixed practical results, with two main difficulties: the design of truly generic objects is poorly understood, and a methodology for broad communication of reuse opportunities is lacking. Some open source communities want to help ease the reuse problem, by providing authors with ways to disseminate information about generally reusable objects and object libraries.

Platform independence

The second characteristic, platform independence, means that programs written in the Java language must run similarly on diverse hardware. One should be able to write a program once

This is achieved by most Java compilers by compiling the Java language code "halfway" to bytecode (specifically Java bytecode)—simplified machine instructions specific to the Java platform. The code is then run on a virtual machine (VM), a program written in native code on the host hardware that interprets and executes generic Java bytecode. Further, standardized libraries are provided to allow access to features of the host machines (such as graphics, threading and networking) in unified ways. Note that, although there's an explicit compiling stage, at some point, the Java bytecode is interpreted or converted to native machine instructions by the JIT compiler.

There are also implementations of Java compilers that compile to native object code, such as GCJ, removing the intermediate bytecode stage, but the output of these compilers can only be run ona single architecture.

Sun's license for Java insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support the RMI and JNI interfaces and had added platform-specific features of their own. In response, Microsoft no longer ships Java with Windows, and in recent versions of Windows, Internet Explorer cannot support Java applets without a third-party plug-in. However, Sun and others have made available Java run-time systems at no cost for those and other versions of Windows.

The first implementations of the language used an interpreted virtual machine to achieve portability. These implementations produced programs that ran more slowly than programs compiled to native executables, for instance written in C or C++, so the language suffered a reputation for poor performance. More recent JVMimplementations produce programs that run significantly faster than before, using multiple techniques.

The first technique is to simply compile directly into native code like a more traditional compiler, skipping bytecodes entirely. This achieves good performance, but at the expense of portability. Another technique, known as just-in-time compilation (JIT), translates the Java bytecodes into native code at the time that the program is run which results in a program that executes faster than interpreted code but also incurs compilation overhead during execution. More sophisticated VMs use dynamic recompilation, in which the VM can analyze the behavior of the running program and selectively recompile and optimize critical parts of the program. Dynamic recompilation can

achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes. JIT compilation and dynamic recompilation allow Java programs to take advantage of the speed of native code without losing portability.

Portability is a technically difficult goal to achieve, and Java's success at that goal has been mixed. Although it is indeed possible to write programs for the Java platform that behave consistently across many host platforms, the large number of available platforms with small errors or inconsistencies led some to parody Sun's "Write once, run anywhere" slogan as "Write once, debug everywhere".

Platform-independent Java is however very successful with server-side applications, such as Web services,servlets, and Enterprise JavaBeans, as well as with Embedded systems based on OSGi, using Embedded Java environments.

Automatic garbage collection

One idea behind Java's automatic memory management model is that programmers should be spared the burden of having to perform manual memory management. In some languages the programmer allocates memory to create any object stored on the heap and is responsible for later manually deallocating that memory to delete any such objects. If a programmer forgets to deallocate memory or writes code that fails to do so in a timely fashion, a memory leak can occur: the program will consume a potentially arbitrarily large amount of memory. In addition, if a region of memory is deallocated twice, the program can become unstable and may crash. Finally, in non garbage collected environments, there is a certain degree of overhead and complexity of user-code to track and finalize allocations.

In Java, this potential problem is avoided by automatic garbage collection. The programmer determines when objects are created, and the Java runtime is responsible for managing the object's lifecycle. The program or other objects can reference an object by holding a reference to it (which, from a low-level point of view, is its address on the heap). When no references to an object remain, the Java garbage collector automatically deletes the unreachable object, freeing memory and preventing a memory leak. Memory leaks may still occur if a programmer's code holds a reference to an object that is no longer needed—in other words, they can still occur but at higher conceptual levels.

The use of garbage collection in a language can also affect programming paradigms. If, for example, the developer assumes that the cost of memory allocation/recollection is low, they may choose to more freely construct objects instead of pre-initializing, holding and reusing them. With the small cost of potential performance penalties (inner-loop construction of large/complex objects), this facilitates thread-isolation (no need to synchronize as different threads work on different object instances) and data-hiding. The use of transient immutable value-objects minimizes side-effect programming.

Features of JAVA

Object Oriented

In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Platform Independent

Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform-independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

Simple Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.

Secure

With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architecture-neutral

Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Portable

Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. The compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

Robust

Java makes an effort to eliminate error-prone situations by emphasizing mainly on compile time error checking and runtime checking.

Multithreaded

With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

Interpreted

Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

High Performance

With the use of Just-In-Time compilers, Java enables high performance.

Distributed

Java is designed for the distributed environment of the internet.

Dynamic

Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry an extensive amount of run-time information that can be used to verify and resolve accesses to objects at run-time.

Java lexical structure

Computer languages, like human languages, have a lexical structure. A source code of a Java program consists of tokens. Tokens are atomic code elements. In Java we have comments, identifiers, literals, operators, separators, and keywords.

Java programs are composed of characters from the Unicode character set.

Java comments

Comments are used by humans to clarify source code. There are three types of comments in Java.

Comment type	**Meaning**
// comment	Single-line comments
/* comment */	Multi-line comments
/** documentation */	Documentation comments

Java white space

White space in Java is used to separate tokens in the source file. It is also used to improve readability of the source code.

```
int i = 0;
```

White spaces are required in some places. For example between the int keyword and the variable name. In other places, white spaces are forbidden. They cannot be present in variable identifiers or language keywords.

```
int a=1;

int b = 2;

int c  =  3;
```

The amount of space put between tokens is irrelevant for the Java compiler. The white space should be used consistently in Java source code.

Java identifiers

Identifiers are names for variables, methods, classes, or parameters. Identifiers can have alphanumerical characters, underscores and dollar signs ($). It is an error to begin a variable name with a number. White space in names is not permitted.

Identifiers are case sensitive. This means that Name, name, or NAME refer to three different variables. Identifiers also cannot match language keywords.

There are also conventions related to naming of identifiers. The names should be descriptive. We should not use cryptic names for our identifiers. If the name consists of multiple words, each subsequent word is capitalized.

```
String name23;

int _col;

short car_age;
```

These are valid Java identifiers.

```
String 23name;

int %col;

short car age;
```

These are invalid Java identifiers.

Java literals

A *literal* is a textual representation of a particular value of a type. Literal types include boolean, integer, floating point, string, null, or character. Technically, a literal will be assigned a value at compile time, while a variable will be assigned at runtime.

Java operators

An *operator* is a symbol used to perform an action on some value. Operators are used in expressions to describe operations involving one or more operands.

+ - * / % ^ & | ! ~

= += -= *= /= %= ^= ++ --

== != <>&= >>= <<= >= <=

|| &&>><< ?:

This is a partial list of Java operators.

Java separators

A *separator* is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data stream.

[ ] ( ) { } , ; . "

String language = "Java";

The double quotes are used to mark the beginning and the end of a string. The semicolon ; character is used to end each Java statement.

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

Parentheses (round brackets) always follow a method name. Between the parentheses we declare the input parameters. The parentheses are present even if the method does not take any parameters. The System.out.println() method takes one parameter, a string value. The dot character separates the class name (System) from the member (out) and the member from the method name (println()).

int[] array = new int[5] { 1, 2, 3, 4, 5 };

The square brackets [] are used to denote an array type. They are also used to access or modify array elements. The curly brackets {} are used to initiate arrays. The curly brackets are also used enclose the body of a method or a class.

int a, b, c;

The comma character separates variables in a single declaration.

Java keywords

A keyword is a reserved word in Java language. Keywords are used to perform a specific task in the computer program. For example, to define variables, do repetitive tasks or perform logical operations.

Java is rich in keywords.

abstract continue for new switch

assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	var
class	finally	long	strictfp	void
const	float	native	super	volatile
				while

Java conventions

Each language can have its own set of conventions. Conventions are not strict rules; they are merely recommendations for writing good quality code. We mention a few conventions that are recognized by Java programmers. (And often by other programmers too).

Class names begin with an uppercase letter.

Method names begin with a lowercase letter.

The public keyword precedes the static keyword when both are used.

The parameter name of the main() method is called args.

Constants are written in uppercase.

Each subsequent word in an identifier name begins with a capital letter.

Data Types in Java

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

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

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

Java Primitive Data Types

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

There are 8 types of primitive data types:

boolean data type

byte data type

char data type

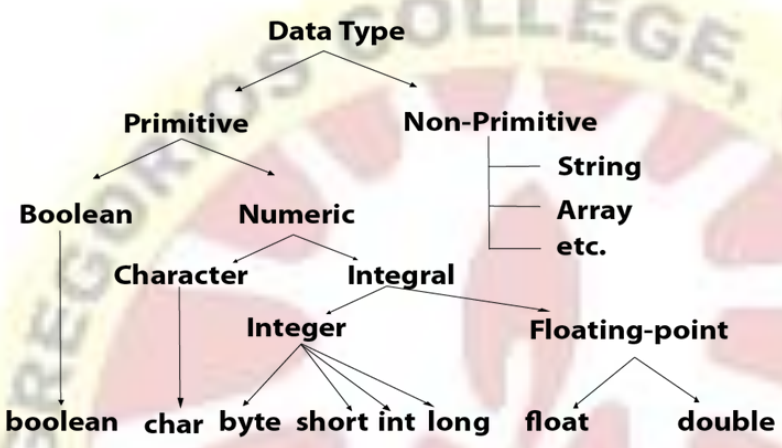
short data type

int data type

long data type

float data type

double data type



Data Type	Default Value	Default size
Boolean	false	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

Boolean Data Type

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

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

Example: Boolean one = false

Byte Data Type

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

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

Example: byte a = 10, byte b = -20

Short Data Type

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

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

Example: short s = 10000, short r = -5000

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

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'

Java Variables

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

String - stores text, such as "Hello". String values are surrounded by double quotes

int - stores integers (whole numbers), without decimals, such as 123 or -123

float - stores floating point numbers, with decimals, such as 19.99 or -19.99

char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes

boolean - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax

type variable = value;

Where *type* is one of Java's types (such as int or String), and *variable* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type String and assign it the value "**John**":

```
String name ="John";

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

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type int and assign it the value **15**:

```
int myNum =15;

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

Example

```
int myNum;

myNum =15;

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

Example

Change the value of myNum from 15 to 20:

```
int myNum =15;

myNum =20;// myNum is now 20

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

Final Variables

we can add the final keyword if we don't want others to overwrite existing values (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

Example

```
finalint myNum =15;
```

```
myNum =20;// will generate an error: cannot assign a value to a final variable
```

Java Identifiers

All Java **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
// Good

int minutesPerHour =60;



// OK, but not so easy to understand what m actually is

int m =60;
```

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as int or boolean) cannot be used as names

Difference Between Type Casting and Type Conversion

S.N.	Type Casting	Type Conversion
1	Type casting is a mechanism in which one data type is converted to another data type using a casting () operator by a programmer.	Type conversion allows a compiler to convert one data type to another data type at the compile time of a program or code.
2	It can be used both compatible data type and incompatible data type.	Type conversion is only used with compatible data types, and hence it does not require any casting operator.
3	It requires a programmer to manually casting one data into	It does not require any programmer intervention to convert one

	another type.	data type to another because the compiler automatically compiles it at the run time of a program.
4	It is used while designing a program by the programmer.	It is used or take place at the compile time of a program.
5	When casting one data type to another, the destination data type must be smaller than the source data.	When converting one data type to another, the destination type should be greater than the source data type.
6	It is also known as narrowing conversion because one larger data	It is also known as widening conversion because one smaller data

	type converts to a smaller data type.	type converts to a larger data type.
7	It is more reliable and efficient.	It is less efficient and less reliable.
8	There is a possibility of data or information being lost in type casting.	In type conversion, data is unlikely to be lost when converting from a small to a large data type.
8	float b = 3.0; int a = (int) b	int x = 5, y = 2, c; float q = 12.5, p; p = q/x;

Operators in Java

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators

5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators
9. instance of operator

- Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.
- * : Multiplication
- / : Division
- % : Modulo
- + : Addition
- – : Subtraction
- **Unary Operators:** Unary operators need only one operand. They are used to increment, decrement or negate a value.
- **– :Unary minus**, used for negating the values.
- **+ :Unary plus**, indicates positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is byte, char, or short. This is called unary numeric promotion.
- **++ :Increment operator**, used for incrementing the value by 1. There are two varieties of increment operator.
- **Post-Increment :** Value is first used for computing the result and then incremented.
- **Pre-Increment :** Value is incremented first and then result is computed.
- **— : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operator.
- **Post-decrement :** Value is first used for computing the result and then decremented.
- **Pre-Decrement :** Value is decremented first and then result is computed.
- **! : Logical not operator**, used for inverting a boolean value.
- **Assignment Operator : '='** Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant.

General format of assignment operator is,

variable = value;

- In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a = a+5, we can write a += 5.
- **+=**, for adding left operand with right operand and then assigning it to variable on the left.
- **-=**, for subtracting left operand with right operand and then assigning it to variable on the left.
- ***=**, for multiplying left operand with right operand and then assigning it to variable on the left.
- **/=**, for dividing left operand with right operand and then assigning it to variable on the left.
- **%=**, for assigning modulo of left operand with right operand and then assigning it to variable on the left.
- **Relational Operators :** These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements. General format is,

variable **relation_operator** value

- Some of the relational operators are-
- **==, Equal to :** returns true if left hand side is equal to right hand side.
- **!=, Not Equal to :** returns true if left hand side is not equal to right hand side.
- **<, less than :** returns true if left hand side is less than right hand side.
- **<=, less than or equal to :** returns true if left hand side is less than or equal to right hand side.
- **>, Greater than :** returns true if left hand side is greater than right hand side.
- **>=, Greater than or equal to:** returns true if left hand side is greater than or equal to right hand side.

- **Logical Operators :** These operators are used to perform "logical AND" and "logical OR" operation, i.e. the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Conditional operators are-
  - **&&, Logical AND :** returns true when both conditions are true.
  - **||, Logical OR :** returns true if at least one condition is true.
- **Ternary operator :** Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. General format is-

condition **?** if true **:** if false

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

- **Bitwise Operators :** These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.
  - **&, Bitwise AND operator:** returns bit by bit AND of input values.
  - **|, Bitwise OR operator:** returns bit by bit OR of input values.
  - **^, Bitwise XOR operator:** returns bit by bit XOR of input values.
  - **~, Bitwise Complement Operator:** This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.
- **Shift Operators :** These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two. General format-

number **shift_op** number_of_places_to_shift;

  - **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.

- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.
- **instance of operator :** Instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface. General format-

object **instance of** class/subclass/interface

Precedence and Associativity of Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude with the top representing the highest precedence and bottom shows the lowest precedence.

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Control Flow in Java

Java compiler executes the java code from top to bottom. The statements 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.

Java provides three types of control flow statements.

1. Decision Making statements
2. Loop statements
3. Jump statements

Decision-Making statements:

Decision-making statements evaluate the Boolean expression and control the program flow depending upon the condition result. There are two types of decision-making statements in java, I.e., If statement and switch statement.

If Statement:

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

1. if statement
2. if-else statement
3. else-if statement
4. Nested if-statement

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

1. if statement:

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

Syntax of if statement is given below.

1. **if**(<condition>) {

```
2.      //block of code
3.      }
4.
5.      Consider the following example in which we have used the if statement i
n the java code.
6.
7.      public class Student {
8.      public static void main(String[] args) {
1.      int x = 10;
9.      int y = 12;
10.     if(x+y > 20) {
11.     System.out.println("x + y is greater than                    20");
12.     }
13.     }
14.
15.
16.     }
```

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.

Consider the following example.

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

```
8.      System.out.println("x + y is greater than 20");
9.      }
10.     }
```

Output:

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

3. lse-if statement

The 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 any block of code. We can also define an else statement at the end of the chain.

Consider the following example.

```
1.      public class Student {
2.      public static void main(String[] args) {
3.      String city = "Delhi";
4.      if(city == "Meerut") {
5.      System.out.println("city is meerut");
6.      }else if (city == "Noida") {
7.      System.out.println("city is noida");
8.      }else if(city == "Agra") {
9.      System.out.println("city is agra");
10.     }else {
11.     System.out.println(city);
12.     }
13.     }
14.     }
15.
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement contains multiple if-else statements as a separate block of code. Consider the following example.

```
1.  public class Student {
2.  public static void main(String[] args) {
3.  String address = "Delhi, India";
4.
5.  if(address.endsWith("India")) {
6.  if(address.contains("Meerut")) {
7.  System.out.println("Your city is meerut");
8.  }else if(address.contains("Noida")) {
9.  System.out.println("Your city is noida");
10. }else {
11. System.out.println(address.split(",")[0]);
12. }
13. }else {
14. System.out.println("You are not living in india");
15. }
16. }
17. }
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement enables us to check the variable for the range of values defined for multiple case statements. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program. The syntax to use the switch statement is given below.

```
1.  switch <variable> {
2.  Case <option 1>:
3.  //block of statements
4.  ..
5.  ..
6.  ..
7.  Case <option n>:
8.  //block of statements
```

```
9.      Default:
10.     //block of statements
11.     }
```

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

```
1.      public class Student implements Cloneable {
2.      public static void main(String[] args) {
3.      int num = 2;
4.      switch (num){
5.      case 0:
6.      System.out.println("number is 0");
7.      break;
8.      case 1:
9.      System.out.println("number is 1");
10.     break;
11.     default:
12.     System.out.println(num);
13.     }
14.     }
15.     }
```

Output:

```
2
```

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

Loop Statements

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

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

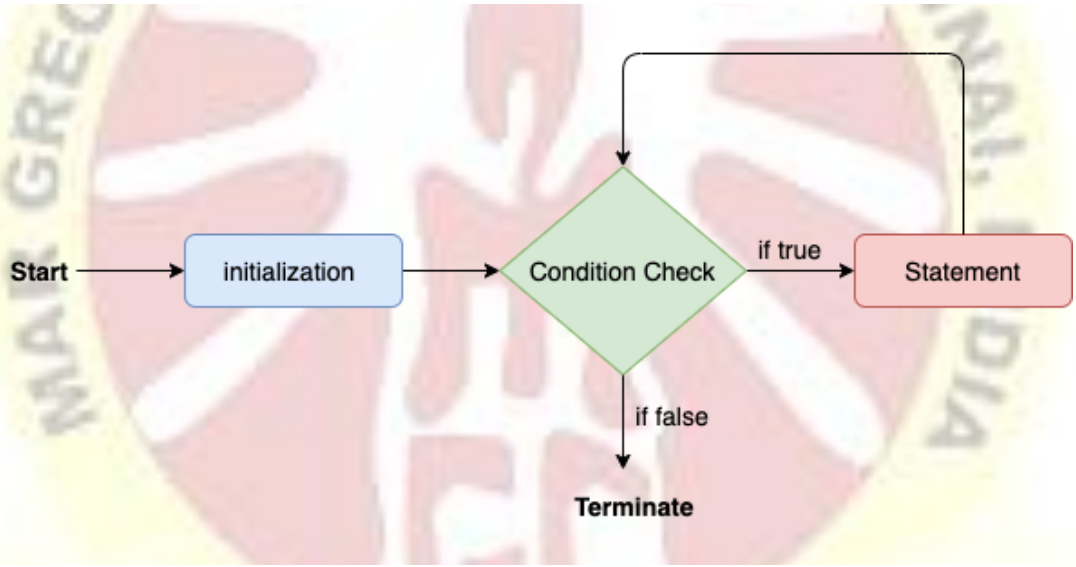
1. for loop
2. while loop
3. do-while loop

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. The syntax to use the for loop is given below.

```
1. for(<initialization>, <condition>, <increment/decrement>) {
2. //block of statements
3. }
```

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



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

```
1. public class Calculattion {
2. public static void main(String[] args) {
3.
4. int sum = 0;
5. for(int j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
```

```
9.      }
10.     }
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

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

```
1.      for(type var : collection){
2.      //statements
3.      }
```

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

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

Output:

```
Printing the content of the array names:


Java
C
C++
Python
JavaScript
```

Java while loop

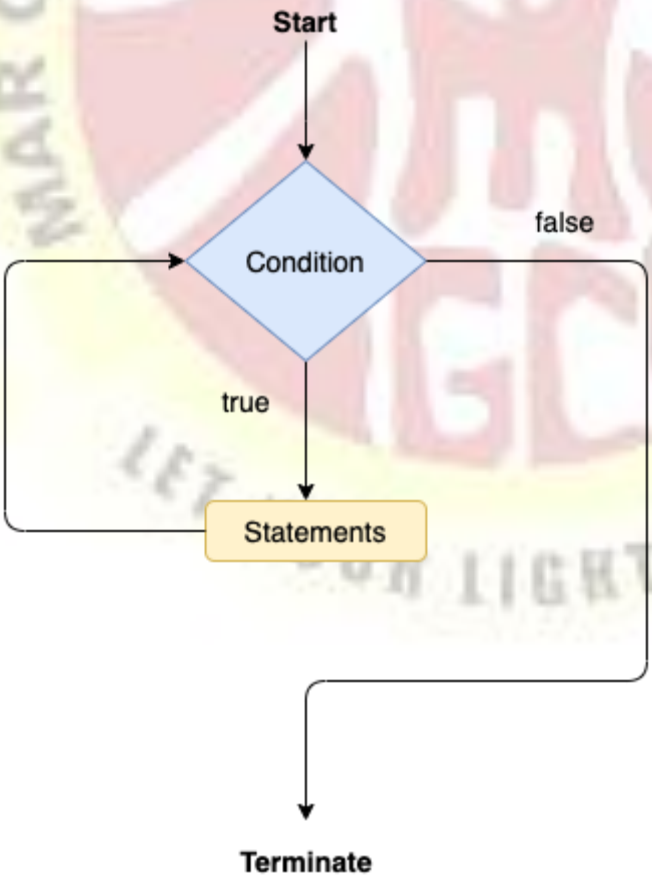
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while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

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

The syntax of the while loop is given below.

```
1.  While(<condition>){
2.  //loop statements
3.  }
```

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



Consider the following example.

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

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

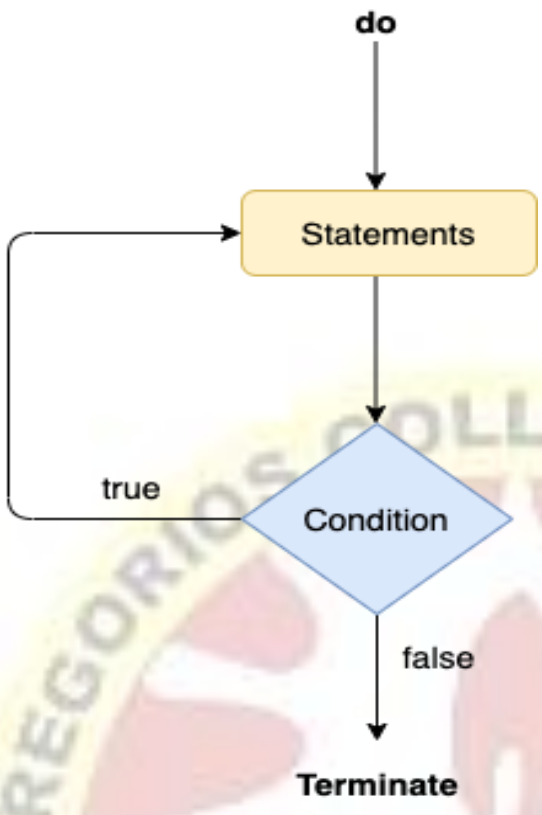
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. However, it is recommended to use the do-while loop if we don't know the condition in advance, and we need the loop to execute at least once.

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

```
1.  do
2.  {
3.  //statements
4.  } while (<Condition>);
```

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



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

```
1.  public class Calculattion {
2.  public static void main(String[] args) {
3.  //int i = 0;
4.  System.out.println("Printing the list of first 10 even numbers \n");
5.  do {
6.  System.out.println(i);
7.  i = i + 2;
8.  }while(i<=10);
9.  }
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 the current flow. It is used to break the loop and 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 loop

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

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

Output:

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

break statement example with labeled for loop

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

Output:

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

Java continue statement

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

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

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

Output:

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

Java Arrays

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

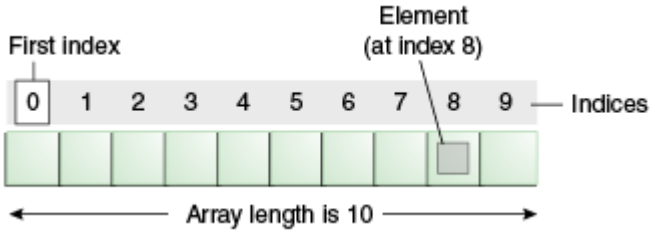
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

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

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

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

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



Advantages

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

Disadvantages

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

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

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

Instantiation of an Array in Java

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

Example of Java Array

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

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
```

```
6.  a[0]=10;//initialization
7.  a[1]=20;
8.  a[2]=70;
9.  a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Output:

```
10
20
70
40
50
```

Multidimensional Array in Java

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

Syntax to Declare Multidimensional Array in Java

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

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

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

Example of Multidimensional Java Array

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

```
1.  //Java Program to illustrate the use of multidimensional array
2.  class Testarray3{
3.  public static void main(String args[]){
4.  //declaring and initializing 2D array
5.  int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6.  //printing 2D array
7.  for(int i=0;i<3;i++){
8.   for(int j=0;j<3;j++){
9.     System.out.print(arr[i][j]+" ");
10.   }
11.   System.out.println();
12.  }
13.  }}
```

Output:

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

Unit II

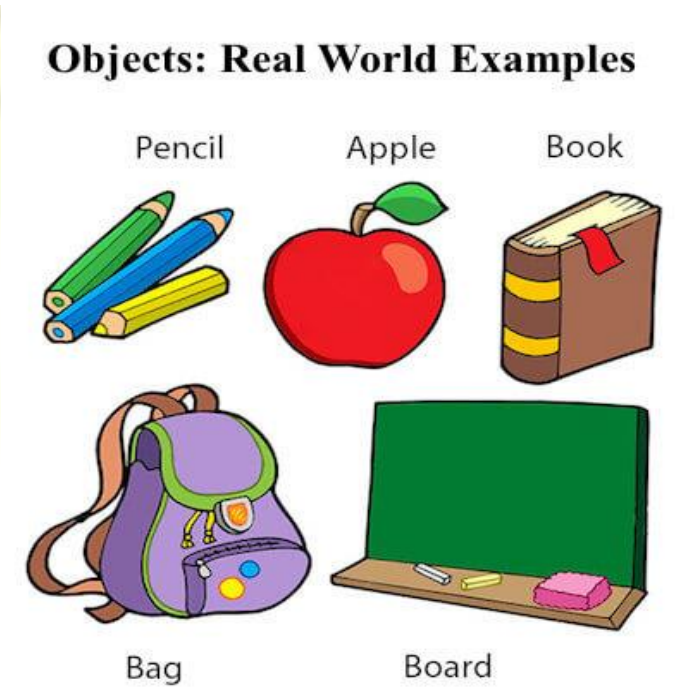
Objects and Classes in Java

1. Object in Java
2. Class in Java
3. Instance Variable in Java
4. Method in Java
5. Example of Object and class that maintains the records of student
6. Anonymous Object

In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

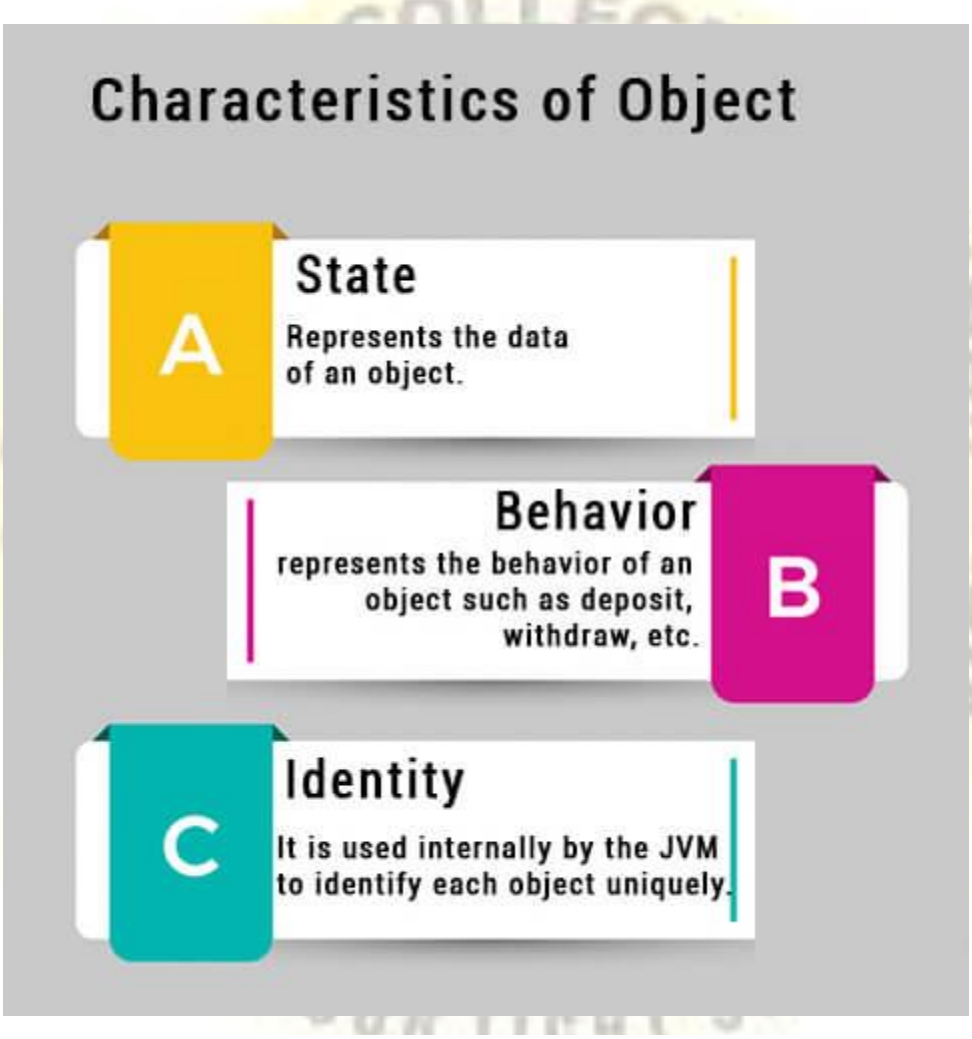
What is an object in Java



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

An object has three characteristics:

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



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

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

Object Definitions:

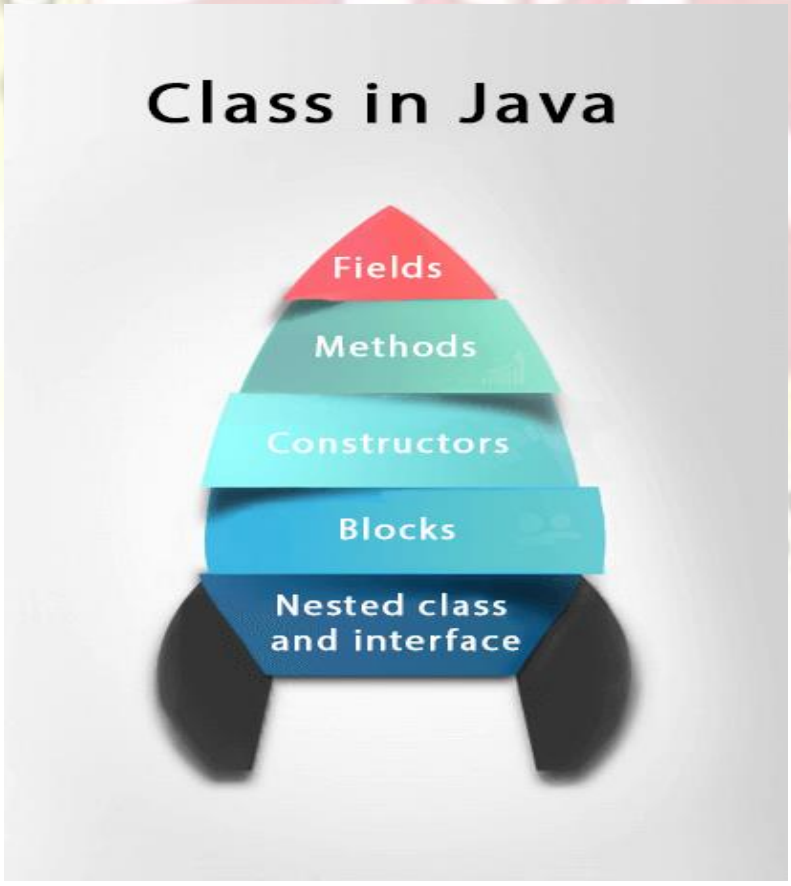
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

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

A class in Java can contain:

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



Syntax to declare a class:

1. **class** <class_name>{

```
2.      field;
3.      method;
4.  }
```

Instance variable in Java

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

Method in Java

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

Advantage of Method

- Code Reusability
- Code Optimization

---

new keyword in Java

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

Object and Class Example: main within the class

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

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

File: Student.java

```
1.  //Java Program to illustrate how to define a class and fields
2.  //Defining a Student class.
3.  class Student{
4.   //defining fields
```

```
5.      int id;//field or data member or instance variable
6.      String name;
7.      //creating main method inside the Student class
8.      public static void main(String args[]){
9.       //Creating an object or instance
10.      Student s1=new Student();//creating an object of Student
11.      //Printing values of the object
12.      System.out.println(s1.id);//accessing member through reference varia
ble
13.      System.out.println(s1.name);
14.     }
15.    }
```

Output:

```
0
null
```

Object and Class Example: main outside the class

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

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

File: TestStudent1.java

```
1.    //Java Program to demonstrate having the main method in
2.    //another class
3.    //Creating Student class.
4.    class Student{
5.     int id;
6.     String name;
7.    }
8.    //Creating another class TestStudent1 which contains the main method
9.    class TestStudent1{
```

```
10.     public static void main(String args[]){
11.      Student s1=new Student();
12.      System.out.println(s1.id);
13.      System.out.println(s1.name);
14.     }
15.    }
```

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

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

1) Object and Class Example: Initialization through reference

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

File: TestStudent2.java

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

```
12.     }
```

Output:

```
101 Sonoo
```

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

File: TestStudent3.java

```
1.      class Student{
2.       int id;
3.       String name;
4.      }
5.      class TestStudent3{
6.       public static void main(String args[]){
7.        //Creating objects
8.        Student s1=new Student();
9.        Student s2=new Student();
10.       //Initializing objects
11.       s1.id=101;
12.       s1.name="Sonoo";
13.       s2.id=102;
14.       s2.name="Amit";
15.       //Printing data
16.       System.out.println(s1.id+" "+s1.name);
17.       System.out.println(s2.id+" "+s2.name);
18.      }
19.     }
```

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

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

File: TestStudent4.java

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

Output:

```
111 Karan
222 Aryan
```

Object and Class Example: Employee

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

File: TestEmployee.java

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

Test it Now

Output:

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

Object and Class Example: Rectangle

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

File: TestRectangle1.java

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

Test it Now

Output:

```
55
45
```

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

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

Constructors in Java

1. Types of constructors

1. Default Constructor
2. Parameterized Constructor
2. Constructor Overloading
3. Does constructor return any value?
4. Copying the values of one object into another
5. Does constructor perform other tasks instead of the initialization

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

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

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

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

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

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

Rules for creating Java constructor

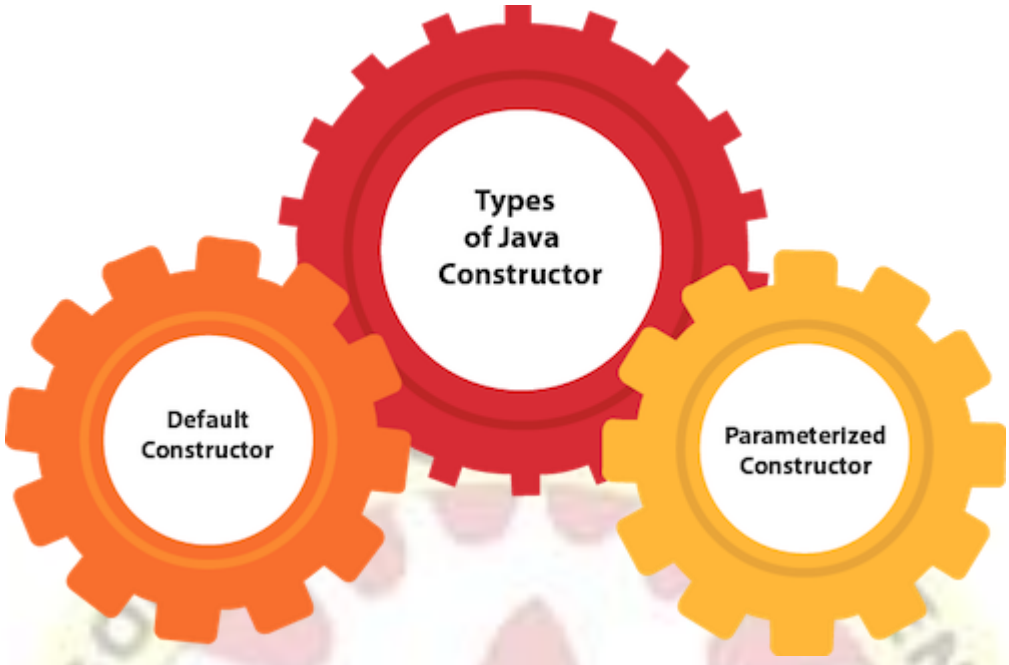
There are two rules defined for the constructor.

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

Types of Java constructors

There are two types of constructors in Java:

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



Java Default Constructor

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

Syntax of default constructor:

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

Example of default constructor

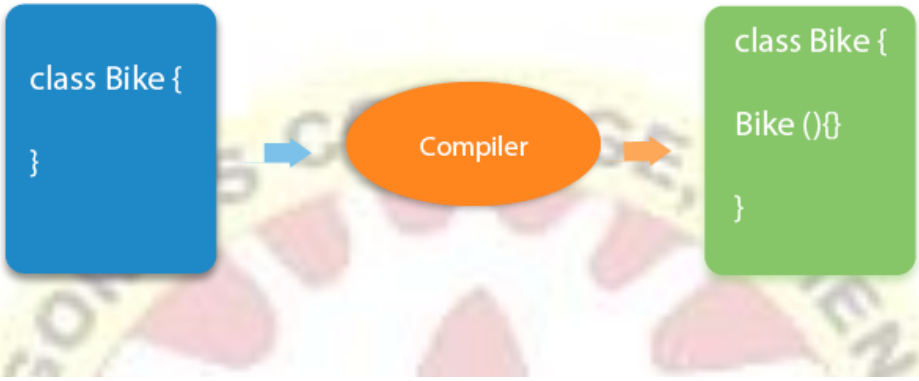
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1.  //Java Program to create and call a default constructor
2.  class Bike1{
3.  //creating a default constructor
4.  Bike1(){System.out.println("Bike is created");}
5.  //main method
6.  public static void main(String args[]){
7.  //calling a default constructor
8.  Bike1 b=new Bike1();
9.  }
10. }
```

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

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

Output:

0 null

0 null

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

Java Parameterized Constructor

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

Why use the parameterized constructor?

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

Example of parameterized constructor

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

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

```
19.         s2.display();
20.        }
21.    }
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

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

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

Example of Constructor Overloading

```
1.  //Java program to overload constructors
2.  class Student5{
3.     int id;
4.     String name;
5.     int age;
6.     //creating two arg constructor
7.     Student5(int i,String n){
8.     id = i;
9.     name = n;
10.    }
11.    //creating three arg constructor
12.    Student5(int i,String n,int a){
13.    id = i;
14.    name = n;
15.    age=a;
16.    }
17.    void display(){System.out.println(id+" "+name+" "+age);}
18.
```

```
19.     public static void main(String args[]){
20.     Student5 s1 = new Student5(111,"Karan");
21.     Student5 s2 = new Student5(222,"Aryan",25);
22.     s1.display();
23.     s2.display();
24.     }
25.  }
```

Output:

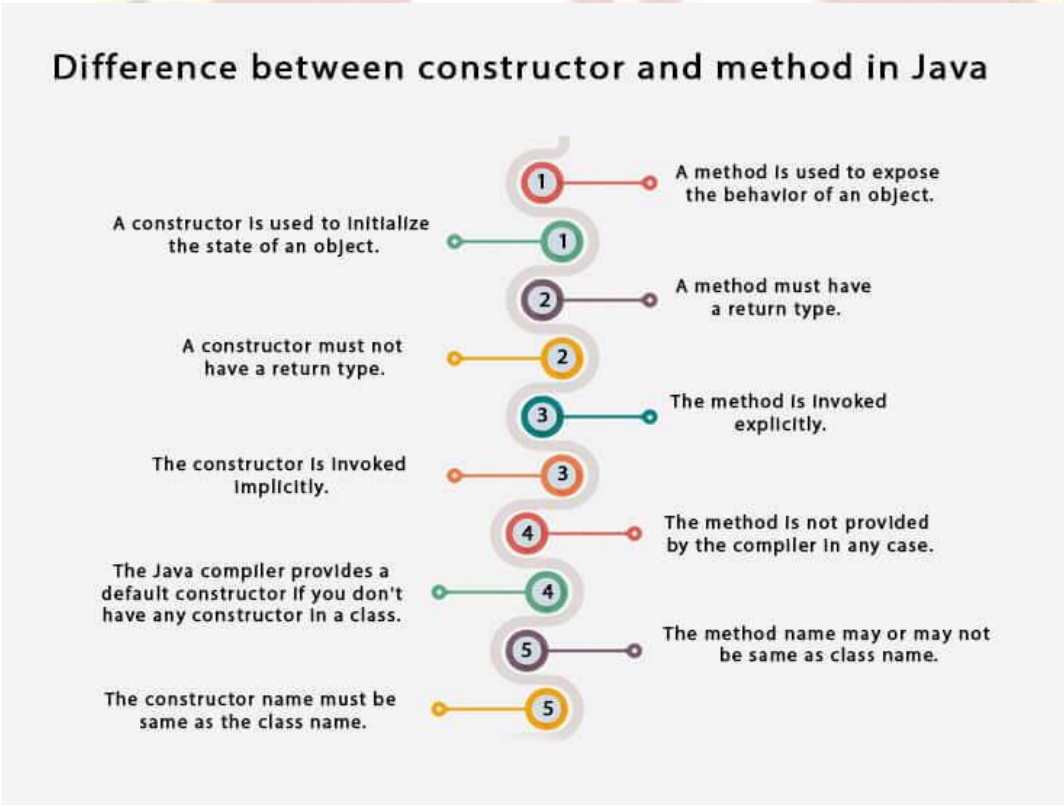
```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in Java

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

Java Constructor	JavaMethod
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.



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.

```
1.  //Java program to initialize the values from one object to another object.
2.  class Student6{
3.      int id;
4.      String name;
5.      //constructor to initialize integer and string
6.      Student6(int i,String n){
7.      id = i;
8.      name = n;
9.      }
10.     //constructor to initialize another object
11.     Student6(Student6 s){
12.     id = s.id;
13.     name =s.name;
14.     }
15.     void display(){System.out.println(id+" "+name);}
16.
17.     public static void main(String args[]){
18.     Student6 s1 = new Student6(111,"Karan");
19.     Student6 s2 = new Student6(s1);
20.     s1.display();
21.     s2.display();
22.    }
23. }
```

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.

```
1.      class Student7{
2.         int id;
3.         String name;
4.         Student7(int i,String n){
5.        id = i;
6.         name = n;
7.         }
8.         Student7(){}
9.         void display(){System.out.println(id+" "+name);}
10.
11.        public static void main(String args[]){
12.        Student7 s1 = new Student7(111,"Karan");
13.        Student7 s2 = new Student7();
14.        s2.id=s1.id;
15.        s2.name=s1.name;
16.        s1.display();
17.        s2.display();
18.       }
19.    }
```

Output:

```
111 Karan
111 Karan
```

Method Overloading in Java

1. Different ways to overload the method

2. By changing the no. of arguments
3. By changing the datatype
4. Why method overloading is not possible by changing the return type
5. Can we overload the main method
6. method overloading with Type Promotion

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

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

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

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

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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

1) Method Overloading: changing no. of arguments

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

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

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
```

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

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

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

Output:

```
22
24.9
```

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

```
8.      }}
```

Output:

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

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

Overloading of java main() method

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

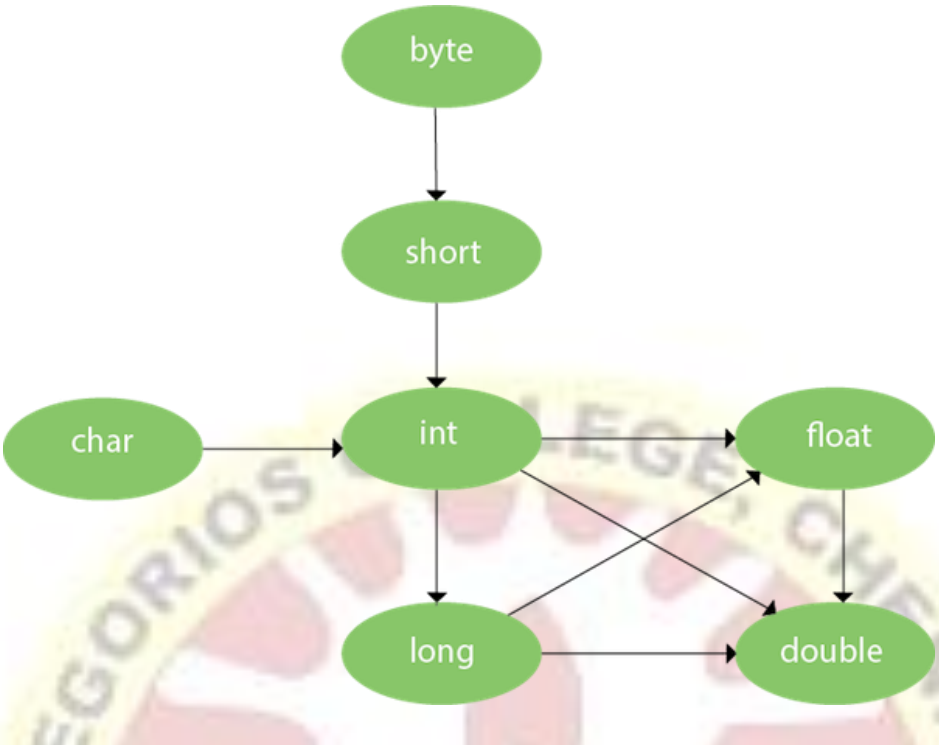
```
1.      class TestOverloading4{
2.      public static void main(String[] args){System.out.println("main with Str
ing[]");}
3.      public static void main(String args){System.out.println("main with Strin
g");}
4.      public static void main(){System.out.println("main without args");}
5.      }
```

Output:

```
main with String[]
```

Method Overloading and Type Promotion

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



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

Example of Method Overloading with TypePromotion

```
1.  class OverloadingCalculation1{
2.    void sum(int a,long b){System.out.println(a+b);}
3.    void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation1 obj=new OverloadingCalculation1();
7.    obj.sum(20,20);//now second int literal will be promoted to long
8.    obj.sum(20,20,20);
9.
10.   }
11.  }
```

Output:40

60

Example of Method Overloading with Type Promotion if matching found

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

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

Output:int arg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

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

Output:Compile Time Error

STATIC AND FIXED METHODS:
Static Method in java:

Static methods are methods that do not operate on objects. A static method acts related to a class declare a static method by using the static keyword as a modifier. To declare a method as static is as follows:

```
Static returntype methodname(parameter)
{
//body of the method
}
```

We can call a static method from the outside of a class as follow: classname.staticmethodname(parameters);

A method declared as static has several restrictions:

- They can only call the other static method.
- They must only access static data.
- They cannot refer to this in any other way.

Program:

```
class staticdemo
{
static void print()
{
System.out.println("java programming");
}
public static void main(String args[])
{
print();
}
}
```

Fixed method:

A fixed method is one which cannot be overridden by a subclass. A fixed method can be declared by prefixing the word-final before a method. We can use the keyword final to donate a constant.

Program:

```
class finaldemo
{
public static void main(String arg[])
{
double paperwidth=8.5;
double paperheight=11;
System.out.println("paper size in centimeter:"
+paperwidth*CM_PER_INCH+"by"+paperheight*CM_PER_INCH);
public static final double CM_PER_INCH=2.45;
}
}
```

Java Inner Classes

1. Java Inner classes
2. Advantage of Inner class
3. Difference between nested class and inner class
4. Types of Nested classes

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
1. class Java_Outer_class{
2.  //code
```

```
3.  class Java_Inner_class{
4.   //code
5.  }
6. }
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Difference between nested class and inner class in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)

1. Member inner class
2. Anonymous inner class
3. Local inner class

STRINGS IN JAVA

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

- The Java platform provides the String class to create and manipulate strings.

- Creating Strings
- The most direct way to create a string is to write −

- String greeting = "Hello world!";
- Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

- As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

- Example

```
publicclassStringDemo{

publicstaticvoid main(String args[]){
char[] helloArray ={'h','e','l','l','o','.'};
String helloString =newString(helloArray);
```

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

This will produce the following result −

Output

hello.

Note − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

The following program is an example of **length()**, method String class.

Example

```
publicclassStringDemo{

publicstaticvoid main(String args[]){
String palindrome ="Dot saw I was Tod";
int len = palindrome.length();
System.out.println("String Length is : "+ len );
}
}
```

This will produce the following result −

Output

String Length is : 17

Concatenating Strings

The String class includes a method for concatenating two strings −

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

- "My name is ".concat("Zara");
- Strings are more commonly concatenated with the + operator, as in −
- "Hello," + " world" + "!"
- which results in −
- "Hello, world!"
- Let us look at the following example −
- Example

```
publicclassStringDemo{

publicstaticvoid main(String args[]){
String string1 ="saw I was ";
System.out.println("Dot "+ string1 +"Tod");
}
}
```

- This will produce the following result −
- Output

Dot saw I was Tod

- Creating Format Strings
- You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.
- Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of −
- Example

```
System.out.printf("The value of the float variable is "+
"%f, while the value of the integer "+
"variable is %d, and the string "+
"is %s", floatVar, intVar, stringVar);
```

- You can write –

Sr.No.	Method & Description

```
String fs;
fs =String.format("The value of the float variable is "+
"%f, while the value of the integer "+
"variable is %d, and the string "+
"is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

1	**char charAt(int index)** **Returns the character at the specified index.**
2	**int compareTo(Object o)** **Compares this String to another Object.**
3	**int compareTo(String anotherString)** **Compares two strings lexicographically.**
4	**int compareToIgnoreCase(String str)** **Compares two strings lexicographically, ignoring case differences.**
5	**String concat(String str)** **Concatenates the specified string to the end of this string.**
6	**boolean contentEquals(StringBuffer sb)** **Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.**
7	**static String copyValueOf(char[] data)** **Returns a String that represents the character sequence in the array specified.**
8	**static String copyValueOf(char[] data, int offset, int count)** **Returns a String that represents the character sequence in the array specified.**
9	**boolean endsWith(String suffix)** **Tests if this string ends with the specified suffix.**
10	**boolean equals(Object anObject)** **Compares this string to the specified object.**

11	**boolean equalsIgnoreCase(String anotherString)** **Compares this String to another String, ignoring case considerations.**
12	**byte[] getBytes()** **Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.**
13	**byte[] getBytes(String charsetName)** **Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.**
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** **Copies characters from this string into the destination character array.**
15	**int hashCode()** **Returns a hash code for this string.**
16	**int indexOf(int ch)** **Returns the index within this string of the first occurrence of the specified character.**
17	**int indexOf(int ch, int fromIndex)** **Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.**
18	**int indexOf(String str)** **Returns the index within this string of the first occurrence of the specified substring.**
19	**int indexOf(String str, int fromIndex)** **Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.**

20	**String intern()** **Returns a canonical representation for the string object.**
21	**int lastIndexOf(int ch)** **Returns the index within this string of the last occurrence of the specified character.**
22	**int lastIndexOf(int ch, int fromIndex)** **Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.**
23	**int lastIndexOf(String str)** **Returns the index within this string of the rightmost occurrence of the specified substring.**
24	**int lastIndexOf(String str, int fromIndex)** **Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.**
25	**int length()** **Returns the length of this string.**
26	**boolean matches(String regex)** **Tells whether or not this string matches the given regular expression.**
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** **Tests if two string regions are equal.**
28	**boolean regionMatches(int toffset, String other, int ooffset, int len)** **Tests if two string regions are equal.**
29	**String replace(char oldChar, char newChar)** **Returns a new string resulting from**

	replacing all occurrences of oldChar in this string with newChar.
30	**String replaceAll(String regex, String replacement** **Replaces each substring of this string that matches the given regular expression with the given replacement.**
31	**String replaceFirst(String regex, String replacement)** **Replaces the first substring of this string that matches the given regular expression with the given replacement.**
32	**String[] split(String regex)** **Splits this string around matches of the given regular expression.**
33	**String[] split(String regex, int limit)** **Splits this string around matches of the given regular expression.**
34	**boolean startsWith(String prefix)** **Tests if this string starts with the specified prefix.**
35	**boolean startsWith(String prefix, int toffset)** **Tests if this string starts with the specified prefix beginning a specified index.**
36	**CharSequence subSequence(int beginIndex, int endIndex)** **Returns a new character sequence that is a subsequence of this sequence.**
37	**String substring(int beginIndex)** **Returns a new string that is a substring of**

	this string.
38	**String substring(int beginIndex, int endIndex)** **Returns a new string that is a substring of this string.**
39	**char[] toCharArray()** **Converts this string to a new character array.**
40	**String toLowerCase()** **Converts all of the characters in this String to lower case using the rules of the default locale.**
41	**String toLowerCase(Locale locale)** **Converts all of the characters in this String to lower case using the rules of the given Locale.**
42	**String toString()** **This object (which is already a string!) is itself returned.**
43	**String toUpperCase()** **Converts all of the characters in this String to upper case using the rules of the default locale.**
44	**String toUpperCase(Locale locale)** **Converts all of the characters in this String to upper case using the rules of the given Locale.**

45	**String trim()** **Returns a copy of the string, with leading and trailing whitespace omitted.**
46	**static String valueOf(primitive data type x)** **Returns the string representation of the passed data type argument.**

Method Overriding in Java

1. Understanding the problem without method overriding
2. Can we override the static method
3. Method overloading vs. method overriding

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

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

Usage of Java Method Overriding

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

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
1.  //Java Program to illustrate the use of Java Method Overriding
2.  //Creating a parent class.
3.  class Vehicle{
4.    //defining a method
5.    void run(){System.out.println("Vehicle is running");}
6.  }
7.  //Creating a child class
8.  class Bike2 extends Vehicle{
9.    //defining the same method as in the parent class
10.   void run(){System.out.println("Bike is running safely");}
11.
12.   public static void main(String args[]){
13.   Bike2 obj = new Bike2();//creating object
14.   obj.run();//calling method
15.    }
16. }
```

Output:

```
Bike is running safely
```

Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

Super Keyword in Java

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

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.

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
1.  class Person{
2.  int id;
3.  String name;
4.  Person(int id,String name){
5.  this.id=id;
6.  this.name=name;
7.  }
8.  }
9.  class Emp extends Person{
10. float salary;
11. Emp(int id,String name,float salary){
12. super(id,name);//reusing parent constructor
13. this.salary=salary;
14. }
15. void display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. class TestSuper5{
18. public static void main(String[] args){
19. Emp e1=new Emp(1,"ankit",45000f);
20. e1.display();
21. }}
```

Output:

```
1 ankit 45000
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

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

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

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

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

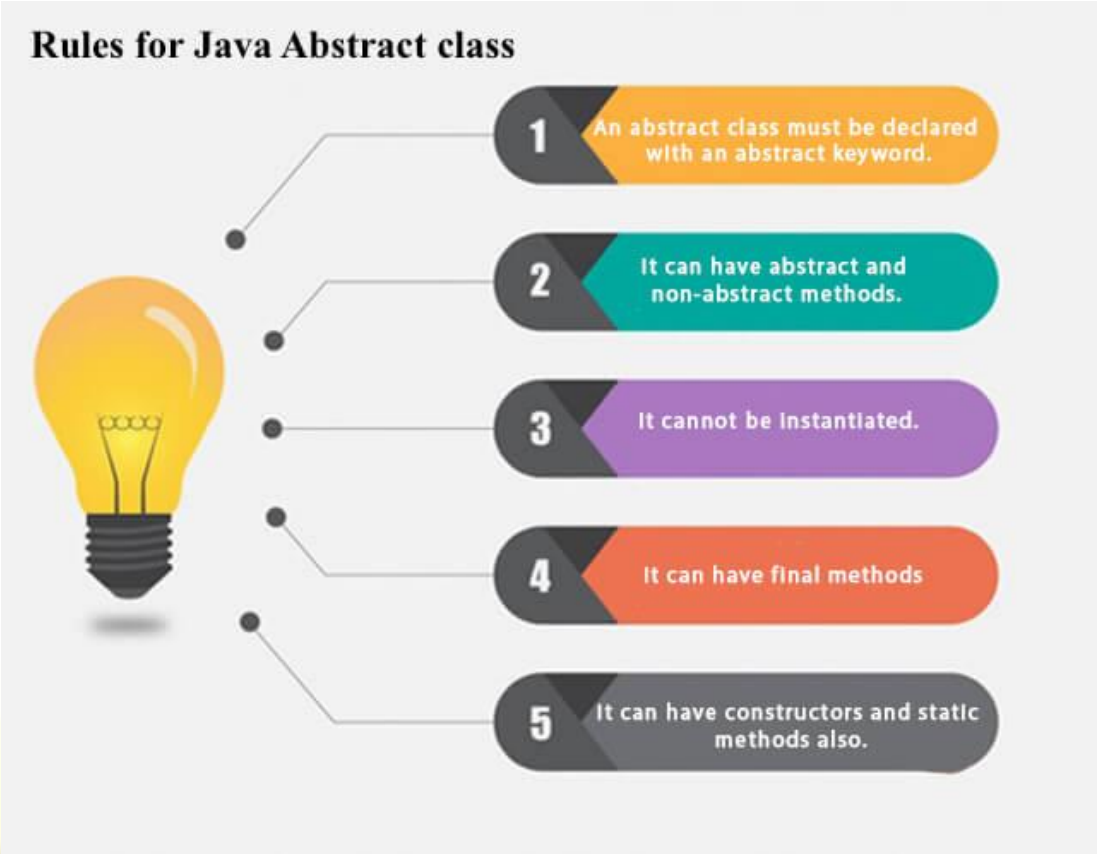
1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

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

Points to Remember

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



Example of abstract class

```
1.  abstract class A{}
```

Abstract Method in Java

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

Example of abstract method

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

Example of Abstract class that has an abstract method

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

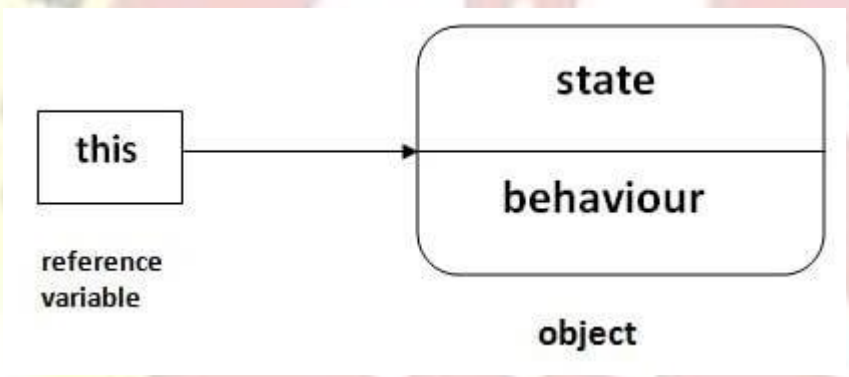
```
1.  abstract class Bike{
2.    abstract void run();
```

```
3.      }
4.      class Honda4 extends Bike{
5.      void run(){System.out.println("running safely");}
6.      public static void main(String args[]){
7.       Bike obj = new Honda4();
8.       obj.run();
9.      }
10.     }
```

running safely

this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.



Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.
7. The **java.lang.Object.finalize()** is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

8. Declaration
9. Following is the declaration for **java.lang.Object.finalize()** method
10. protected void finalize()
11. Parameters
12. NA
13. Return Value
14. This method does not return a value.
15. Exception
16. **Throwable** − the Exception raised by this method
17. Example
18. The following example shows the usage of lang.Object.finalize() method.

```
19. package com.tutorialspoint;
20.
21. import java.util.*;
22.
23. publicclassObjectDemoextendsGregorianCalendar{
24.
25. publicstaticvoid main(String[] args){
26. try{
27. // create a new ObjectDemo object
28. ObjectDemo cal =newObjectDemo();
29.
30. // print current time
31. System.out.println(""+ cal.getTime());
32.
33. // finalize cal
34. System.out.println("Finalizing...");
35.         cal.finalize();
36. System.out.println("Finalized.");
37.
38. }catch(Throwable ex){
39.         ex.printStackTrace();
```

```
40. }
41. }
42. }
```

43. Let us compile and run the above program, this will produce the following result −

```
44. Sat Sep 22 00:27:21 EEST 2012
45. Finalizing...
46. Finalized.
```

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.

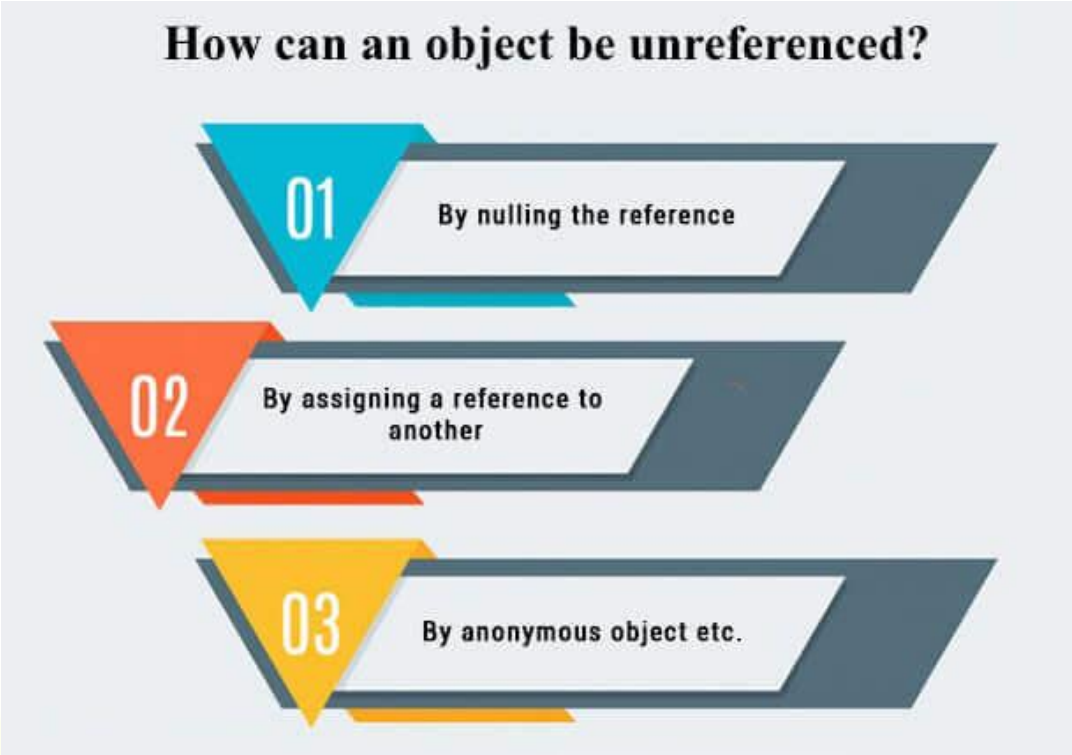
Advantage of Garbage Collection

- 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:

```
1.  Employee e=new Employee();
2.  e=null;
```

2) By assigning a reference to another:

```
1.  Employee e1=new Employee();
2.  Employee e2=new Employee();
3.  e1=e2;//now the first object referred by e1 is available for garbage collect
ion
```

3) By anonymous object:

```
1.  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:

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

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.

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

Simple Example of garbage collection in java

```
1.  public class TestGarbage1{
2.   public void finalize(){System.out.println("object is garbage collected");
}
3.   public static void main(String args[]){
4.    TestGarbage1 s1=new TestGarbage1();
5.    TestGarbage1 s2=new TestGarbage1();
6.    s1=null;
7.    s2=null;
8.    System.gc();
9.   }
10.  }
```

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

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.

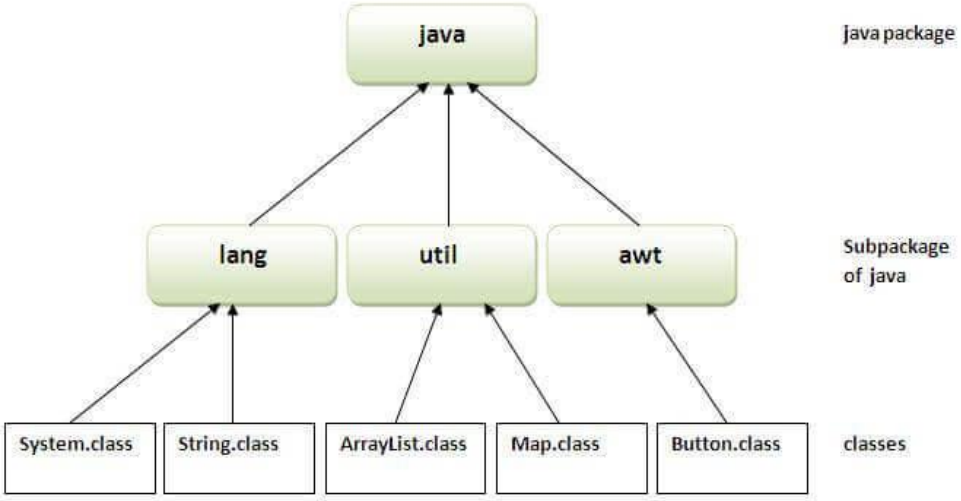
Here, we will have the detailed learning of creating and using user-defined packages.

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.

```
1.  //save as Simple.java
2.  package mypack;
3.  public class Simple{
4.   public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7.  }
```

How to compile java package

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

```
1.  javac -d directory javafilename
```

For **example**

```
1.  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 . r

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.*

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

```
8.          obj.msg();
9.          }
10.       }
```

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

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

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.

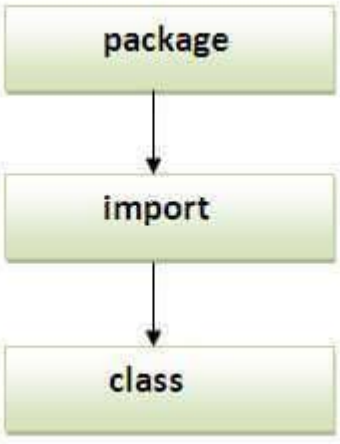
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

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

Output:Hello

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.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Example of Subpackage

```
1.  package com.javatpoint.core;
2.  class Simple{
3.   public static void main(String args[]){
4.    System.out.println("Hello subpackage");
5.   }
6.  }
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are −

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example

Variables and methods can be declared without any modifiers, as in the following examples −

```
String version ="1.5.1";

boolean processOrder(){
returntrue;
}
```

Private Access Modifier - Private

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example

The following class uses private access control −

```
publicclassLogger{
privateString format;

publicString getFormat(){
returnthis.format;
}

publicvoid setFormat(String format){
this.format = format;
```

```
}
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

Public Access Modifier - Public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example

The following function uses public access control −

```
publicstaticvoid main(String[] arguments){
// ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier - Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method −

```
classAudioPlayer{
protectedboolean openSpeaker(Speaker sp){
// implementation details
}
}

classStreamingAudioPlayerextendsAudioPlayer{
boolean openSpeaker(Speaker sp){
// implementation details
}
}
```

Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used protected modifier.

Access Control and Inheritance

The following rules for inherited methods are enforced −

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.

Interface in Java

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

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

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

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

It cannot be instantiated just like the abstract class.

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

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

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

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

How to declare an interface?

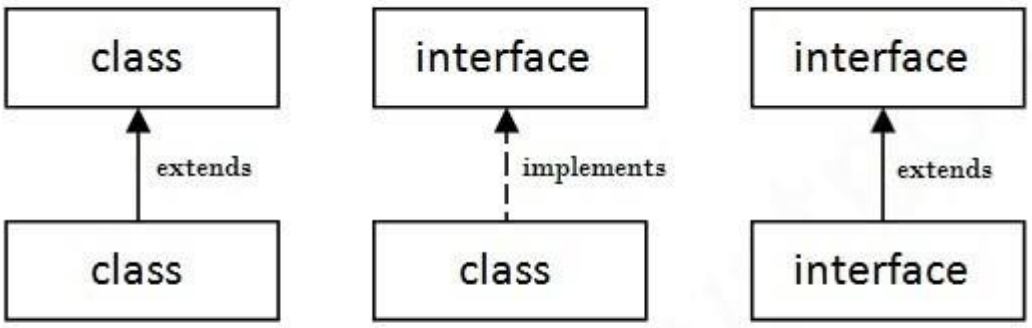
An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the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

Syntax:

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

The relationship between classes and interfaces

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



Java Interface Example

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

.

```
1.  interface printable{
2.  void print();
3.  }
4.  class A6 implements printable{
5.  public void print(){System.out.println("Hello");}
6.
7.  public static void main(String args[]){
8.  A6 obj = new A6();
9.  obj.print();
10.  }
11.  }
```

Output:

```
Hello
```

Java Interface Example: Drawable

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

File: TestInterface1.java

```
1.  //Interface declaration: by first user
2.  interface Drawable{
3.  void draw();
4.  }
5.  //Implementation: by second user
6.  class Rectangle implements Drawable{
7.  public void draw(){System.out.println("drawing rectangle");}
8.  }
9.  class Circle implements Drawable{
10.  public void draw(){System.out.println("drawing circle");}
11.  }
12.  //Using interface: by third user
13.  class TestInterface1{
```

```
14.  public static void main(String args[]){
15.  Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
16.  d.draw();
17.  }}
```

Output:

```
drawing circle
```

Java Interface Example: Bank

Example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

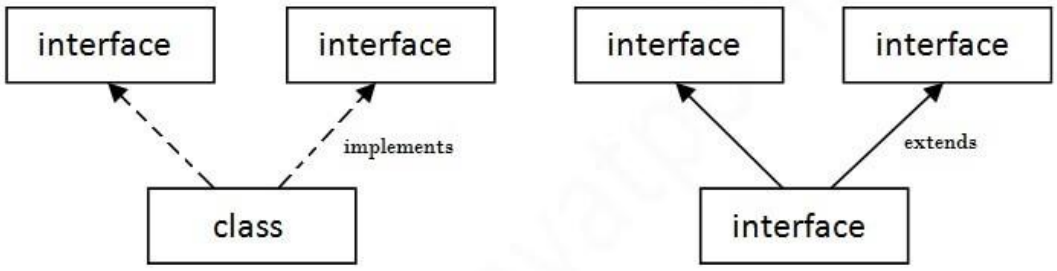
```
1.  interface Bank{
2.  float rateOfInterest();
3.  }
4.  class SBI implements Bank{
5.  public float rateOfInterest(){return 9.15f;}
6.  }
7.  class PNB implements Bank{
8.  public float rateOfInterest(){return 9.7f;}
9.  }
10.  class TestInterface2{
11.  public static void main(String[] args){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

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

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

Output

Hello

Welcome

Multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable{
5.  void print();
6.  }
7.
8.  class TestInterface3 implements Printable, Showable{
9.  public void print(){System.out.println("Hello");}
10. public static void main(String args[]){
11. TestInterface3 obj = new TestInterface3();
12. obj.print();
13.  }
14. }
```

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1.

Interface inheritance

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

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

```
13.     obj.print();
14.     obj.show();
15.      }
16.     }
```

Output:

```
Hello
Welcome
```

Default Method in Interface

Let's see an example:

File: TestInterfaceDefault.java

```
1.     interface Drawable{
2.     void draw();
3.     default void msg(){System.out.println("default method");}
4.     }
5.     class Rectangle implements Drawable{
6.     public void draw(){System.out.println("drawing rectangle");}
7.     }
8.     class TestInterfaceDefault{
9.     public static void main(String args[]){
10.    Drawable d=new Rectangle();
11.    d.draw();
12.    d.msg();
13.    }}
```

Output:

```
drawing rectangle
default method
```

Java 8 Static Method in Interface

we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
1.  interface Drawable{
2.  void draw();
3.  static int cube(int x){return x*x*x;}
4.  }
5.  class Rectangle implements Drawable{
6.  public void draw(){System.out.println("drawing rectangle");}
7.  }
8.
9.  class TestInterfaceStatic{
10. public static void main(String args[]){
11. Drawable d=new Rectangle();
12. d.draw();
13. System.out.println(Drawable.cube(3));
14. }}
```

Output:

```
drawing rectangle
27
```

Exception Handling in Java

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

In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions.

What is Exception in Java

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

What is Exception Handling

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

Advantage of Exception Handling

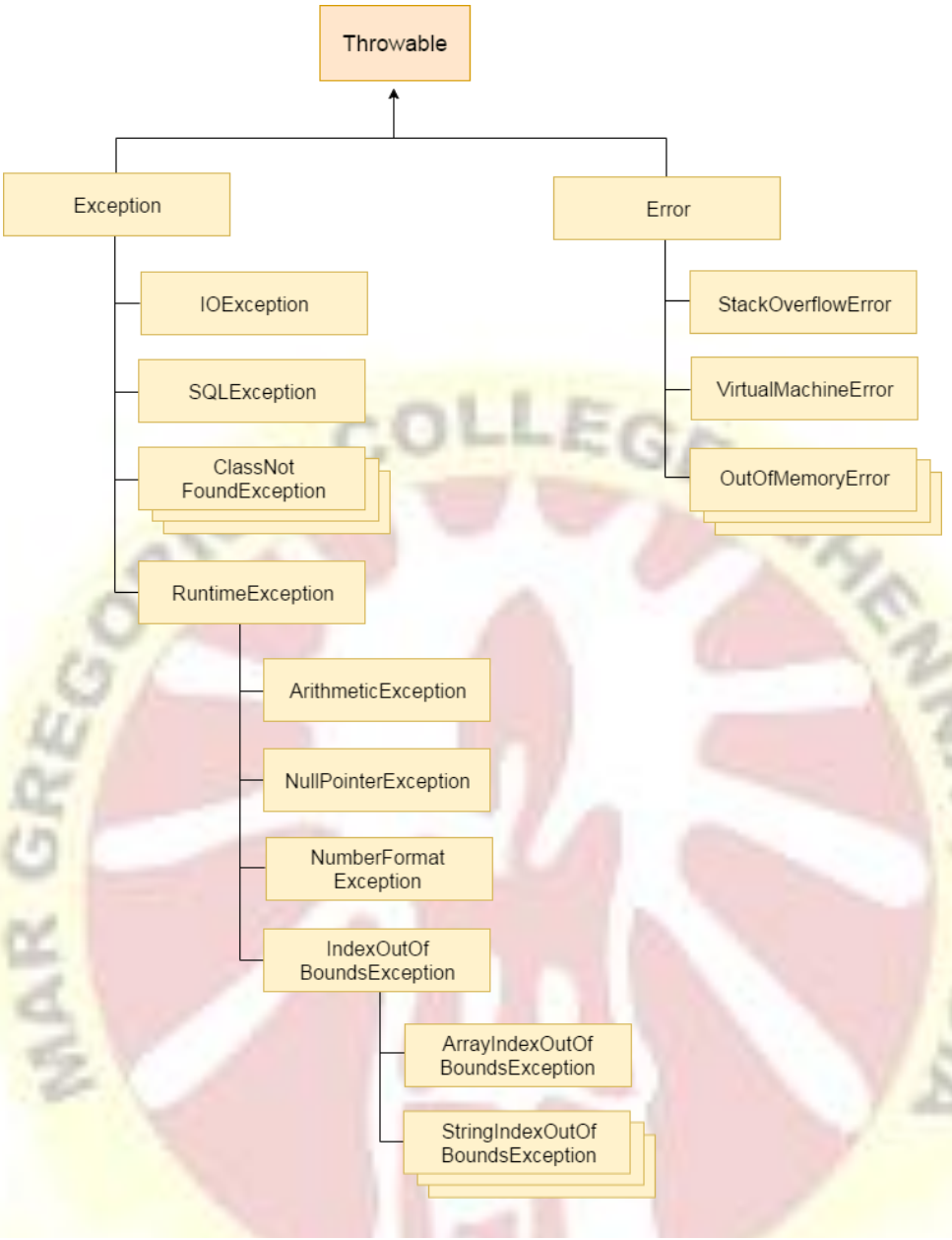
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

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

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement 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 which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw

	an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

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

```
1.  public class JavaExceptionExample{
2.    public static void main(String args[]){
3.    try{

4.      //code that may raise exception
5.      int data=100/0;
6.    }catch(ArithmeticException e){System.out.println(e);}
7.    //rest code of the program
8.    System.out.println("rest of the code...");
9.    }
10.  }
```

Output:

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

Both throw and throws are the concepts of exception handing in which throw is used to explicitly throw an exception from a method or any block of code while throws are used in the signature of the method to indicate that this method might throw one of the listed type exceptions.

The following are the important differences between throw and throws.

Sr. No.	Key	throw	throws
1	Definition	Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code.	Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code.
2	Internal implementation	Internally throw is implemented as it is allowed to throw only single exception	On other hand we can declare multiple exceptions with throws keyword

		at a time i.e we cannot throw multiple exception with throw keyword.	that could get thrown by the function where throws keyword is used.
3	Type of exception	With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw.	On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws

			keyword followed by specific exceptio n class name.
4	Syntax	Syntax wise throw keyword is followed by the instance variable.	On other hand syntax wise throws keyword is followed by exceptio n class names.
5	Declaration	In order to use throw keyword we should know that throw keyword is used within the	On other hand throws keyword is used with the method signature .

		method.	

THREADS IN JAVA

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

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

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

Life Cycle of a Thread

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

Following are the stages of the life cycle −

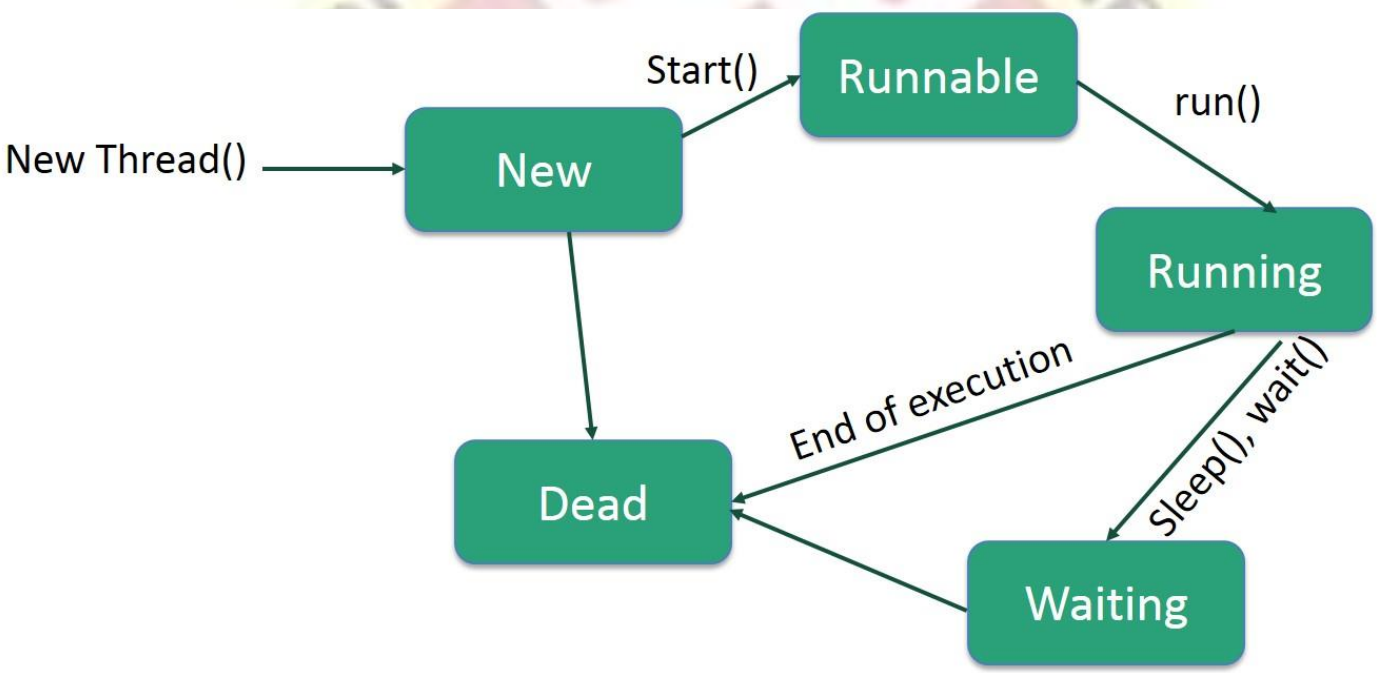
- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

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

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

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

Thread Priorities



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

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

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

Create a Thread by Implementing a Runnable Interface

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

Step 1

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

public void run( )

Step 2

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

Thread(Runnable threadObj, String threadName);

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

Step 3

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

void start();

Example

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

```
classRunnableDemoimplementsRunnable{
privateThread t;
privateString threadName;

RunnableDemo(String name){
    threadName = name;
System.out.println("Creating "+  threadName );
}

publicvoid run(){
```

```
System.out.println("Running "+  threadName );
try{
for(int i =4; i >0; i--){
System.out.println("Thread: "+ threadName +", "+ i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
}catch(InterruptedException e){
System.out.println("Thread "+  threadName +" interrupted.");
}
System.out.println("Thread "+  threadName +" exiting.");
}

publicvoid start (){
System.out.println("Starting "+  threadName );
if(t ==null){
      t =newThread(this, threadName);
      t.start ();
}
}
}

publicclassTestThread{

publicstaticvoid main(String args[]){
RunnableDemo R1 =newRunnableDemo("Thread-1");
    R1.start();

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

This will produce the following result −

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run( )** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −

```
public void run( )
```

Step 2

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

```
void start( );
```

Example

Here is the preceding program rewritten to extend the Thread −

```
classThreadDemoextendsThread{
privateThread t;
privateString threadName;

ThreadDemo(String name){
    threadName = name;
System.out.println("Creating "+  threadName );
}

publicvoid run(){
System.out.println("Running "+  threadName );
try{
for(int i =4; i >0; i--){
System.out.println("Thread: "+ threadName +", "+ i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
}catch(InterruptedException e){
System.out.println("Thread "+  threadName +" interrupted.");
}
System.out.println("Thread "+  threadName +" exiting.");
}

publicvoid start (){
System.out.println("Starting "+  threadName );
if(t ==null){
      t =newThread(this, threadName);
      t.start ();
}
}
}
```

```
publicclassTestThread{

publicstaticvoid main(String args[]){
ThreadDemo T1 =newThreadDemo("Thread-1");
    T1.start();

ThreadDemo T2 =newThreadDemo("Thread-2");
    T2.start();
}
}
```

This will produce the following result −

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread Methods

Following is the list of important methods available in the Thread class.

Sr.No.	Method & Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

Sr.No.	Method & Description
1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds.
3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable** −

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable

publicclassDisplayMessageimplementsRunnable{
privateString message;
```

```
publicDisplayMessage(String message){
this.message = message;
}

publicvoid run(){
while(true){
System.out.println(message);
}
}
}
```

Following is another class which extends the Thread class −

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread

publicclassGuessANumberextendsThread{
privateint number;
publicGuessANumber(int number){
this.number = number;
}

publicvoid run(){
int counter =0;
int guess =0;
do{
      guess =(int)(Math.random()*100+1);
System.out.println(this.getName()+" guesses "+ guess);
      counter++;
}while(guess != number);
System.out.println("** Correct!"+this.getName()+"in"+ counter +"guesses.**");
}
}
```

Following is the main program, which makes use of the above-defined classes −

```
// File Name : ThreadClassDemo.java
publicclassThreadClassDemo{

publicstaticvoid main(String[] args){
Runnable hello =newDisplayMessage("Hello");
Thread thread1 =newThread(hello);
    thread1.setDaemon(true);
    thread1.setName("hello");
System.out.println("Starting hello thread...");
    thread1.start();

Runnable bye =newDisplayMessage("Goodbye");
Thread thread2 =newThread(bye);
    thread2.setPriority(Thread.MIN_PRIORITY);
    thread2.setDaemon(true);
System.out.println("Starting goodbye thread...");
    thread2.start();

System.out.println("Starting thread3...");
Thread thread3 =newGuessANumber(27);
    thread3.start();
try{
       thread3.join();
}catch(InterruptedException e){
System.out.println("Thread interrupted.");
}
System.out.println("Starting thread4...");
Thread thread4 =newGuessANumber(75);

    thread4.start();
System.out.println("main() is ending...");
}
}
```

This will produce the following result. You can try this example again and again and you will get a different result every time.

Output

Starting hello thread...

Starting goodbye thread...

Hello

Hello

Hello

Hello

Hello

Hello

Goodbye

Goodbye

Goodbye

Goodbye

Goodbye

.....

DEADLOCK

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example.

Example

```
publicclassTestThread{
publicstaticObjectLock1=newObject();
publicstaticObjectLock2=newObject();

publicstaticvoid main(String args[]){
ThreadDemo1 T1 =newThreadDemo1();
ThreadDemo2 T2 =newThreadDemo2();
    T1.start();
    T2.start();
```

```
}

privatestaticclassThreadDemo1extendsThread{
publicvoid run(){
synchronized(Lock1){
System.out.println("Thread 1: Holding lock 1...");

try{Thread.sleep(10);}
catch(InterruptedException e){}
System.out.println("Thread 1: Waiting for lock 2...");

synchronized(Lock2){
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}
}
privatestaticclassThreadDemo2extendsThread{
publicvoid run(){
synchronized(Lock2){
System.out.println("Thread 2: Holding lock 2...");

try{Thread.sleep(10);}
catch(InterruptedException e){}
System.out.println("Thread 2: Waiting for lock 1...");

synchronized(Lock1){
System.out.println("Thread 2: Holding lock 1 & 2...");
}
}
}
}
}
```

When you compile and execute the above program, you find a deadlock situation and following is the output produced by the program −

Output

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock,

APPLETS

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.

- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;


publicclassHelloWorldAppletextendsApplet{
publicvoid paint (Graphics g){
    g.drawString ("Hello World",25,50);
}
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet −

```
<html>
<title>The Hello, World Applet</title>
<hr>
<appletcode="HelloWorldApplet.class"width="320"height="120">
    If your browser was Java-enabled, a "Hello, World"
    message would appear here.
</applet>
<hr>
</html>
```

Note − You can refer to HTML Applet Tag to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown −

```
<applet codebase = "https://amrood.com/applets" code = "HelloWorldApplet.class"
  width = "320" height = "120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example −

```
<applet  = "mypackage.subpackage.TestApplet.class"
  width = "320" height = "120">
```

Getting Applet Parameters

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.

The second color and the size of each square may be specified as parameters to the applet within the document.

CheckerApplet gets its parameters in the init() method. It may also get its parameters in the paint() method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.

The applet viewer or browser calls the init() method of each applet it runs. The viewer calls init() once, immediately after loading the applet. (Applet.init() is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The Applet.getParameter() method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of CheckerApplet.java −

```
import java.applet.*;
import java.awt.*;

publicclassCheckerAppletextendsApplet{
int squareSize =50;// initialized to default size
publicvoid init(){}
privatevoid parseSquareSize (String param){}
privateColor parseColor (String param){}
publicvoid paint (Graphics g){}
}
```

Here are CheckerApplet's init() and private parseSquareSize() methods −

```
publicvoid init (){
String squareSizeParam = getParameter ("squareSize");
  parseSquareSize (squareSizeParam);

String colorParam = getParameter ("color");
```

```
Color fg = parseColor (colorParam);

  setBackground (Color.black);
  setForeground (fg);
}

privatevoid parseSquareSize (String param){
if(param ==null)return;
try{
    squareSize =Integer.parseInt (param);
}catch(Exception e){
// Let default value remain
}
}
```

The applet calls parseSquareSize() to parse the squareSize parameter. parseSquareSize() calls the library method Integer.parseInt(), which parses a string and returns an integer. Integer.parseInt() throws an exception whenever its argument is invalid.

Therefore, parseSquareSize() catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls parseColor() to parse the color parameter into a Color value. parseColor() does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet work.

Specifying Applet Parameters

The following is an example of an HTML file with a CheckerApplet embedded in it. The HTML file specifies both parameters to the applet by means of the <param> tag.

```
<html>
<title>Checkerboard Applet</title>
<hr>
<appletcode="CheckerApplet.class"width="480"height="320">
<paramname="color"value="blue">
<paramname="squaresize"value="30">
</applet>
```

```
<hr>
</html>
```

Note − Parameter names are not case sensitive.

Application Conversion to Applets

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the Java program launcher) into an applet that you can embed in a web page.

Following are the specific steps for converting an application to an applet.

- Make an HTML page with the appropriate tag to load the applet code.
- Supply a subclass of the JApplet class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the main method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
- Move any initialization code from the frame window constructor to the init method of the applet. You don't need to explicitly construct the applet object. The browser instantiates it for you and calls the init method.
- Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
- Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
- If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
- Don't call setVisible(true). The applet is displayed automatically.

Event Handling

Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.

In order to react to an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

publicclassExampleEventHandlingextendsAppletimplementsMouseListener{
```

```
StringBuffer strBuffer;

publicvoid init(){
    addMouseListener(this);
    strBuffer =newStringBuffer();
    addItem("initializing the apple ");
}

publicvoid start(){
    addItem("starting the applet ");
}

publicvoid stop(){
    addItem("stopping the applet ");
}

publicvoid destroy(){
    addItem("unloading the applet");
}

void addItem(String word){
System.out.println(word);
    strBuffer.append(word);
    repaint();
}

publicvoid paint(Graphics g){
// Draw a Rectangle around the applet's display area.
    g.drawRect(0,0,
    getWidth()-1,
    getHeight()-1);

// display the string inside the rectangle.
    g.drawString(strBuffer.toString(),10,20);
```

```
}
publicvoid mouseEntered(MouseEventevent){
}
publicvoid mouseExited(MouseEventevent){
}
publicvoid mousePressed(MouseEventevent){
}
publicvoid mouseReleased(MouseEventevent){
}
publicvoid mouseClicked(MouseEventevent){
    addItem("mouse clicked! ");
}
}
```

Now, let us call this applet as follows −

```
<html>
<title>Event Handling</title>
<hr>
<appletcode="ExampleEventHandling.class"
width="300"height="300">
</applet>
<hr>
</html>
```

Initially, the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle, "mouse clicked" will be displayed as well.

Displaying Images

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the drawImage() method found in the java.awt.Graphics class.

Following is an example illustrating all the steps to show images −

```
import java.applet.*;
import java.awt.*;
import java.net.*;
```

```
publicclassImageDemoextendsApplet{
privateImage image;
privateAppletContext context;

publicvoid init(){
    context =this.getAppletContext();
String imageURL =this.getParameter("image");
if(imageURL ==null){
      imageURL ="java.jpg";
}
try{
      URL url =new URL(this.getDocumentBase(), imageURL);
      image = context.getImage(url);
}catch(MalformedURLException e){
      e.printStackTrace();
// Display in browser status bar
      context.showStatus("Could not load image!");
}
}

publicvoid paint(Graphics g){
    context.showStatus("Displaying image");
    g.drawImage(image,0,0,200,84,null);
    g.drawString("www.javalicense.com",35,100);
}
}
```

Now, let us call this applet as follows −

```
<html>
<title>The ImageDemo applet</title>
<hr>
<appletcode="ImageDemo.class"width="300"height="200">
<paramname="image"value="java.jpg">
</applet>
```

```
<hr>
</html>
```

Playing Audio

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including −

- **public void play()** − Plays the audio clip one time, from the beginning.
- **public void loop()** − Causes the audio clip to replay continually.
- **public void stop()** − Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is an example illustrating all the steps to play an audio −

```
import java.applet.*;
import java.awt.*;
import java.net.*;

publicclassAudioDemoextendsApplet{
privateAudioClip clip;
privateAppletContext context;

publicvoid init(){
    context =this.getAppletContext();
String audioURL =this.getParameter("audio");
if(audioURL ==null){
        audioURL ="default.au";
}
try{
        URL url =new URL(this.getDocumentBase(), audioURL);
        clip = context.getAudioClip(url);
}catch(MalformedURLException e){
        e.printStackTrace();
```

```
        context.showStatus("Could not load audio file!");
    }
    }

    publicvoid start(){
    if(clip !=null){
        clip.loop();
    }
    }

    publicvoid stop(){
    if(clip !=null){
        clip.stop();
    }
    }
    }
```

Now, let us call this applet as follows −

```
<html>
<title>The ImageDemo applet</title>
<hr>
<appletcode="ImageDemo.class"width="0"height="0">
<paramname="audio"value="test.wav">
</applet>
<hr>
</html>
```

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
publicclassCopyFile{

publicstaticvoid main(String args[])throwsIOException{
FileInputStreamin=null;
FileOutputStreamout=null;

try{
in=newFileInputStream("input.txt");
out=newFileOutputStream("output.txt");

int c;
while((c =in.read())!=-1){
out.write(c);
}
}finally{
```

```
if(in!=null){
in.close();
}
if(out!=null){
out.close();
}
}
}
}
```

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
publicclassCopyFile{


publicstaticvoid main(String args[])throwsIOException{
FileReaderin=null;
FileWriterout=null;
```

```
try{
in=newFileReader("input.txt");
out=newFileWriter("output.txt");

int c;
while((c =in.read())!=-1){
out.write(c);
}
}finally{
if(in!=null){
in.close();
}
if(out!=null){
out.close();
}
}
}
```

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.

- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.

- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

Example

```
import java.io.*;
publicclassReadConsole{

publicstaticvoid main(String args[])throwsIOException{
InputStreamReader cin =null;

try{
      cin =newInputStreamReader(System.in);
System.out.println("Enter characters, 'q' to quit.");
char c;
do{
      c =(char) cin.read();
System.out.print(c);
}while(c !='q');
}finally{
if(cin !=null){
      cin.close();
}
}
}
}
```

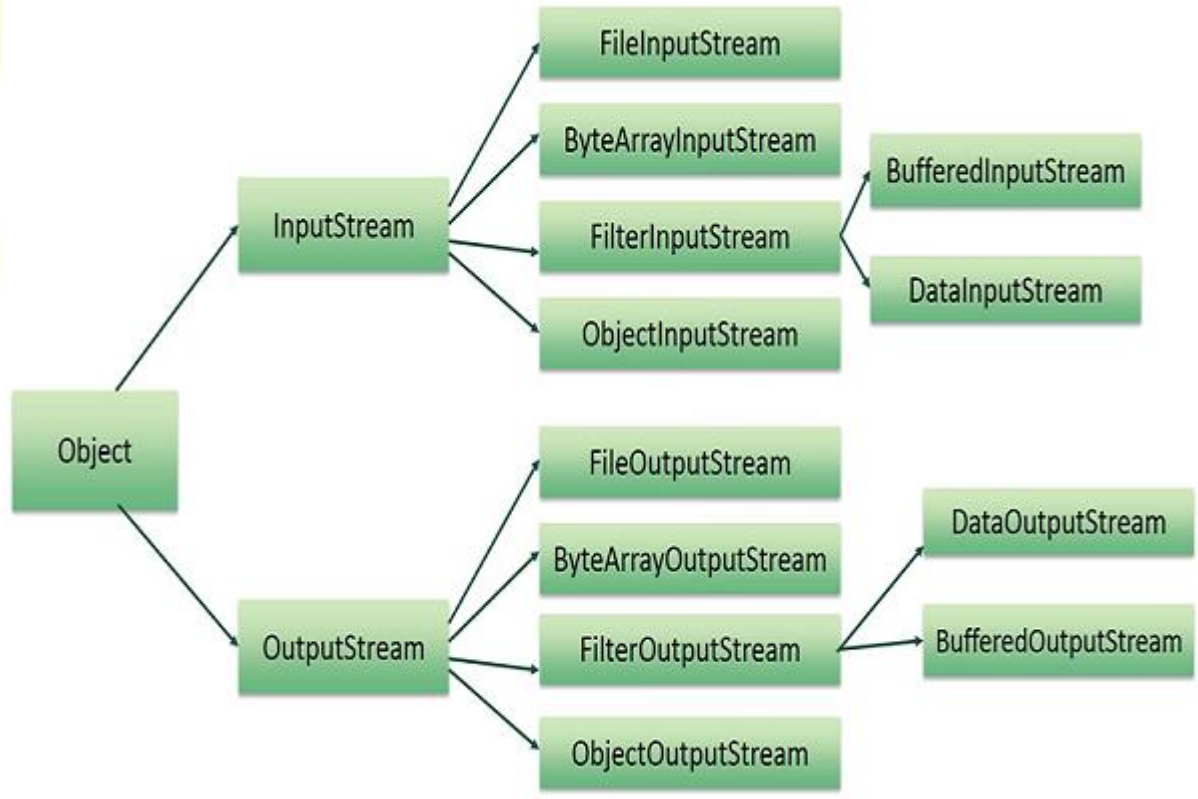
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws**

	IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links −

- **ByteArrayInputStream**
- **DataInputStream**

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream.

	Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links −

- **ByteArrayOutputStream**
- **DataOutputStream**

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;
publicclass fileStreamTest {

publicstaticvoid main(String args[]){

try{
byte bWrite []={11,21,3,40,5};
OutputStream os =newFileOutputStream("test.txt");
for(int x =0; x < bWrite.length ; x++){
```

```
            os.write( bWrite[x]);// writes the bytes
        }
            os.close();

        InputStreamis=newFileInputStream("test.txt");
        int size =is.available();

        for(int i =0; i < size; i++){
        System.out.print((char)is.read()+"  ");
        }
        is.close();
        }catch(IOException e){
        System.out.print("Exception");
        }
        }
        }
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- **File Class**
- **FileReader Class**
- **FileWriter Class**

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

Example

```
import java.io.File;
publicclassCreateDir{

publicstaticvoid main(String args[]){
String dirname ="/tmp/user/java/bin";
File d =newFile(dirname);

// Create directory now.
    d.mkdirs();
}
}
```

Compile and execute the above code to create "/tmp/user/java/bin".

Note − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

Example

```
import java.io.File;
publicclassReadDir{
```

```
publicstaticvoid main(String[] args){
File file =null;
String[] paths;

try{
// create new file object
      file =newFile("/tmp");

// array of files and directory
      paths = file.list();

// for each name in the path array
for(String path:paths){
// prints filename and directory name
System.out.println(path);
}
}catch(Exception e){
// if any error occurs
      e.printStackTrace();
}
}
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory –

Output

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

Unit IV

ABSTRACT DATA TYPE

The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

Let us see some operations of those mentioned ADT −

- Stack −
  - isFull(), This is used to check whether stack is full or not
  - isEmpry(), This is used to check whether stack is empty or not
  - push(x), This is used to push x into the stack
  - pop(), This is used to delete one element from top of the stack
  - peek(), This is used to get the top most element of the stack
  - size(), this function is used to get number of elements present into the stack
- Queue −
  - isFull(), This is used to check whether queue is full or not
  - isEmpry(), This is used to check whether queue is empty or not
  - insert(x), This is used to add x into the queue at the rear end
  - delete(), This is used to delete one element from the front end of the queue
  - size(), this function is used to get number of elements present into the queue
- List −
  - size(), this function is used to get number of elements present into the list

- insert(x), this function is used to insert one element into the list
- remove(x), this function is used to remove given element from the list
- get(i), this function is used to get element at position i
- replace(x, y), this function is used to replace x with y value

ARRAY IMPLEMENTATION OF LISTS LIST:

A list is a sequential data structure, ie. a collection of items accessible one after another beginning at the head and ending at the tail.

It is a widely used data structure for applications which do not need random access Addition and removals can be made at any position in the list

lists are normally in the form of a_1,a_2,a_3.....an. The size of this list is n.The first element of the list is a1,and the last element is an.The position of element ai in a list is i.

List of size 0 is called as null list.

Basic Operations on a List

Creating a list

Traversing the list

Inserting an item in the list

Deleting an item from the list

Concatenating two lists into one

Implementation of List:

A list can be implemented in two ways

1. Array list
2. Linked list

1.Storing a list in a static data structure(Array List)

This implementation stores the list in an array.

The position of each element is given by an index from 0 to n-1, where n is the number of elements.

The element with the index can be accessed in constant time (ie) the time to access does not depend on the size of the list.

The time taken to add an element at the end of the list does not depend on the size of the list. But the time taken to add an element at any other point in the list depends on the size of the list because the subsequent elements must be shifted to next index value.So the additions near the start of the list take longer time than the additions near the middle or end.

Similarly when an element is removed,subsequent elements must be shifted to the previous index value. So removals near the start of the list take longer time than removals near the middle or end of the list.

Problems with Array implementation of lists:

Insertion and deletion are expensive. For example, inserting at position 0 (a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is O(n).

Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, if there are many lists of unknown size.

Simple arrays are generally not used to implement lists. Because the running time for insertion and deletion is so slow and the list size must be known in advance

2. Storing a list in a dynamic data structure(Linked List)

The Linked List is stored as a sequence of linked nodes which are not necessarily adjacent in memory.

Each node in a linked list contains data and a reference to the next node The list can grow and shrink in size during execution of a program.

The list can be made just as long as required.It does not waste memory space because successive elements are connected by pointers.

The position of each element is given by an index from 0 to n-1, where n is the number of elements.

The time taken to access an element with an index depends on the index because each element of the list must be traversed until the required index is found.

The time taken to add an element at any point in the list does not depend on the size of the list,as no shifts are required

Additions and deletion near the end of the list take longer than additions near the middle or start of the list. because the list must be traversed until the required index is found

1. Array versus Linked Lists 1.Arrays are suitable for

- Randomly accessing any element.
- Searching the list for a particular value
- Inserting or deleting an element at the end.

2.Linked lists are suitable for

-Inserting/Deleting an element.

-Applications where sequential access is required.

-In situations where the number of elements cannot be predicted beforehand.

LINKED LIST IMPLEMENTATION OF LISTS(POINTERS)

The Linked List is stored as a sequence of linked nodes which are not necessarily adjacent in memory.

Each node in a linked list contains data and a reference to the next node The list can grow and shrink in size during execution of a program.

In the array implementation,

1. we are constrained to use contiguous space in the memory
2. Insertion, deletion entail shifting the elements

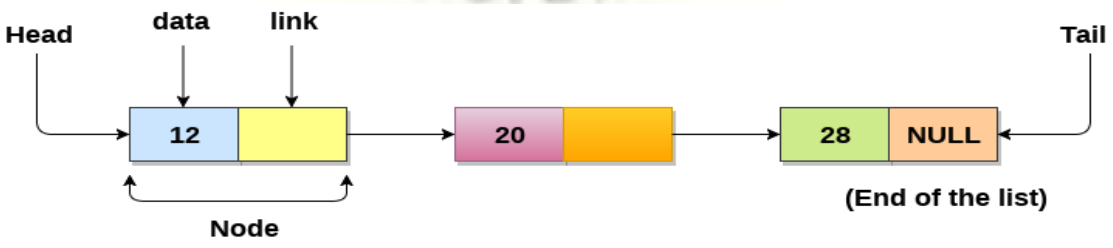
Pointers overcome the above limitations at the cost of extra space for pointers.

data structure that will be used for the nodes. For list where each node holds a float:example, the following struct could be used to create a

```
struct Node
{
int node ;
struct Node *next;
};
```

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer 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 node can reside any where in the memory and linked together to make a list. This achieves 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 the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number 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 node stores 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 other words, we can say that each node contains only next pointer, therefore we can not traverse 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 node in 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.

Complexity

Time Complexity

a p
t a
a c
e

S
t C
r o
u m
c p
t l

u r e									e i t y
	Average				Worst				o r s t
	c c e s s	e a r c h	n s e r t i o n	e l e t i o n	c c e s s	e a r c h	n s e r t i o n	e l e t i o n	
i n g l y L i n k	(n)	(n)	(1)	(1)	(n)	(n)	(1)	(1)	(n)

e d L i s t									

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
1.  struct node
2.  {
3.      int data;
4.      struct node *next;
5.  };
6.  struct node *head, *ptr;
7.  ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

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 end of 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

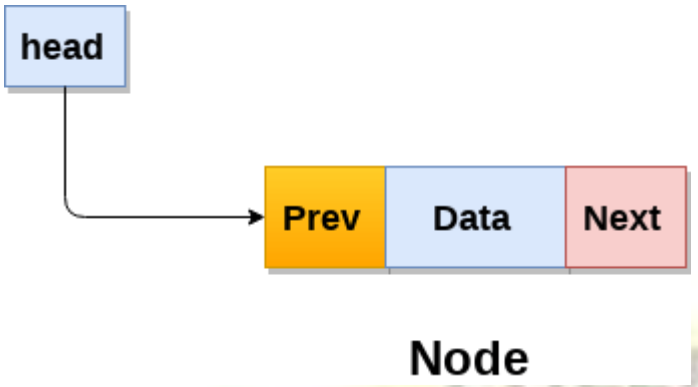
The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

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.

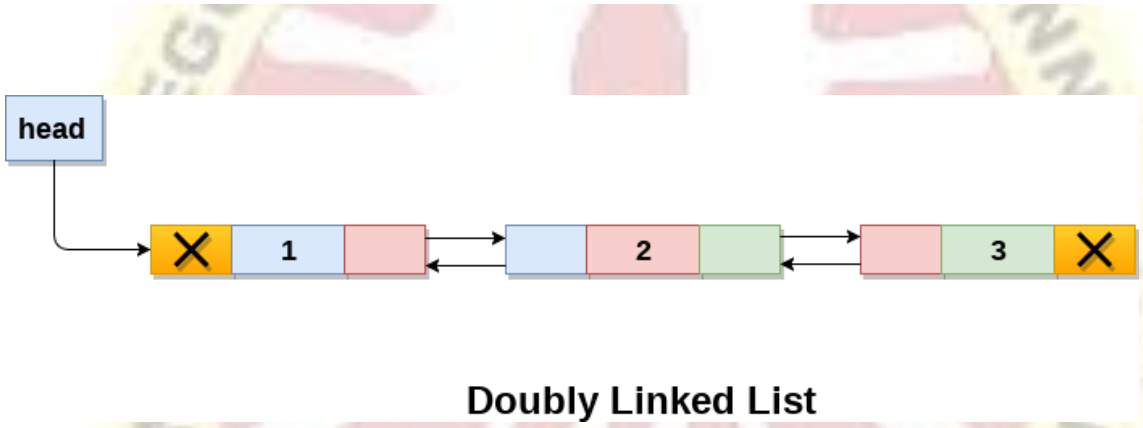
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired 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. .

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

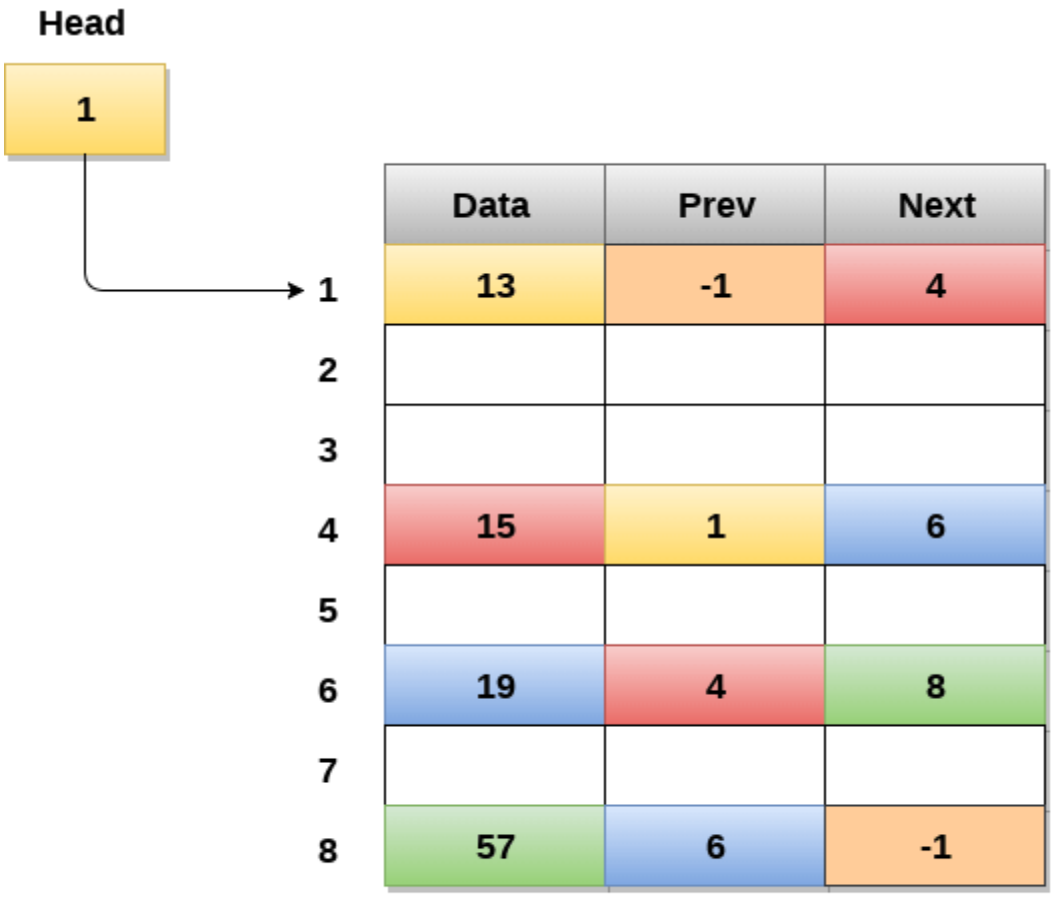
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

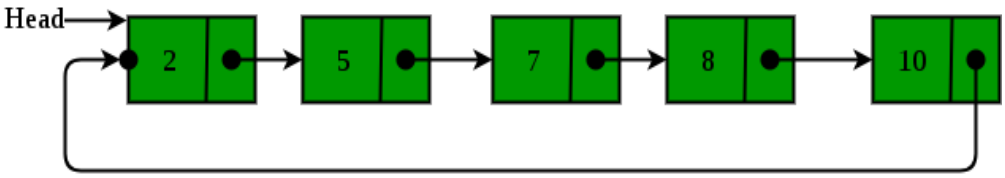
Operations on doubly linked list

Node Creation

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc

Circular linked list *is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*



Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

STACK

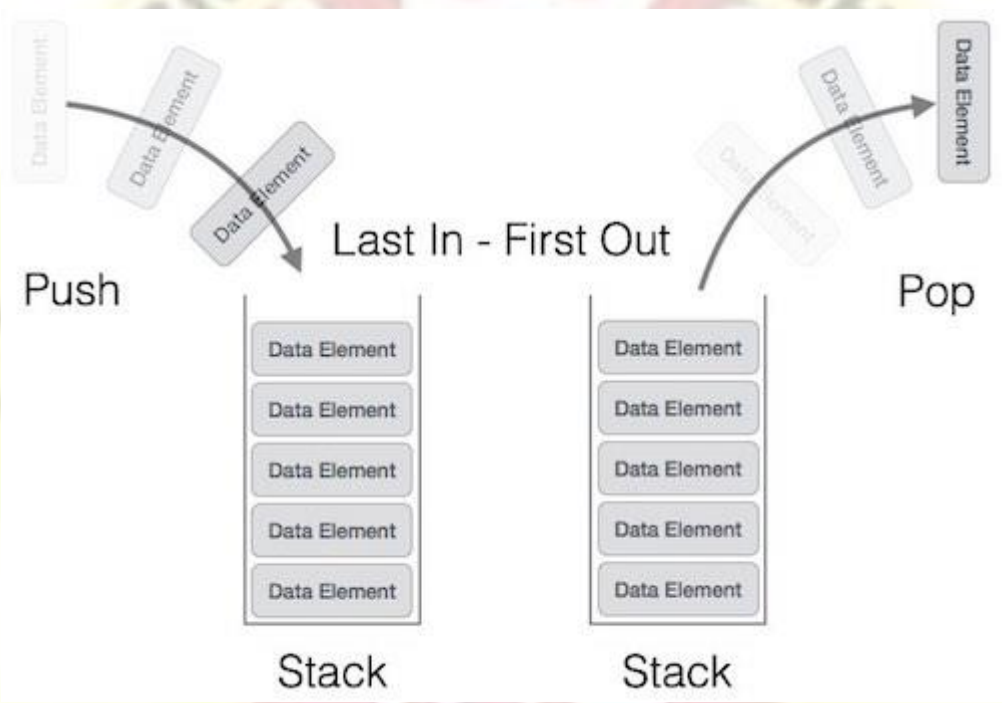
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

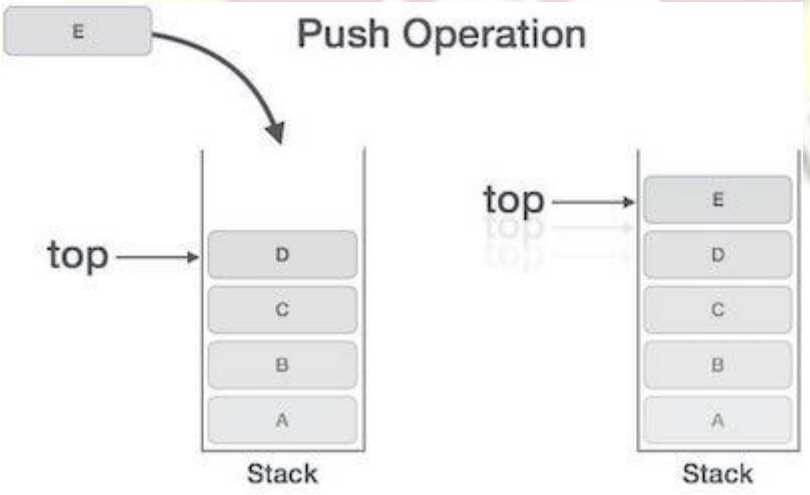
- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



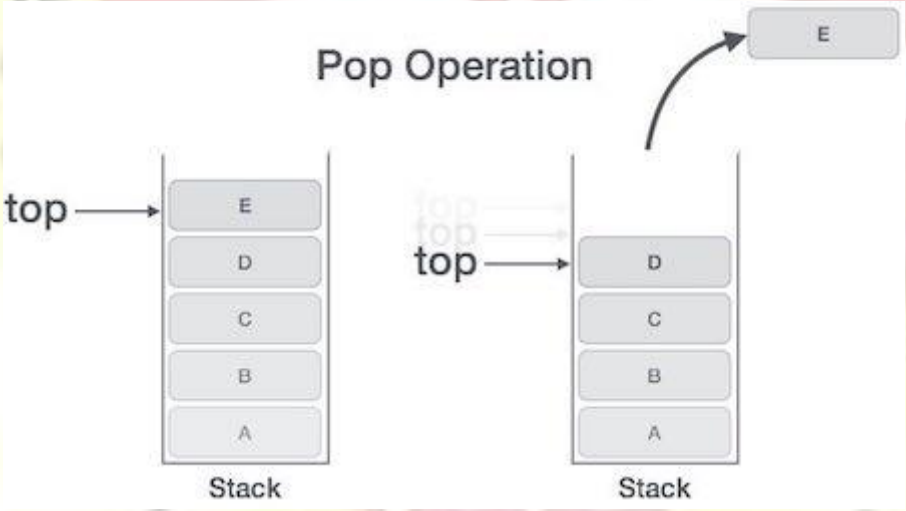
If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Applications of Stack

Following are some of the important applications of a Stack data structure:

1. Stacks can be used for expression evaluation.
2. Stacks can be used to check parenthesis matching in an expression.
3. Stacks can be used for Conversion from one form of expression to another.
4. Stacks can be used for Memory Management.
5. Stack data structures are used in backtracking problems.

Expression Evaluation

Stack data structure is used for evaluating the given expression. For example, consider the following expression

5 * (6 + 2) - 12 / 4

Since parenthesis has the highest precedence among the arithmetic operators, **(6 +2) = 8** will be evaluated first. Now, the expression becomes

5 * 8 - 12 / 4

* and / have equal precedence and their associativity is from left-to-right. So, start evaluating the expression from left-to-right.

5 * 8 = 40 and **12 / 4 = 3**

Now, the expression becomes

40 - 3

And the value returned after the subtraction operation is **37**.

Parenthesis Matching

Given an expression, you have to find if the parenthesis is either correctly matched or not. For example, consider the expression **(a + b) * (c + d).**

In the above expression, the opening and closing of the parenthesis are given properly and hence it is said to be a correctly matched parenthesis expression. Whereas, the expression, **(a + b * [c + d)** is not a valid expression as the parenthesis are incorrectly given.

Click here to know how balanced parenthesis concept is implemented using a Stack Data Structure.

Expression Conversion

Converting one form of expressions to another is one of the important applications of stacks.

- Infix to prefix
- Infix to postfix
- Prefix to Infix

- Prefix to Postfix
- Postfix to Infix
- Postfix to Infix

Infix	Prefix	Postfix
a + b	+ b a	a b +
(a + b) * (c + d)	* + d c + b a	a b + c d + *
b * b - 4 * a * c	- * c * a 4 * b b	b b * 4 a * c * -

Arithmetic Expression Evalution

- Difficulty Level : Easy
- Last Updated : 24 Mar, 2021

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. Polish notation (prefix notation) – It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e., +AB

2. Reverse Polish notation(postfix notation) – It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e., AB+

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below:

Highest: Exponentiation (^)

Next highest: Multiplication (*) and division (/)

Lowest: Addition (+) and Subtraction (-)

For example –

Infix notation: (A-B)*[C/(D+E)+F]

Post-fix notation: AB- CDE +/F +*

Here, we first perform the arithmetic inside the parentheses (A-B) and (D+E). The division of C/(D+E) must be done prior to the addition with F. After that multiply the two terms inside the parentheses and bracket.

Now we need to calculate the value of these arithmetic operations by using a stack.

The procedure for getting the result is:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounters then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression, the final result remains on the top of the stack.

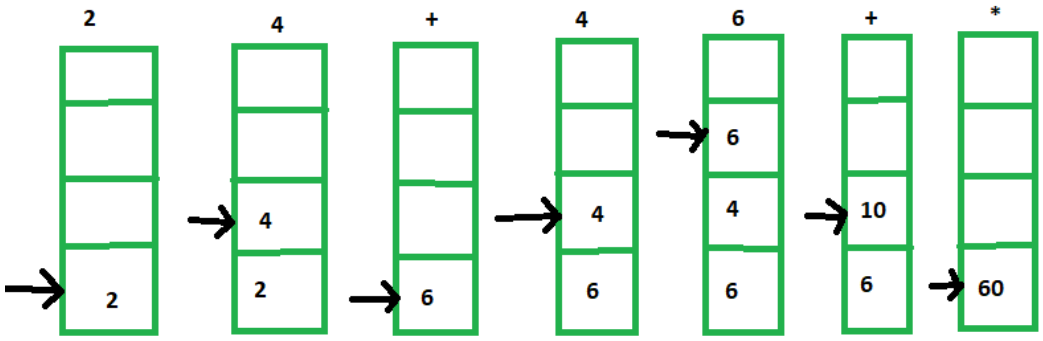
For example –

Infix notation: (2+4) * (4+6)

Post-fix notation: 2 4 + 4 6 + *

Result: 60

The stack operations for this expression evaluation is shown below:



Stack operations to evaluate (2+4)*(4+6)

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.
Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.
Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left. Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +
The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan. The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.
Algorithm
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Output:

abcd^e-fgh*+^*+i-

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.

2. Queue is a FIFO(First in First Out) structure.

3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

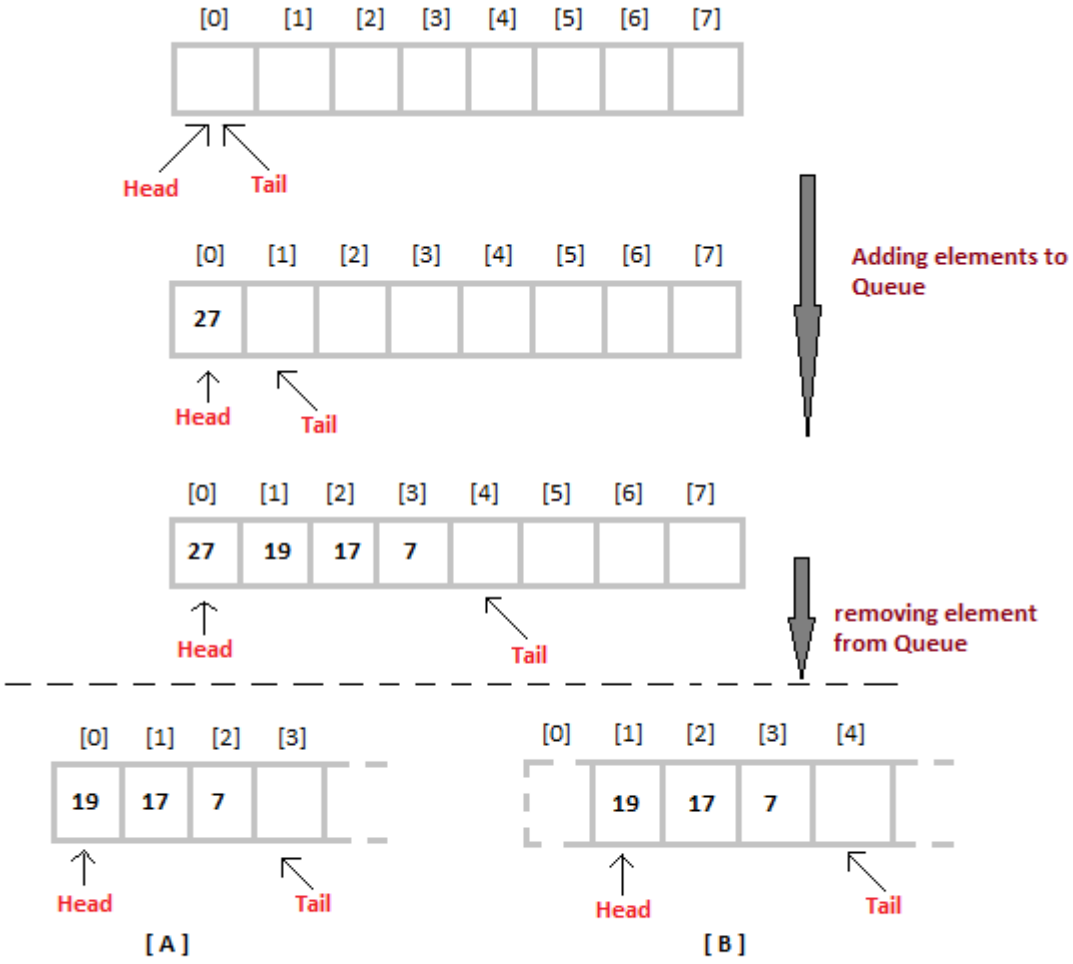
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

Applications of Queue Data Structure

- Difficulty Level : Basic

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

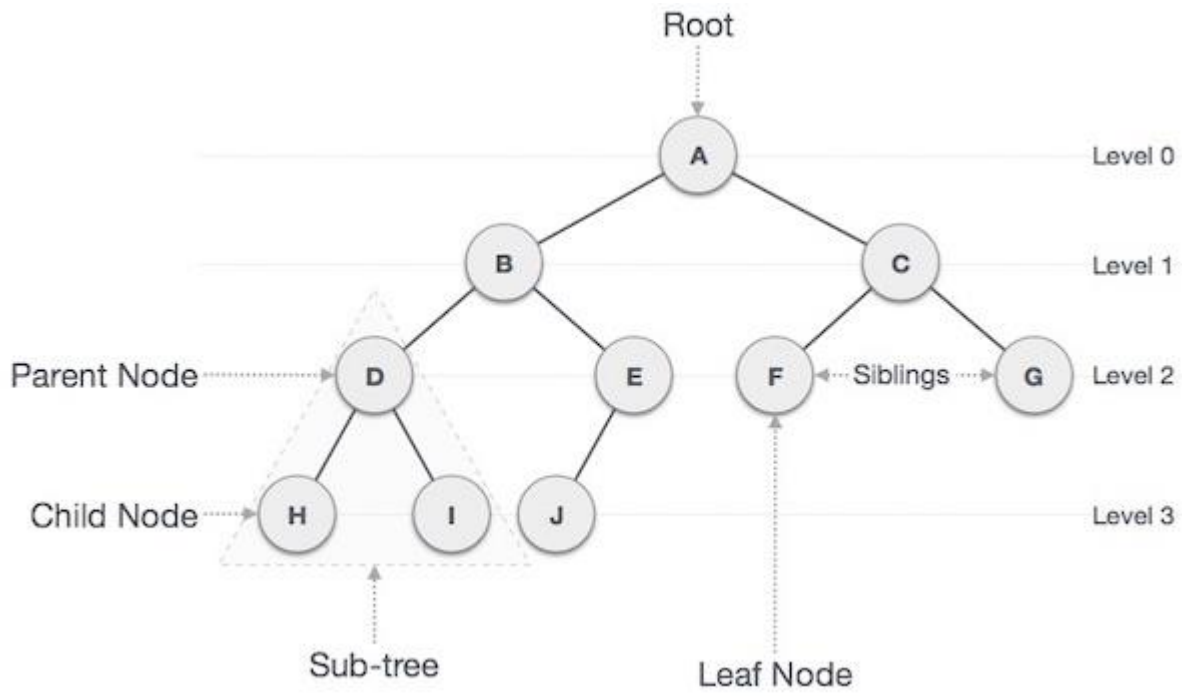
1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

UNIT V

Trees

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

Following are the important terms with respect to tree.

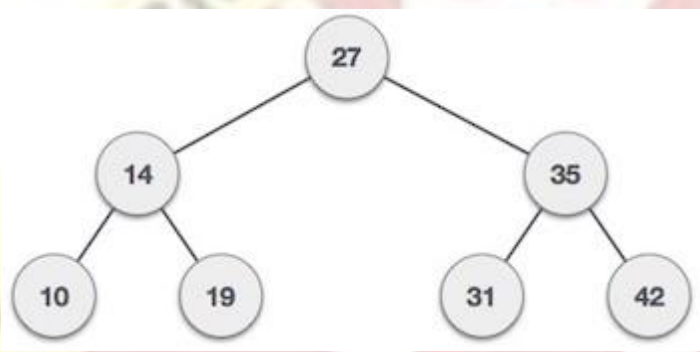
- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.

- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
int data;
struct node *leftChild;
struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.

- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

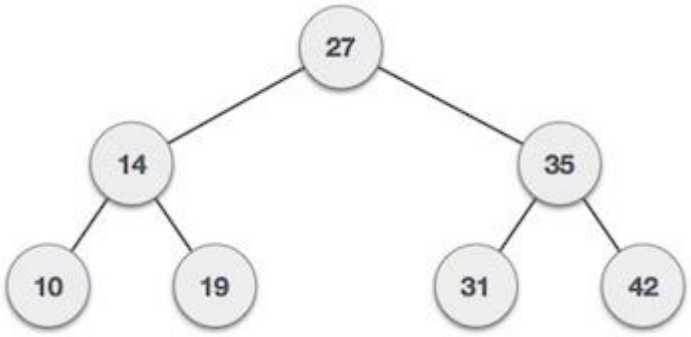
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) < node (key) ≤ right_subtree (keys)

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
int data;
struct node *leftChild;
struct node *rightChild;
};
```

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

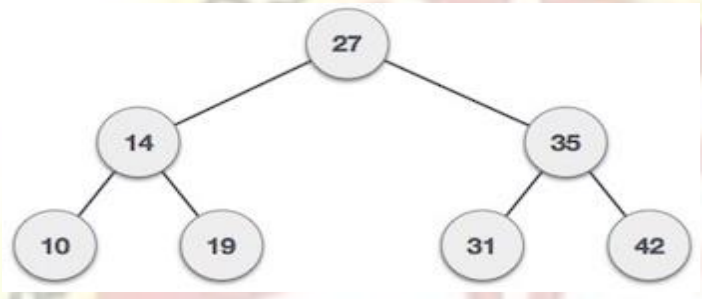
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) < node (key) ≤ right_subtree (keys)

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
int data;
struct node *leftChild;
struct node *rightChild;
};
```

Graph Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

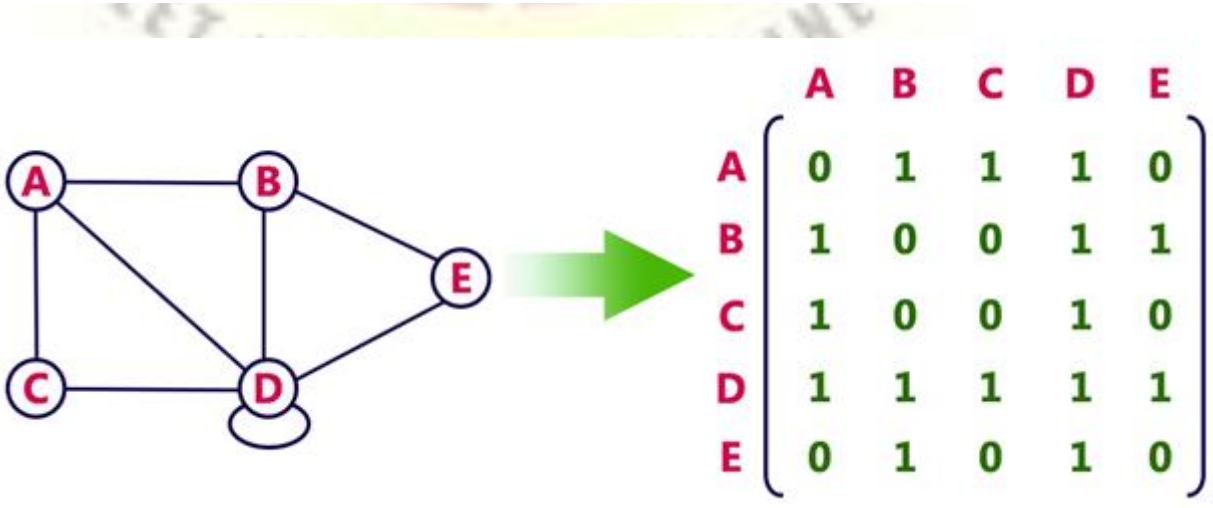
1. Adjacency Matrix

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

Example

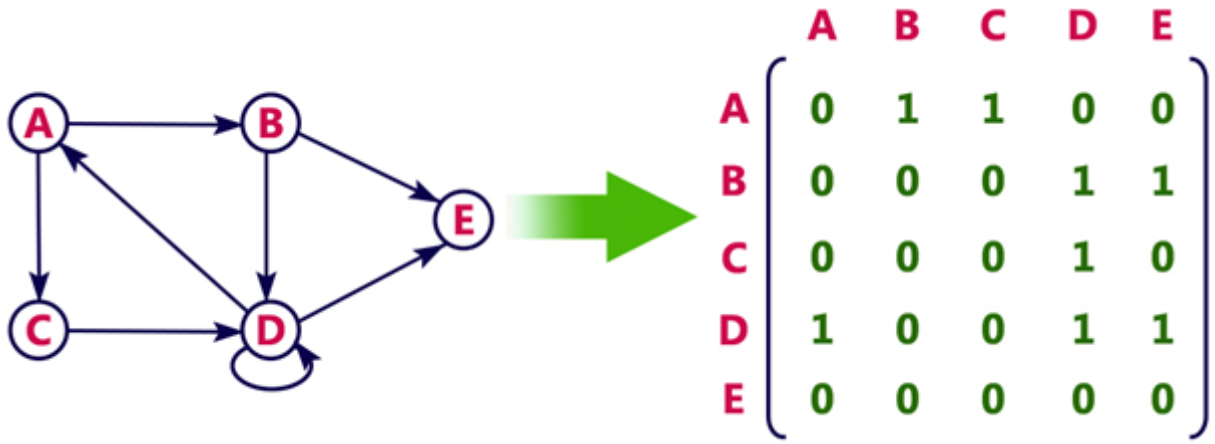
Consider the following **undirected graph representation**:

Undirected graph representation



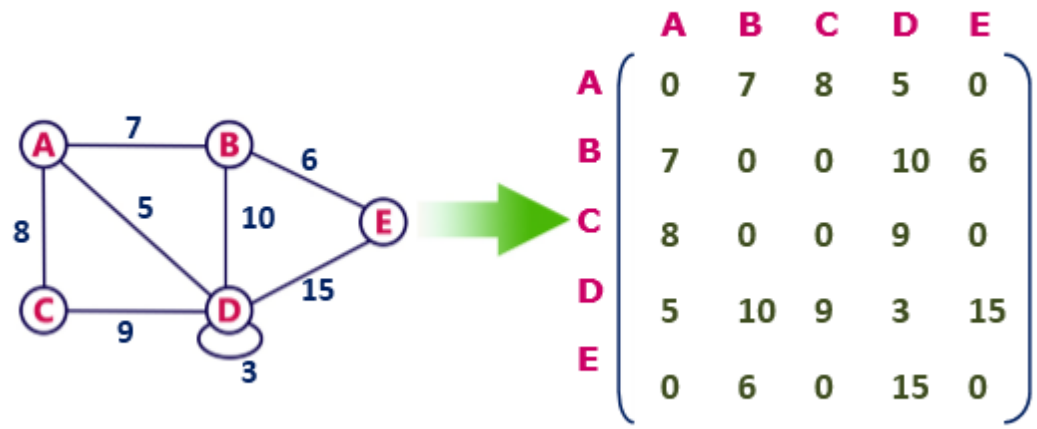
Directed graph represenation

See the directed graph representation:



In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

Undirected weighted graph represenation



Pros: Representation is easier to implement and follow.

Cons: It takes a lot of space and time to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

2. Incidence Matrix

In **Incidence matrix representation**, graph can be represented using a matrix of size:

Total number of vertices by total number of edges.

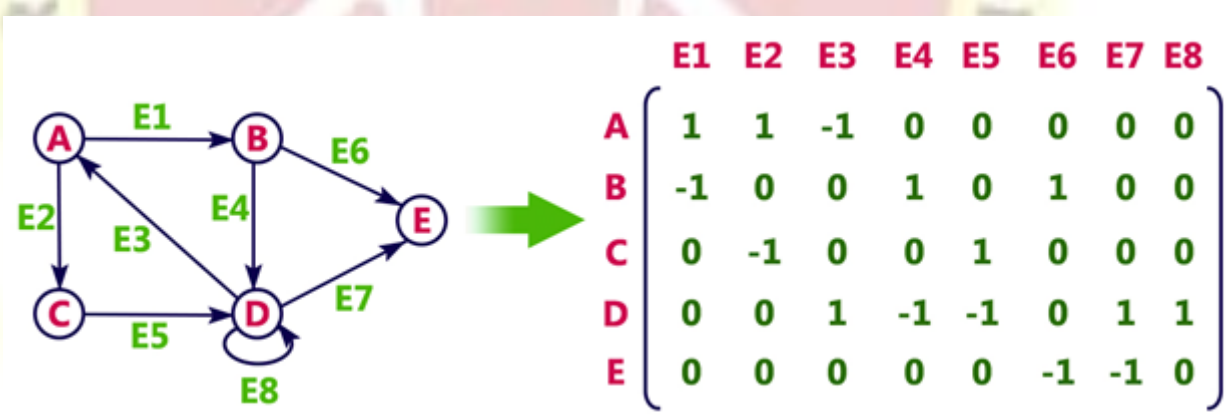
It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.

This matrix is filled with either **0 or 1** or -1. Where,

- 0 is used to represent row edge which is not connected to column vertex.
- 1 is used to represent row edge which is connected as outgoing edge to column vertex.
- -1 is used to represent row edge which is connected as incoming edge to column vertex.

Example

Consider the following directed graph representation.



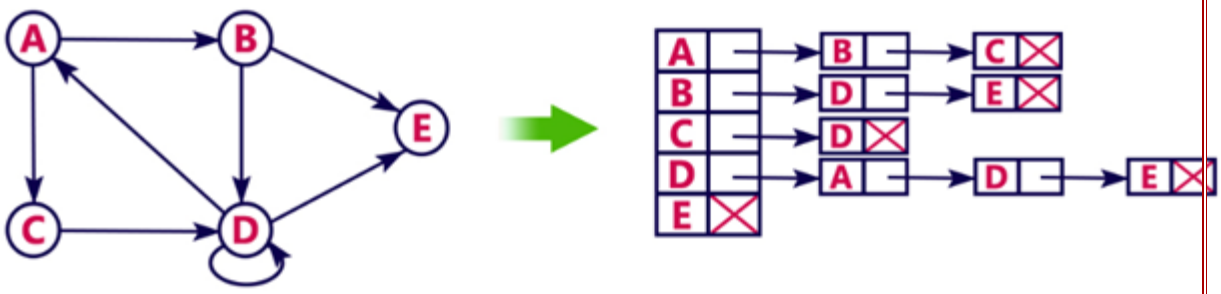
	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

3. **Adjacency List**

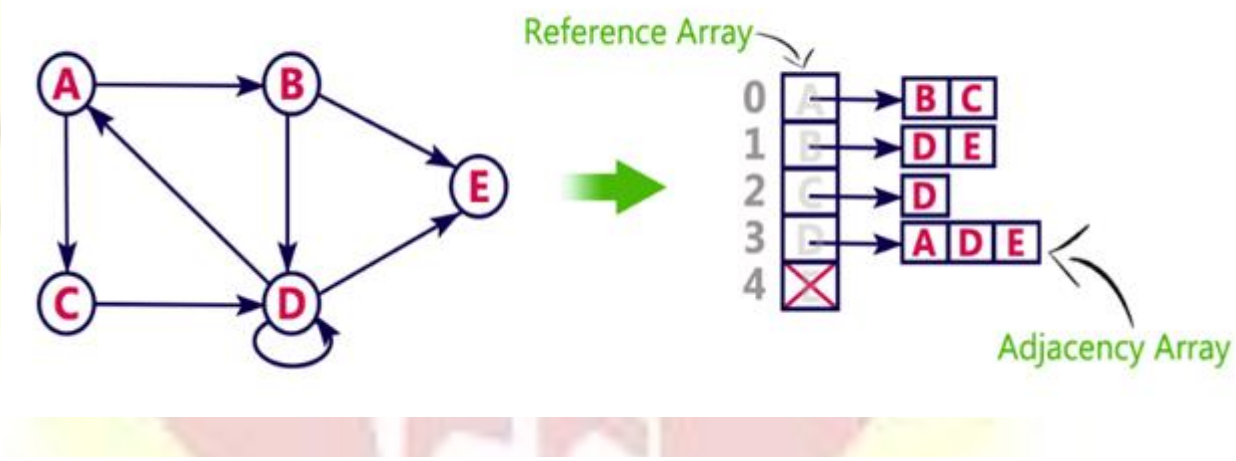
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.

Example

Let's see the following directed graph representation implemented using linked list:



We can also implement this representation using array as follows:



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Cons:

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

****