
Java™

Contents

Day 0

Java is the #1 programming language and development platform. It reduces costs, shortens development timeframes, drives innovation, and improves application services. With millions of developers running more than 51 billion Java Virtual Machines worldwide, Java continues to be the development platform of choice for enterprises and developers.

JAVA is a JVM based programming language

Android apps can be made using JAVA. A significant portion of Android SDK is made using Java.

We also have other JVM based languages.

- **Kotlin JVM based Programming language**

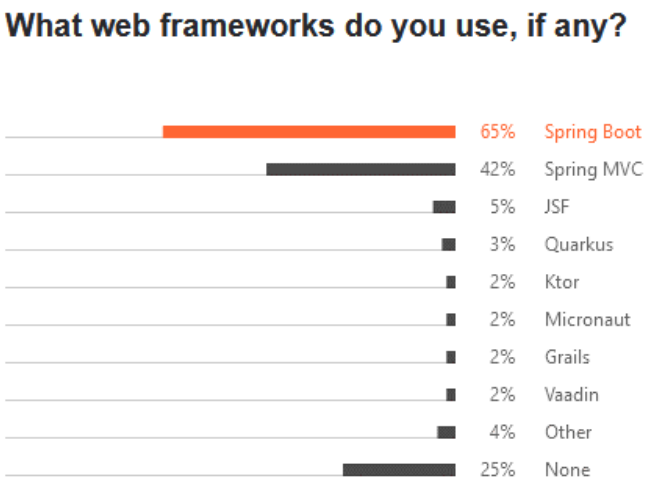
  Kotlin is a cross-platform, statically typed, general-purpose programming language with type inference. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library, but type inference allows its syntax to be more concise.JAVA is an OOP language.]

  Kotlin is also considered a standard language for android development, as Swift is for Mac and iOS.

- **SCALA programming language**

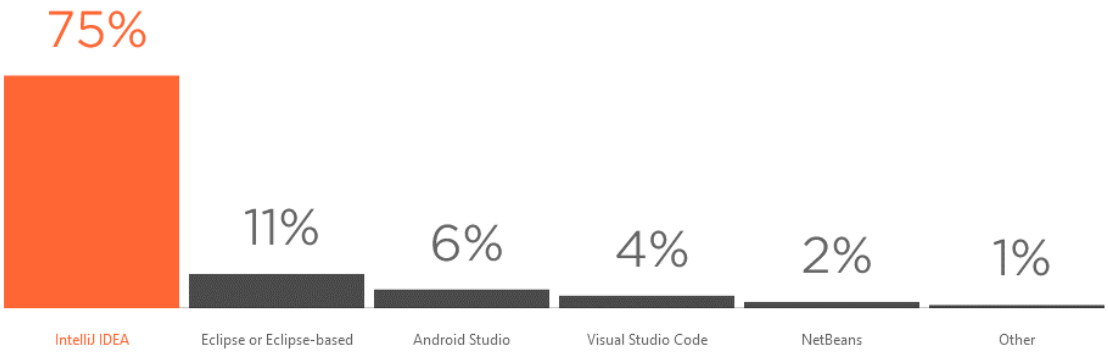
  Scala is a type-safe JVM language that incorporates both object-oriented and functional programming into an extremely concise, logical, and extraordinarily powerful language. Some may be surprised to know that Scala is not quite as new as they thought, having first been introduced in 2003.

Few Popular Frameworks of Java



Popular IDEs

Editors	IDE
• Notepad • Notepad ++ • Visual Code • Sublime Text	• Eclipse • Intelli J • NetBeans



JAVA versioning terminology

Till JDK 8 it was 1.1, 1.2, 1.3 …**1.8**

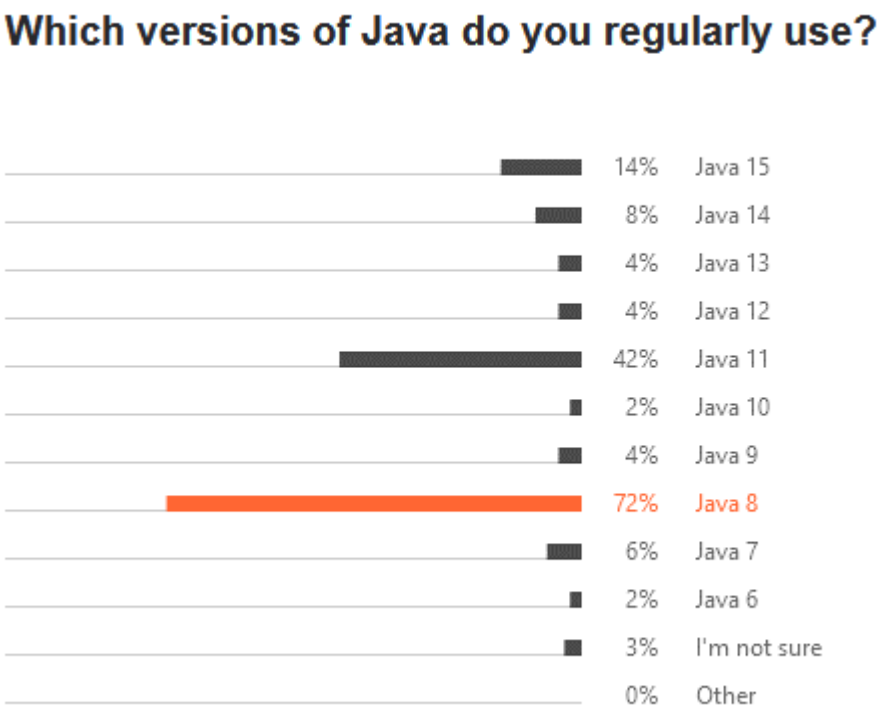
Starting with Java 9 or 10, the 1.X notation also disappeared from the output of `java -version` (which caused some code that depended on parsing it to break), and people have pretty much stopped using it.

We now have Java 15, Java 16, Java 17, JDK 18

1.8 (JDK 8) tops among all the versions it gains the lion's share of about 60 - 80% of JDKs .

It has support from 2015 – to 2030 since it's an LTS(Long Term Support) version.

The latest version of Java is **Java 18** or JDK 18 released on March, 22nd 2022. JDK 18 is a regular update release, and **JDK 17 is the latest Long Term Support (LTS)** release of the Java SE platform



DATA Source:

https://www.jetbrains.com/lp/devecosystem-2021/java/

To Do

SHA JDK 17

873d600f4f26a285f40446dfa612843e2442325cfe0976cefbf0f02299c462b2

LTS releases from JAVA

JDK 8, 11, 17

Our concern should be to master **JDK 8.**

Some popular firms using JAVA

- 90 % of banking firms
- Netflix
- Minecraft
- Zepto

And all the top companies like

- Uber.
- Airbnb.
- Google.
- Pinterest.
- Netflix.
- Instagram.
- Spotify.
- Amazon.

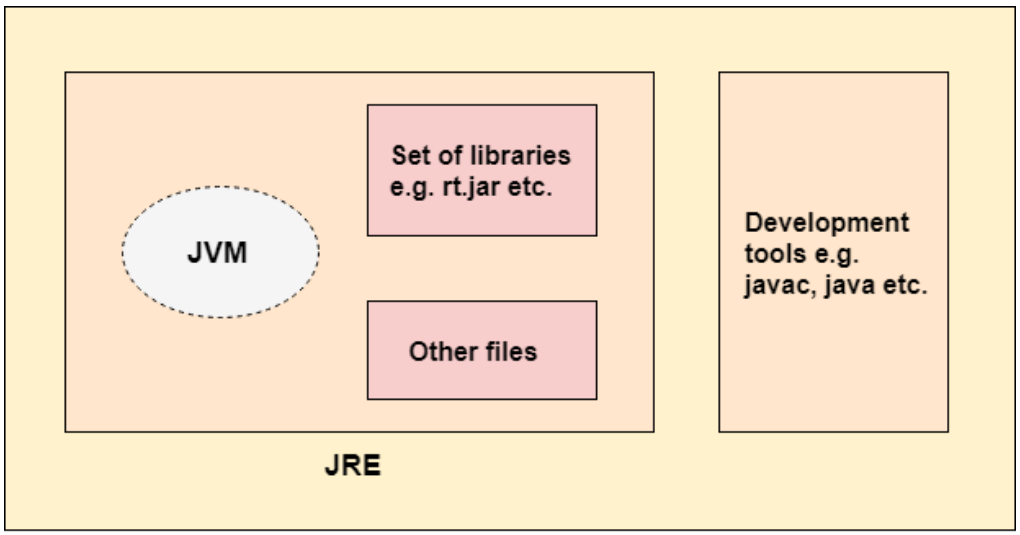
Use some part of Java in their project.

JDK vendors

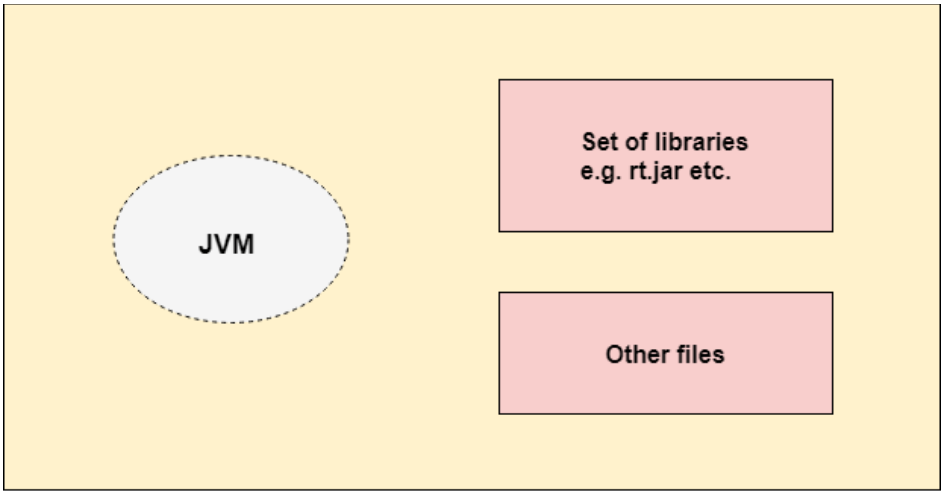
- Oracle
- **Adoptiuum OPEN-JDK [free]**
- Amazon (Coretto)
- Microsoft
- IBM
- Graal
    - https://www.theserverside.com/definition/GraalVM

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.



JDK



JRE

RAM

JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

- A specification where the working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
- An implementation Its implementation is known as JRE (Java Runtime Environment).
- Runtime Instance Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Purpose of JVM

JVM performs:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

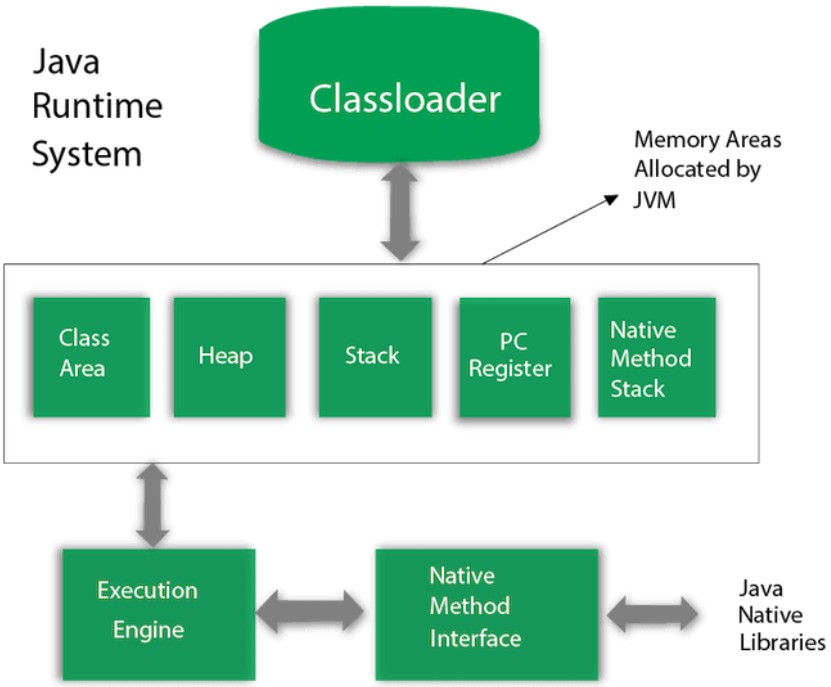
JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

https://www.javatpoint.com/jvm-java-virtual-machine

JVM architecture

JVM contains classloader, memory area, execution engine etc.



https://www.javatpoint.com/jvm-java-virtual-machine

8529f1d6fa9d

JiT	AoT
Loads the application slower than AoT since it needs to compile the application when running for the first time	Loads the page more quickly than the JiT compilation
It download the compiler and compiles the application before displaying.	It doesn't want to download the compiler, since AoT already compiles the code when building the application
Since the code include the compiler code also the bundle size will be higher.	Since it created fully compiled code and its optimized so it bundle size will be half the bundle size compiled by JiT
Suitable in development mode	Suitable in the case of production
Following command use JiT ng build, ng serve	Following command use AoT ng build --aot, ng serve --aot, ng build --prod
Template binding errors can be viewed at the time of displaying the application.	Template binding errors are shown at the time of building.

https://levelup.gitconnected.com/just-in-time-jit-and-ahead-of-time-aot-compilation-in-angular-

Few Highlights of JDK 8

1. **Lambda expressions,**
2. **Method references,**
3. **Functional interfaces,**
4. **Stream API,**
5. **Default methods,**
6. **Base64 Encode Decode,**
7. **Static methods in interface,**
8. **Optional class,**
9. **Collectors class,**
10. **ForEach() method,**
11. **Nashorn JavaScript Engine,**
12. Parallel Array Sorting,
13. Type and Repeating Annotations,
14. IO Enhancements,
15. Concurrency Enhancements,
16. JDBC Enhancements etc.
17. Joda Date time API

https://www.javatpoint.com/java-8-features

JDK installation

Installation Of JDK

1. Download jdk8 or jdk of you rchgoice
2. Set Environment Path Variables
3. Test the installation using the following commands

```
C:\Users\suman>javac -version
javac 17.0.2
C:\Users\suman>java -version
java version "17.0.2" 2022-01-18 LTS
Java(TM) SE Runtime Environment (build 17.0.2+8-LTS-86)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.2+8-LTS-86, mixed mode, sharing)
C:\Users\suman>
```

Not the above result is for JDK 17,

JDK 8 result might differ

SOURCE

https://adoptium.net/temurin/releases/?version=8

https://www.oracle.com/java/technologies/downloads/#jdk17-windows

STEPS for JDK installtion

https://youtu.be/IJ-PJbvJBGs

ECLIPSE IDE

https://www.eclipse.org/downloads/packages/release/2022-03/r/eclipse-ide-enterprise-java-and-web-developers#:~:text=Download%20Links-,Windows,-x86_64%0AmacOS%20x86_64

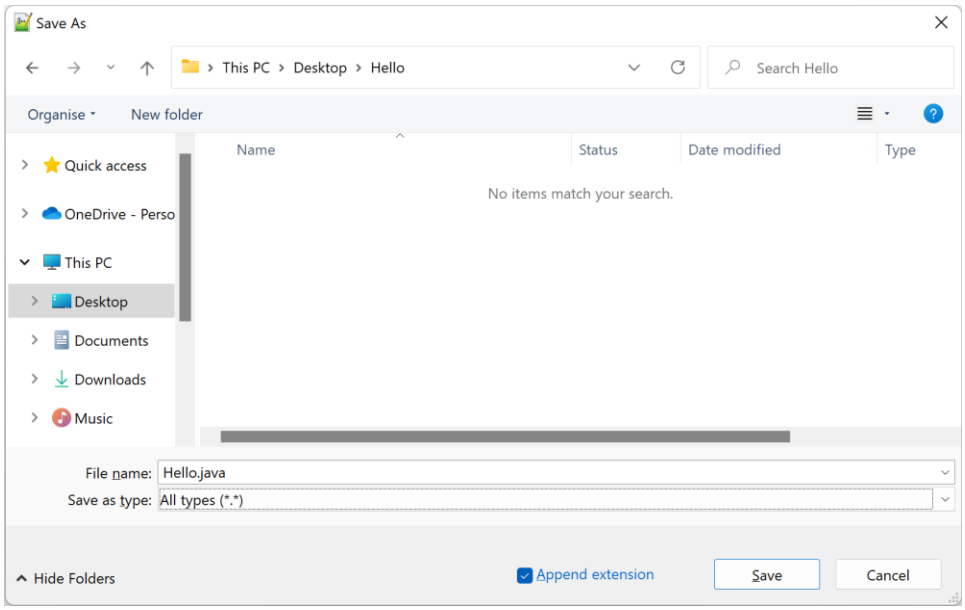
Version Check for JDK 8

```
PS C:\Users\suman> java -version
openjdk version "1.8.0_332"
OpenJDK Runtime Environment (Temurin)(build 1.8.0_332-b09)
OpenJDK 64-Bit Server VM (Temurin)(build 25.332-b09, mixed mode)
PS C:\Users\suman> javac -version
javac 1.8.0_332
PS C:\Users\suman>
```

Hello World

Hello.java

```java
public class Hello{

	public static void main(String simran[]){

	System.out.println("Hello Simran");
	System.out.println(simran.length);
//	System.out.print(simran[0]);

	}
}
```




```
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ ls
Hello.java  Hello.java.bak

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ pwd
/c/Users/suman/OneDrive/Desktop/Hello

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ javac -version
javac 11.0.15

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java -version
openjdk version "11.0.15" 2022-04-19
OpenJDK Runtime Environment Temurin-11.0.15+10 (build 11.0.15+10)
OpenJDK 64-Bit Server VM Temurin-11.0.15+10 (build 11.0.15+10, mixed mode)

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ javac Hello.java

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ ls
Hello.class  Hello.java  Hello.java.bak

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello
Hello Simran
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ javac Hello.java
```

```
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello
Hello Simran0
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ javac Hello.java

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello
Hello Simran
0
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello Mickey
Hello Simran
1
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello Mickey Minnie
Hello Simran
2
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ javac Hello.java

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello Mickey Minnie
Hello Simran
2Mickey
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ javac Hello.java

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello Mickey Minnie
Hello Simran
2
Mickey
suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$ java Hello
Hello Simran
0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
        at Hello.main(Hello.java:7)

suman@MrHour MINGW64 ~/OneDrive/Desktop/Hello
$
```

Package Naming Convention

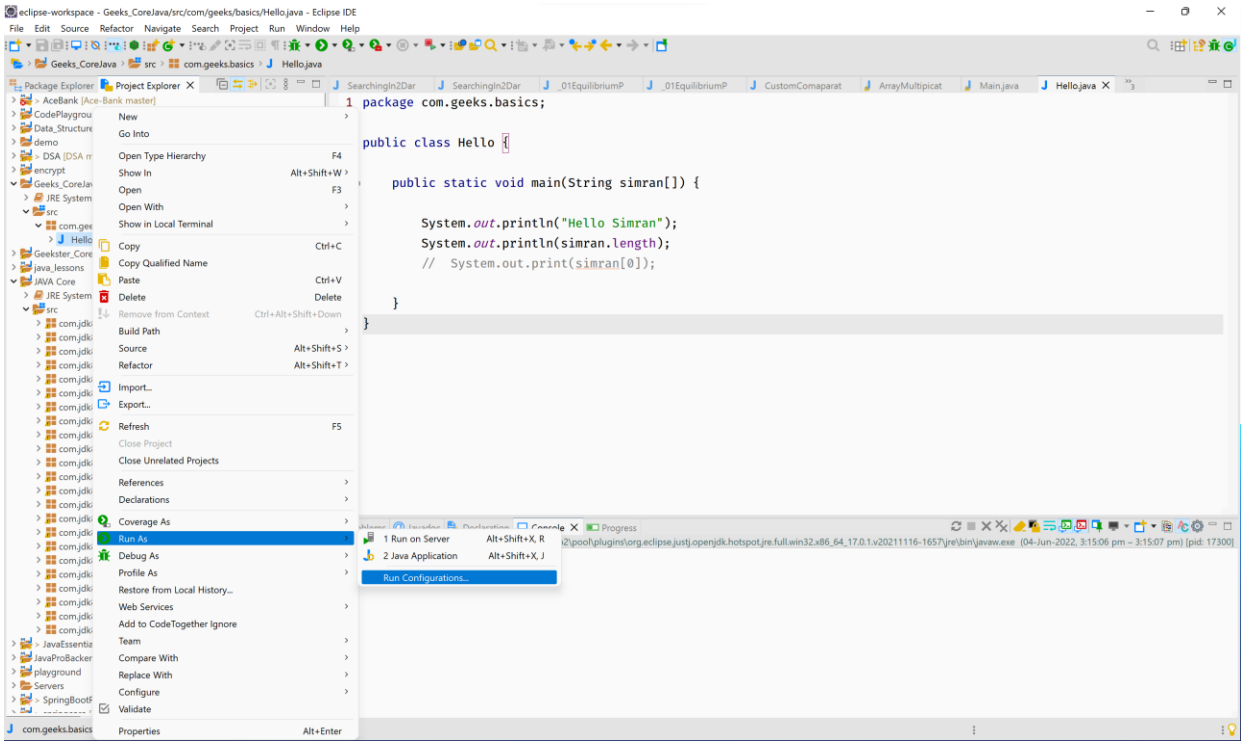
Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

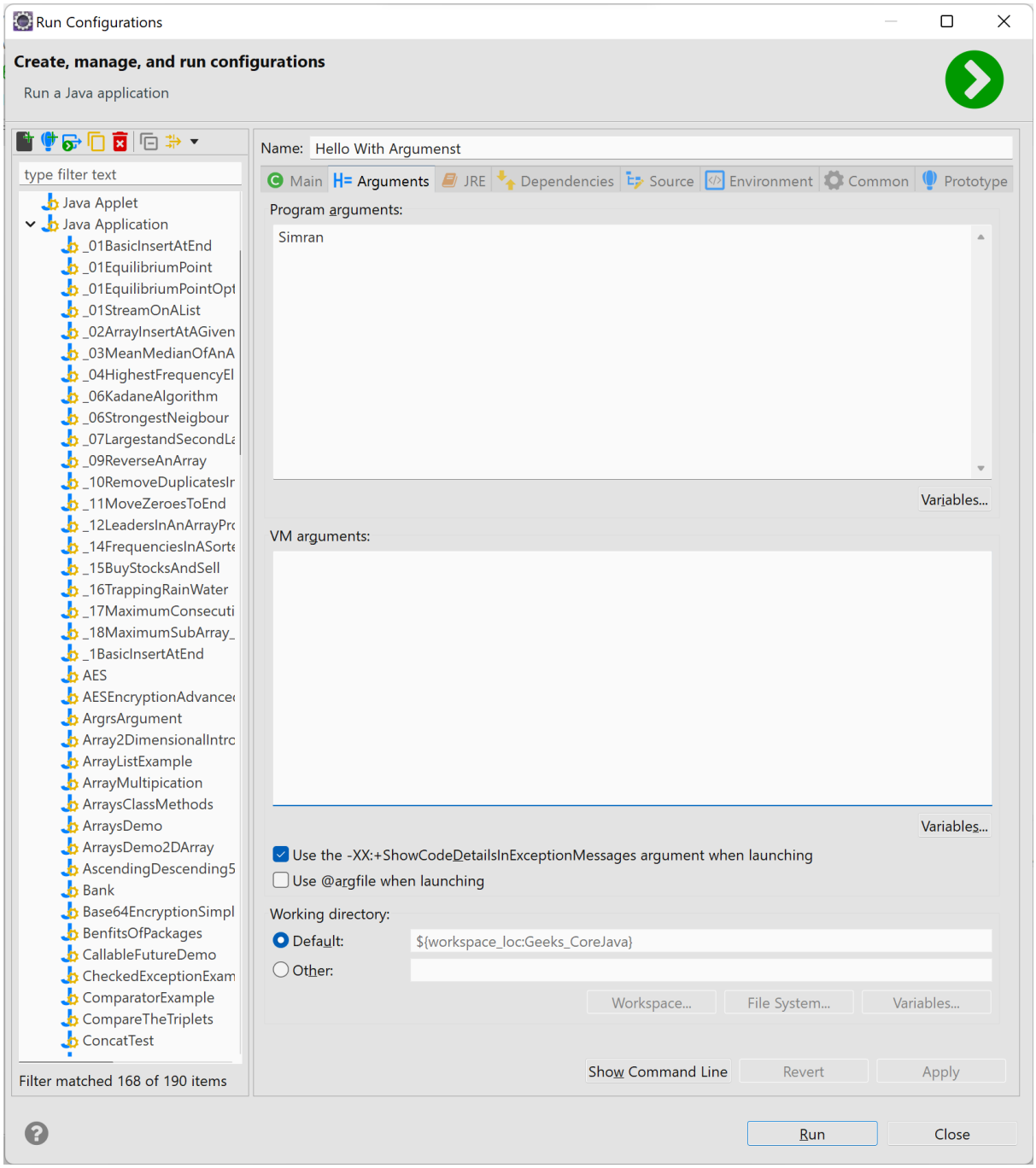
Companies use their reversed Internet domain name to begin their package names—for example, com.example.mypackage for a package named mypackage created by a programmer working at *example.com*

Website: geeks.com

Root Package: com.geeks.basics

Passing VM arguments from Eclipse




Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
Java Applet
Java Application
_01BasicInsertAtEnd
_01EquilibriumPoint
_01EquilibriumPointOpt
_01StreamOnAList
_02ArrayInsertAtAGiven
_03MeanMedianOfAnA
_04HighestFrequencyEl
_06KadaneAlgorithm
_06StrongestNeigbour
_07LargestandSecondLa
_09ReverseAnArray
_10RemoveDuplicatesIr
_11MoveZeroesToEnd
_12LeadersInAnArrayPr
_14FrequenciesInASorte
_15BuyStocksAndSell
_16TrappingRainWater
_17MaximumConsecuti
18MaximumSubArray
_1BasicInsertAtEnd
AES
AESEncryptionAdvance
ArgrsArgument
Array2DimensionalIntrc
ArrayListExample
ArrayMultipication
ArraysClassMethods
ArraysDemo
ArraysDemo2DArray
AscendingDescending5
Bank
Base64EncryptionSimpl
BenfitsOfPackages
CallableFutureDemo
CheckedExceptionExam
ComparatorExample
CompareTheTriplets
ConcatTest
Filter matched 168 of 190 items
Name: Hello With Argumenst
Main
Arguments
JRE
Dependencies
Source
Environment
Common
Prototype
Program arguments:
Simran
Variables...
VM arguments:
Variables...
Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching
Use @argfile when launching
Working directory:
Default:
${workspace_loc:Geeks_CoreJava}
Other:
Workspace...
File System...
Variables...
Show Command Line
Revert
Apply
Run
Close

Day 1

Variable Types and Its memory size

Type	Size (bits)	Minimum	Maximum	Examples
byte	8	-2^7	$2^7 - 1$	*byte b = 100;*
short	16	-2^{15}	$2^{15} - 1$	*short s = 30_000;*
int	32	-2^{31}	$2^{31} - 1$	*int i = 100_000_000;*
long	64	-2^{63}	$2^{63} - 1$	*long l = 100_000_000_000_000;*
float	32	-2^{-149}	$(2-2^{-23})\cdot 2^{127}$	*float f = 1.456f;*
double	64	-2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	*double f = 1.456789012345678;*
char	16	0	$2^{16} - 1$	*char c = 'c';*
boolean	1	–	–	*boolean b = true;*

The Unicode Consortium

The Unicode Consortium develops the Unicode Standard. Their goal is to replace the existing character sets with its standard Unicode Transformation Format (UTF).

The Unicode Standard has become a success and is implemented in HTML, XML, Java, JavaScript, E-mail, ASP, PHP, etc. The Unicode standard is also supported in many operating systems and all modern browsers.

The Unicode Consortium cooperates with the leading standards development organizations, like ISO, W3C, and ECMA.

The Unicode Character Sets

Unicode can be implemented by different character sets. The most commonly used encodings are UTF-8 and UTF-16:

Character-set	Description
UTF-8	A character in UTF8 can be from 1 to 4 bytes long. UTF-8 can represent any character in the Unicode standard. UTF-8 is backwards compatible with ASCII. UTF-8 is the preferred encoding for e-mail and web pages
UTF-16	16-bit Unicode Transformation Format is a variable-length character encoding for Unicode, capable of encoding the entire Unicode repertoire. UTF-16 is used in major operating systems and environments, like Microsoft Windows, Java and .NET.

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	⁄	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

Figure 1 Ascii Chart

Important ASCII Values

- 65 - A
- 97 - a
- 48 - **0**

Keywords in Java

Keywords are reserved words in Java

1.	**abstract**	Specifies that a class or method will be implemented later, in a subclass
2.	**assert**	Assert describes a predicate placed in a java program to indicate that the developer thinks that the predicate is always true at that place.
3.	**boolean**	A data type that can hold True and False values only
4.	**break**	A control statement for breaking out of loops.
5.	**byte**	A data type that can hold 8-bit data values
6.	**case**	Used in switch statements to mark blocks of text
7.	**catch**	Catches exceptions generated by try statements
8.	**char**	A data type that can hold unsigned 16-bit Unicode characters
9.	**class**	Declares a new class
10.	**continue**	Sends control back outside a loop
11.	**default**	Specifies the default block of code in a switch statement
12.	**do**	Starts a do-while loop
13.	**double**	A data type that can hold 64-bit floating-point numbers
14.	**else**	Indicates alternative branches in an if statement
15.	**enum**	A Java keyword is used to declare an enumerated type. Enumerations extend the base class.
16.	**extends**	Indicates that a class is derived from another class or interface
17.	**final**	Indicates that a variable holds a constant value or that a method will not be overridden
18.	**finally**	Indicates a block of code in a try-catch structure that will always be executed
19.	**float**	A data type that holds a 32-bit floating-point number
20.	**for**	Used to start a for loop
21.	**if**	Tests a true/false expression and branches accordingly
22.	**implements**	Specifies that a class implements an interface
23.	**import**	References other classes
24.	**instanceof**	Indicates whether an object is an instance of a specific class or implements an interface
25.	**int**	A data type that can hold a 32-bit signed integer
26.	**interface**	Declares an interface
27.	**long**	A data type that holds a 64-bit integer
28.	**native**	Specifies that a method is implemented with native (platform-specific) code
29.	**new**	Creates new objects
30.	**null**	This indicates that a reference does not refer to anything
31.	**package**	Declares a Java package

32.	**private**	An access specifier indicating that a method or variable may be accessed only in the class it's declared in
33.	**protected**	An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package)
34.	**public**	An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
35.	**return**	Sends control and possibly a return value back from a called method
36.	**short**	A data type that can hold a 16-bit integer
37	**static**	Indicates that a variable or method is a class method (rather than being limited to one particular object)
38.	**strictfp**	A Java keyword is used to restrict the precision and rounding of floating-point calculations to ensure portability.
39.	**super**	Refers to a class's base class (used in a method or class constructor)
40.	**switch**	A statement that executes code based on a test value
41.	**synchronized**	Specifies critical sections or methods in multithreaded code
42.	**this**	Refers to the current object in a method or constructor
43.	**throw**	Creates an exception
44.	**throws**	Indicates what exceptions may be thrown by a method
45.	**transient**	Specifies that a variable is not part of an object's persistent state
46.	**try**	Starts a block of code that will be tested for exceptions
47.	**void**	Specifies that a method does not have a return value
48.	**volatile**	This indicates that a variable may change asynchronously
49.	**while**	Starts a while loop

Statically & Dynamically Typed Languages

A programming language is statically typed if the type of a variable is known at compile time. A language is dynamically typed **if the type of a variable is checked during run-time.**

Statically-Typed

We call a language "statically-typed" if it follows type checking during compilation. So, every detail about the variables and all the data types must be known before we do the compiling process.

Some examples of statically-typed languages are Java, C, C++, C#, Swift, Scala, Kotlin, Fortran, Pascal, Rust, Go, COBOL, etc.

Though JAVA has a dynamic type in recent JDKs

```
int mangoes = 10;
```

Dynamically -typed

We call a language "dynamically typed" if type checking takes place while the program runs (run-time). In this type of language, there is no need to specify the data type of each variable while writing code.

Most modern programming languages are dynamically typed. Some examples of dynamically-typed languages are Python, Javascript, Ruby, Perl, PHP, R, Dart, Lua, Objective-C, etc.

Sample.py

```
i =6.0
print(type(i))
```

Execution for a JAVA code

JVM, JRE and JDK are platform **dependent** because the configuration of each OS differs. But Java is platform-independent.

1. **Java Development Kit (JDK**) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

JDK usage [Build, Compile, and Execute Application]

Java follows WORA (Write once read anywhere)

2. **JRE** (Java Runtime Environment) is an installation package that provides an environment to only run (not develop) the java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

3. **JVM** (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.

A computer only understands Binary Numbers 0, 1

Steps to Execute JAVA Code

1. Write your Java code in a .java file [Source Code]
2. The compilation (Conversion of a Source code to a more machine-friendly language)
   The compilation will give you a .class file (.class file is also known as *ByteCode*)
3. Execution
   Run the compiled .class file using the following commands.

```
C:\Users\suman\Downloads\Y Hills\Source Code\Basics>javac Main.java

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java Main
Light Speed2.9979445867687994E8
Light Speed@
```

RULES

- You must have to obey
- Apart from imports, annotations and package names, all will reside inside a class
- The name of the file should match the class name
- Care should be taken for `public static void main(String[] args) {`

Convention [Good Habits]

You should follow for a becoming a good dev

- The naming convention for variable and method name is = camelCase simranBhat
- The naming convention for constants: MAX_VALUE
- The naming convention For Classes is: PascalCase SimranBhat

VARIABLES

- We cannot declare keywords as a variable name
- Cannot start a variable name with a number
- No fixed length for variable name

PrimitiveDataTypes.java

```
package com.geeks.basics;

public class PrimitiveDataTypes {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

//		bit = 0/1
//		qBit = 0/1/ [0|1]

//		***************** integers *****************
		byte a = 9;
		short b = 50;
		int c = 40000;
		long d = 67l;

		short sh = (short) c;// explicit type casting

//		***************** decimals/real numbers *****************

		float e = 6.5f;
		double f = 6.5;// IEEE format to store

//		***************** character *****************
		char g = 'd';
		int x = g;
		System.out.println(x);// UTF 8
		char rupee = '₹';// if added UTF character the, java file should also be
encoded in character UTF
		x = rupee;// (implicit type casting) char to rupee
		System.out.println(x);
		int m = c;
		char exp = (char) m;// explicit type casting [int to char]
		System.out.println(exp);

		m = Integer.MAX_VALUE;
		exp = (char) m;// explicit type casting
		System.out.println(exp);

//		***************** boolean *****************
		boolean isHarrd = true;
		isHarrd = false;

	}

}
```

PrimitiveDataTypesSpeciaCase.java

```
package com.geeks.basics;

public class PrimitiveDataTypesSpeciaCase {

	public static void main(String[] args) {

		System.out.println(Byte.MAX_VALUE);
		byte b = 127;// range is checked
		b += 2;
//		b= b+1;
		System.out.println(b);// -128

//		Static type language advantage is fast and stable execution
		int x =9;
		float m = x;

//		JDK 10/11 we have dynamic type option
//		var x =9;
//		var g = x;

	}

}
```

Wrapper Classes

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need for Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

https://www.geeksforgeeks.org/wrapper-classes-java

WrapperDataTypes.java

```java
package com.geeks.basics;

import java.math.BigDecimal;
import java.math.BigInteger;

public class WrapperDataTypes {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

//		bit = 0/1
//		qBit = 0/1/ [0|1]

//		Byte byte1 = new Byte((byte) 7);//Old way

//		Integers
		Byte a = 9;
		Short b = 50;
		Integer c = 40000;
		Long d = 67l;

//		Decimals/Real
		Float e = 6.5f;
		Double f = 6.5;// IEEE format to store

//		Character
		Character g = 'd';
		int maxValue = Integer.MAX_VALUE;

//		Boolean
		Boolean isEasy = true;
	}

}
```

Operators in Java

Main.java

```
//package name
//import statements
//Annotations

public class Main {

          public static void main(String[] args) {
//                  INTEGER
                    byte a = 5;
                    short s = 77;
                    int age = 24;
                    int totalAges = age + 56 + 87;
                    long distance = 453543;

                    age++;// Post Increment Operator
                    ++age;// Pre Increment
                    --age;// Pre Decrement
                    age--;// Post Decrement

//                  DECIMAL
                    float bankBalance = 1000.67f;
                    double speedOfLight = 299_794_458.67__687_996;

//                  BOOLEAN
                    boolean isDay = false;

//                  CHAR
                    char firstAlphabet = 'A';// UTF16

                    System.out.println("Light Speed" + speedOfLight);
                    System.out.println("BALANCE " + bankBalance);
                    System.out.println("Light Speed" + --firstAlphabet);

          }
}
```

BasicOperators.java

```
package com.geeks.basics;

public class BasicOperators {

	public static void main(String[] args) {

		// Increment, Decrement Operators

		// POST Increment Operators
		int x = 9;
		x++;// Post Increment
		System.out.println(x);// 10
		System.out.println(x++);// 10
		System.out.println(x);// 11

		// PRE Increment Operators
		int y = 9;
//		++y;
//		System.out.println(y);
		System.out.println(++y);// 10

//		Q1
		x = 7;
		y = 9;
		y = ++x + ++y + y++ + ++x + y++;// 8 + 10 + 10 + 9 + 11
		System.out.println(y);// 48

//		Q2 [to solve]
		x = 7;
		y = 9;
		y = ++x + ++y + y++ + -y-- + --y - ++x + +--y + y++ + x--;// 8 + 10 + 10 + 9
+ 11
		System.out.println(y);

//		Shorthand Operators format
		int m = 9;
		m *= 7;
		System.out.println(m);

//		Bit Wise Operators
		System.out.println(7 & 90);
		System.out.println(7 | 90);
		System.out.println(~-7);
		System.out.println(6 ^ 7);

//		Power method in Java
		double pow = Math.pow(2, 8);
		System.out.println((int) pow);

	}
}
```

TRUTH table of few logical bitwise operator

AND		
x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
x	y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

NOT	
x	~x
0	1
1	0

LoopsAndIfCondition.java

```
public class LoopsAndIfCondition {

	public static void main(String[] args) {
		// SUM of 1st n natural numbers
		int n = 10000;
		int sum = 0;

//		Solution 1 [For iteration]
		for (int i = 1; i <= n; i++) {
			sum = sum + i;
		}
		System.out.println("SUM 1 ~> "+sum);


//		Solution 2 [Maths]
		long start = System.nanoTime();
		sum = n*(n+1)/2;
		long end = System.nanoTime();
		System.out.println("SUM 2 ~> "+sum);
		System.out.println("TIME ~> "+(end-start));
	}
}
```

IsEvenNumber.java

```
package com.geeks.dsa.maths;

import java.util.Scanner;

public class IsEvenNumber {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner scanner = new Scanner(System.in);

		while (true) {
			System.out.println("Enter a number for even check");
			int num = scanner.nextInt();

			boolean ans = isEven5(num);
			System.out.println(ans);
		}

	}

	private static boolean isEven1(int num) {
		if (num % 2 == 0)
			return true;
		else if (num % 2 != 0)
			return false;

		return false;
	}

	private static boolean isEven2(int num) {
		if (num % 2 == 0)
			return true;
		else
			return false;
	}

	private static boolean isEven3(int num) {
		return (num % 2 == 0);
	}

	private static boolean isEven4(int num) {
		if ((num & 1) == 0)
			return true;
		else
			return false;
	}

	private static boolean isEven5(int num) {
		return (num % 2 != 0) ? false : true;//ternary operator
	}

}
```

We will cover it in the next class from here

Day 2

Objects

Everything in the world is an object and every object has maily 2 parts

- Has
  - Colour
  - Weight
  - SA
- Does
  - Input
  - RGB lights

Packages

Package Intro

A java package is a group of similar types of classes, interfaces and sub-packages.

Advantage of Java Packages

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Naming Convention

Say Website is: www.jdk8.com

So package name will start as **com.jdk8.####**

Internally it will create a nested folder structure like

- com
  - jdk8
    - ####

Loops

Scope Of Variables

The reach of a variable will depend on where it is declared.

Continue and Break Statements

- Continue
- Break

The above two keywords can be used to further control the loops

While Loop

While loop can be also used for looping

- do while
- while

The basic difference between do-while and while loop is that do while will at least execute for one time.

SimpleForLoop.java

```
package com.jdk8.basics;

public class SimpleForLoop {

	public static void main(String[] args) {
		// SUM of 1st n natural numbers
		int n = 10000;
		int sum = 0;
//		Solution 1 [For iteration]
		for (int i = 1; i <= n; i++) {
			sum = sum + i;
		}
		System.out.println("SUM 1 ~> " + sum);

//		System.out.println(i);
		for (int i = 1; i <= n;) {
			sum = sum + i;
			int z = 8;// declaration
			i++;
		}

//		Solution 2 [Maths]
		long start = System.nanoTime();
		sum = n * (n + 1) / 2;
		long end = System.nanoTime();
		System.out.println("SUM 2 ~> " + sum);
		System.out.println("TIME ~> " + (end - start));

//		Break And Continue Statements
		System.out.println("---> ");
		for (int z = 1; z <= 20; z++) {
			if (z % 5 == 0) {
//				break;
				continue;
			}
			System.out.println(z);
		}

	}
}
```

WhileLoop.java

```
package com.jdk8.basics;

public class WhileLoop {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		int i = 11;
//		while loop
		while (i <= 10) {
			System.out.println(i);
			i++;
		}

		int j = 11;
//		do while loop
		do {
			System.out.println("REACHED DO");
			j++;
		} while (j < 10);

//There are other loops which we will discuss later
//		foreach
//		interator

	}
}
```

Wrapper Classes

Intro

The Wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Autoboxing and Unboxing

We also have autoboxing and unboxing features that convert primitives into objects and objects into primitives automatically. The automatic conversion of a primitive into an object is known as autoboxing and vice-versa unboxing.

WrapperClass.java

```
package com.jdk8.oop.basics;

public class WrapperClass {

	public static void main(String[] args) {

			String name = "rishav";
//			Primitive Variables
			int number = 10;
			boolean isSad = false;
//			Wrapping int to an Integer Object
			Integer integer = new Integer(10);

//			Wrapping boolean  to an Boolean Object
			Boolean boolean1 = new Boolean(true);

			byte by = 8;
			int p = by;

			// Generally You cannot fit a higher space data in lower space [Bucket - Mug
			// Analogy]
//			int m = 8;
//			byte c=m;

//			Here we have exception as long[64bit] is getting stored in float[32 bit],
//			The reason is that both have different storing algorithms, and floating
numbers is  getting stored using higher efficiency IEEE algo
			long lon = Long.MAX_VALUE;
			float fl = lon;
			System.out.println(fl);

			byte b = 10;
			short s = 20;
			int i = 30;
			long l = 40;
			float f = 50.0F;
			double d = 60.0D;
			char c = 'a';
			boolean b2 = true;

			// Autoboxing: Converting primitives into objects
			// With this mechanism So no need of "new" keyword
			Byte byteobj = b;
			Short shortobj = s;
			Integer intobj = i;
			Long longobj = l;
			Float floatobj = f;
			Double doubleobj = d;
			Character charobj = c;
			Boolean boolobj = b2;

			// Unboxing: Converting Objects to Primitives Automatically ,again no use of
			// "new" keyword
			byte bytevalue = byteobj;
			short shortvalue = shortobj;
			int intvalue = intobj;
			long longvalue = longobj;
			float floatvalue = floatobj;
			double doublevalue = doubleobj;
			char charvalue = charobj;
			boolean boolvalue = boolobj;

	}

}
```

OOP [Object Orientation Programming] Intro

Intro

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behaviour.

Say a **mixer grinder**

1. Its Properties [has]
    a. Wattage
    b. Colour
    c. Speed
2. Its Functionality [does]
    a. grind()
    b. blend()
    c. juiceOut()

We use classes in OOP,

Classes are just like Formula/Recipe or a prototype for an Object.

Think like to make a Cake we need a Recipe

Recipe ---> Cake

Secondary [HDD, SSD 10x, Optane13x] *permanent*

Primary [RAM DDR3, 4, 5] *volatile*

CPU <-> RAM

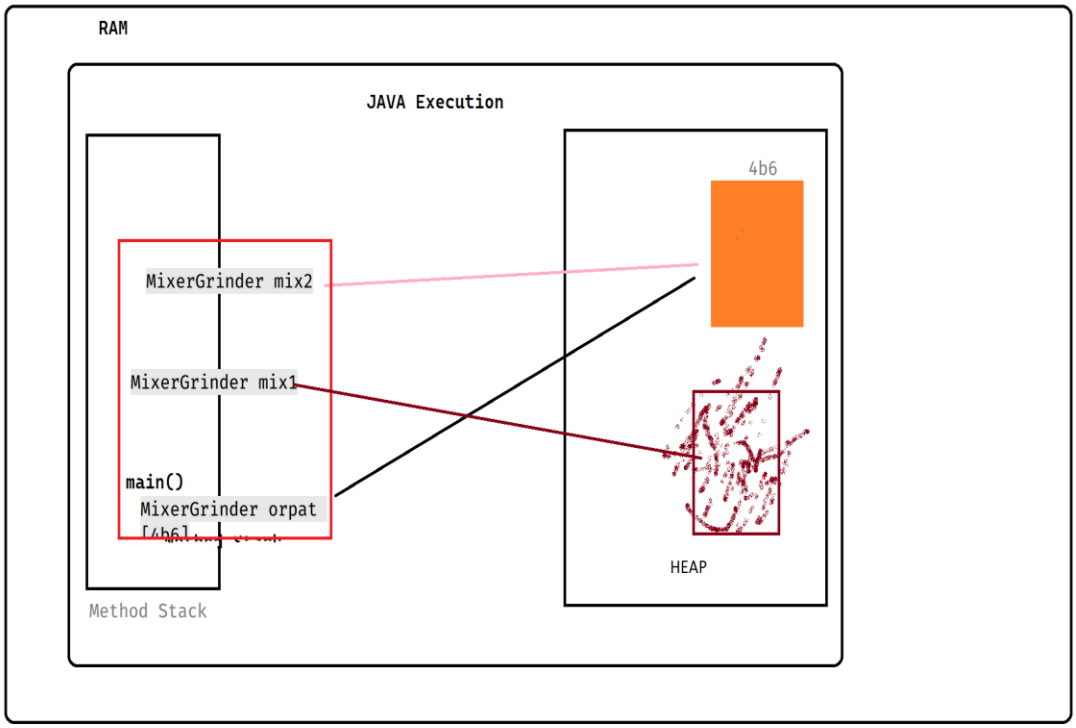


Figure 2 OOP Memory Discussion

MixerGrinder.java

```
package com.jdk8.oop.basics;

public class MixerGrinder {

	//Instance variable/Object Variables
	int wattage;
	String color;
	int speed;

	void grind(){
		System.out.println("GRINDING");
	}

	void mixing(){
		System.out.println("MIXING");
	}

	void blending(){
		System.out.println("BLENDING");
	}
```

```
	void readOutSpecs() {
		System.out.println("Wattage -> "+wattage);
		System.out.println("Speed -> "+speed);
		System.out.println("Color -> "+color);

	}

}
```

MixerBuy.java

```
package com.jdk8.oop.basics;

public class MixerBuy {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		MixerGrinder orpat =new MixerGrinder();//orpat as a Object Reference
		//MixerGrinder()
		System.out.println(orpat);//By default it will print the memory address



		orpat.readOutSpecs();// preint the defaulyt values


//		Default Values
//		integer --> 0
//		float/double -> 0.0
//		Objetct refer nce - null
//		String - null
//		char -> null char
//		boolean --> false

		orpat.color="Pink";
		orpat.speed=20000;
		orpat.wattage=240;

		orpat.readOutSpecs();



		System.out.println("\n\n\n\n mix1");
		MixerGrinder mix1;
		mix1 = new MixerGrinder();
		System.out.println(mix1);
		mix1.readOutSpecs();


		MixerGrinder mix2= orpat;
		System.out.println(mix2);
		mix2.readOutSpecs();

		orpat.readOutSpecs();
//		mix1.readOutSpecs();

		mix1=null;
		orpat=null;

	}

}
```

Garbage and Garbage Collector

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Day 3

Scanner Class

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program.

ScannerInput.java

```
package com.jdk8.basics;

import java.util.Scanner;

public class ScannerInput {

	public static void main(String[] args) {

//			Scanner Class [Inbuilt Class for taking input]
			Scanner sc = new Scanner(System.in);

			System.out.println("Enter your Age");// this will go to next line after
printing
			System.out.print("Enter your name");// this will not change the line after
printing
			System.out.println();// blank empty line
//			System.out.print();//Error

//			String next = sc.next();//inputs only one word
			String next = sc.nextLine();// inputs whole sentence
			System.out.println(next);

			float height = sc.nextFloat();
			System.out.println("Height ~> " + height);
			System.out.println("Enter a number to be squared");
			int integer = sc.nextInt();
			int result = findSqr(integer);
			System.out.println(result);

			sc.close();// closing the Scanner resource

	}

	static int findSqr(int x) {// formal parameters
			int result = x * x;
			return result;
	}

}
```

ScannerInputCornerCase.java

```
package com.jdk8.basics;

import java.util.Scanner;

public class ScannerInputCornerCase {

	public static void main(String[] args) {

//			Scanner Class [Inbuilt]
			System.out.println("START");
			Scanner sc = new Scanner(System.in);

			System.out.println("Enter the age");
			int age = sc.nextInt();
			System.out.println("AGE -> " + age);

			System.out.println("Enter the name");
			sc.nextLine();// Flushing
			String name = sc.nextLine();
			System.out.println("NAME -> " + name);

			System.out.println("END");

	}

}
```

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.

Advantages

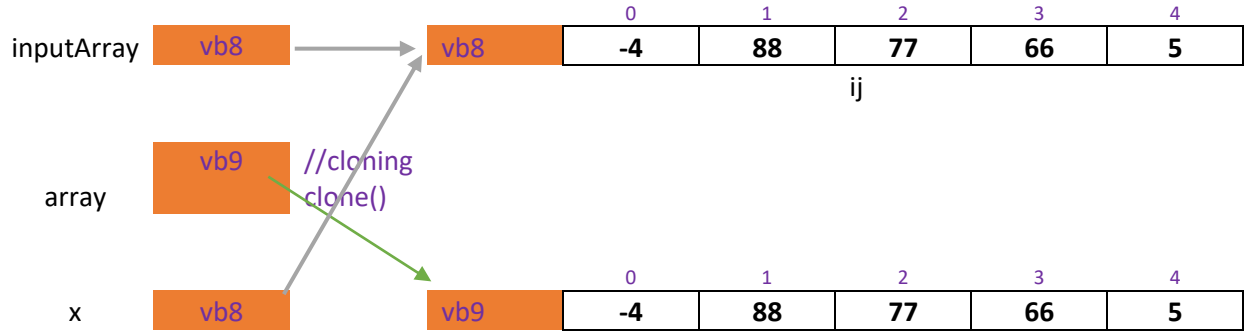
- Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
- Random access: We can get any data located at an index position.

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.

1 Dimesional Array

Memory Discusion



Examples

OneDArray.java

```
package com.jdk8.arrays;

import java.util.Scanner;

public class OneDArray {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		int zeroLengthArray[] = new int[0];// Zero length array is possible in Java
		int[] x = new int[3];

		Scanner sc = new Scanner(System.in);
		for (int i = 0; i < x.length; i++) {// Print the initial contents of array.
It will show the default values
			System.out.print(x[i] + " ");
		}

		for (int i = 0; i < x.length; i++) {// Taking Input
			x[i] = sc.nextInt();
		}

		for (int i = 0; i < x.length; i++) {// Print the contents of array
			System.out.print(x[i] + " ");
		}

	}

}
```

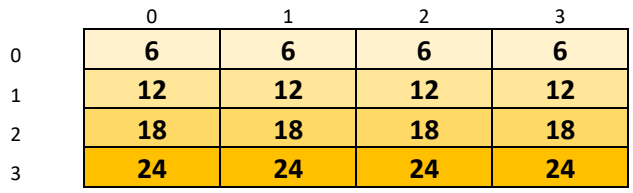
ReverseAnArray.java

```
package com.jdk8.arrays;

public class ReverseAnArray {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		int inputArray[] = { 5, 66, 77, 6, 88, -4 };// hardcoding a 1D array
elements

		int array[] = inputArray.clone();// cloning a different Array
		print1DArray(array);
		System.out.println();
//		int[] reversedArray = reverseArray(inputArray);
//		print1DArray(reversedArray);

		reverseArrayOptimized(array);
		print1DArray(inputArray);

	}

	static void reverseArrayOptimized(int[] x) {
		int i = 0, temp;
//		System.out.println(temp);//Error
		int j = x.length - 1;
		while (i <= j) {
			temp = x[i];
			x[i] = x[j];
			x[j] = temp;
			i++;
			j--;
		}
//		return inputArray;
	}

	static int[] reverseArray(int[] inputArray) {

		int[] reversedArray = new int[inputArray.length];

		int z = 0;
		for (int i = inputArray.length - 1; i >= 0; i--) {
			reversedArray[z++] = inputArray[i];
		}

		return reversedArray;

	}

	static void print1DArray(int[] inputArray) {

		for (int i = 0; i < inputArray.length; i++)

			System.out.print(inputArray[i] + " ");

	}

}
```

2 Dimensional Array

It will contain Rows and Columns, A 2D array is just a collection of several 1 D arrays.

	0	1	2	3
0	6	6	6	6
1	12	12	12	12
2	18	18	18	18
3	24	24	24	24



So here we have 4 1D arrays in this 2D array.

Examples

ArrayMultipication.java

```
package com.jdk8.arrays;

public class ArrayMultipication {

	static void printMatrix2D(int M[][]) {

		for (int i = 0; i < M.length; i++) {// number of 1D array in a 2d array
			for (int j = 0; j < M[i].length; j++)
				System.out.print(M[i][j] + " ");

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

Array2DimensionalIntro.java

```
package com.jdk8.arrays;

import java.util.Scanner;

public class Array2DimensionalIntro {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		int[][] array2D = {
					{ 1, 2, 3 },
					{ 4, 5, 6 }// 2*3 2Darray
		};

		ArrayMultipication.printMatrix2D(array2D);

		int[][] arrayExample = new int[2][3];
		ArrayMultipication.printMatrix2D(arrayExample);

		Scanner scanner = new Scanner(System.in);
		for (int i = 0; i < arrayExample.length; i++) {// number of 1D array in a 2d
array
			for (int j = 0; j < arrayExample[i].length; j++)
				arrayExample[i][j] = scanner.nextInt();
		}

		ArrayMultipication.printMatrix2D(arrayExample);


	}

}
```

VarArray.java

```
package com.java8.oops.arrays;

import java.util.Arrays;

public class VarArray {

	public static void main(String... args) {
		printString("Fox", "Lion", "Bear", "Fish", "Koala");
	}

	private static void printString(String s) {
		System.out.println(s);
	}

	private static void printString(String s1, String s2) {
		System.out.println(s1 + " " + s2);
	}

	private static void printString(String... str) {
//		System.out.println(s1);
		System.out.println(Arrays.toString(str));
	}

}
```

Day 4

2D jagged Array

2D Array Can also be jagged, which means it also can be non-symmetric. So individual 1D arrays of a 2D array will be of different lengths.

	0	1	2	3
0	**6**	**6**	**6**	**6**
1	**12**	**12**	**12**	
2	**18**	**18**		
3	**24**	**24**	**24**	

In the above example, the four 1D arrays are of different lengths, hence it's a jagged 2D array.

JaggedArray.java

```
package com.jdk8.arrays;

public class JaggedArray {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		int B[][] = { { 1, 1, 1, 1 }, { 2, 2, 2 }, { 3, 3, 3, 3, 5, 7 } };

//		Printing the 2D jagged Array
		for (int i = 0; i < B.length; i++) {// Here B.length = 3
			for (int j = 0; j < B[i].length; j++) {
				System.out.print(B[i][j] + " ");
			}
```

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

2D Matrix Multiplication Problem

	0	1	2
0	1	1	1
1	2	2	2
2	3	3	3
3	4	4	4

Matrix A

*m * n* *4*3*

×

	0	1	2	3
0	1	1	1	1
1	2	2	2	2
2	3	3	3	3

Matrix B

*n*k* *3*4*

=

	0	1	2	3
0	1*1 + 2*1 + 3*1	1*1 + 2*1 + 3*1	1*1 + 2*1 + 3*1	1*1 + 2*1 + 3*1
1	1*2 + 2*2 + 3*2	1*2 + 2*2 + 3*2	1*2 + 2*2 + 3*2	1*2 + 2*2 + 3*2
2	1*3 + 2*3 + 3*3	1*3 + 2*3 + 3*3	1*3 + 2*3 + 3*3	1*3 + 2*3 + 3*3
3	1*4 + 2*4 + 3*4	1*4 + 2*4 + 3*4	1*4 + 2*4 + 3*4	1*4 + 2*4 + 3*4

Resultant Matrix C

*m*k* *4*4*

	0	1	2	3
0	6	6	6	6
1	12	12	12	12
2	18	18	18	18
3	24	24	24	24

Resultant Matrix C

ArrayMultipication.java

```
package com.jdk8.arrays;

public class ArrayMultipication {

	static void printMatrix2D(int M[][]) {

		for (int i = 0; i < M.length; i++) {// number of 1D array in a 2d array
			for (int j = 0; j < M[i].length; j++)
				System.out.print(M[i][j] + " ");

			System.out.println();
		}
	}

	// Method to multiply two matrix
	// two matrices A[][] and B[][] should be symmetric
	static void multiplyMatrix(int A[][], int B[][]) {

		int row1 = A.length;
		int col1 = A[0].length;

		int row2 = B.length;
		int col2 = B[0].length;

		// Print the matrices A and B
		System.out.println("\nMatrix A:");
		printMatrix2D(A);
		System.out.println("\nMatrix B:");
		printMatrix2D(B);

		// Checking for if multiplication is Possible
		if (row2 != col1) {
			System.out.println("\nMultiplication Not Possible EXITING");
			return;
//			System.exit(3);
		}

		// Initiating Matrix to store the result
		// The product matrix will be of size row1 x col2
		int C[][] = new int[row1][col2];

		int i, j, k;
		// Multiply the two matrices
		for (i = 0; i < C.length; i++) {// row1 = A.length = C.length
			for (j = 0; j < C[0].length; j++) {// col2 = B[0].length = C[0].length

				for (k = 0; k < B.length; k++)//
					C[i][j] = C[i][j] + (A[i][k] * B[k][j]);

			}
		}

		// Print the Resultant Matrix
		System.out.println("\nResultant Matrix:");
		printMatrix2D(C);
	}

	public static void main(String[] args) {
		System.out.println("START");
		int row1 = 4, col1 = 3, row2 = 3, col2 = 4;

		int A[][] = { { 1, 1, 1 }, { 2, 2, 2 }, { 3, 3, 3 }, { 4, 4, 4 } };// (m*n)=4 * 3

		int B[][] = { { 1, 1, 1, 1 }, { 2, 2, 2, 2 }, { 3, 3, 3, 3 } };// (n*k)=3*4
//		int B[][] = { { 1, 1, 1, 1 }, { 2, 2, 2, 2 }, { 3, 3, 3, 3 }, { 4, 4, 4, 4 } };//
(n*k)=3*4
		multiplyMatrix(A, B);
		System.out.println("EXIT");
	}
}
```

Operators in Java

Operators.java

```
package com.jdk8.basics;

public class Operators {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		int i = 9;//assignment operator

		i++;// Post Increment
		i--;// Post Decrement

		++i;// Pre increment
		--i;// Pre Decrement

//		Shorthand Operator
		i+=7;//i=i+7
		i-=8;
		i%=7;

		if(i==5)//Comparison Operator >   <  >=   <=  !=
			System.out.println("Equals 5");

//		BitWise Operator
		int x = 8;//TO DO


	}

}
```

There are few operators in Java. Which we will discuss later.

String Args []

I we want to use some runtime arguments, We can use String args[] of main()

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

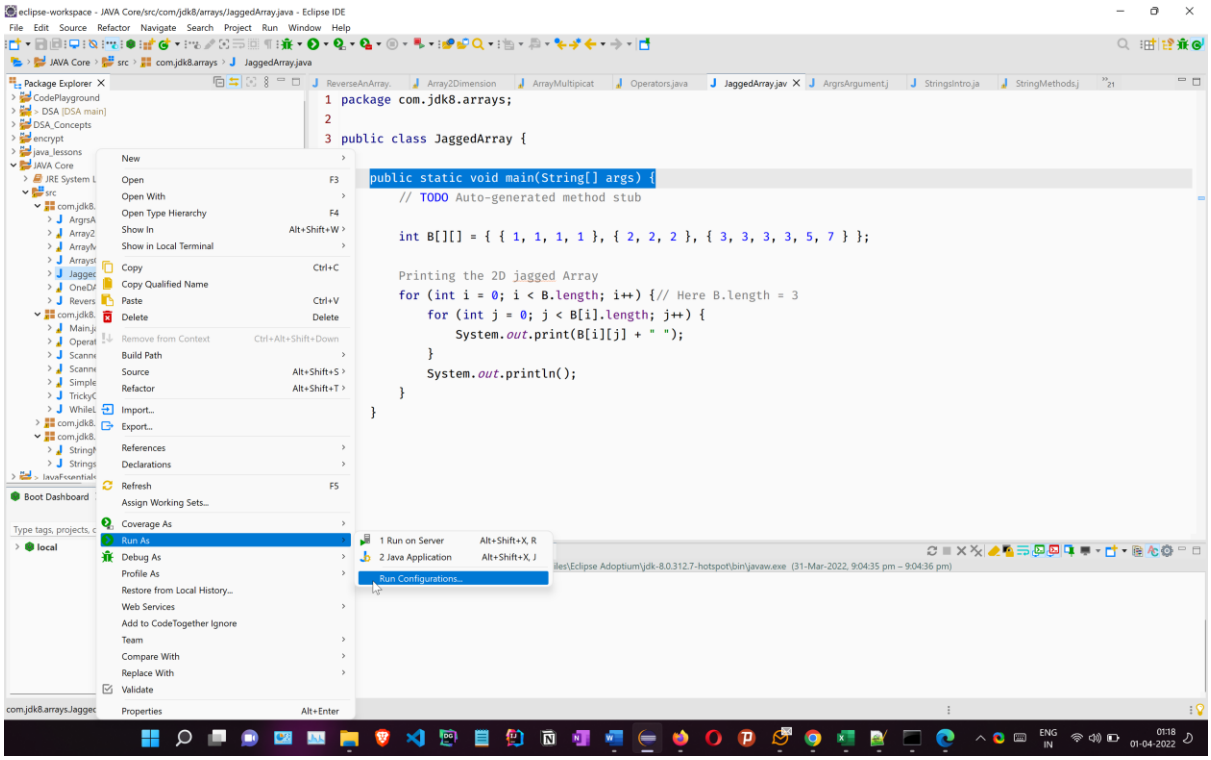
ArgrsArgument.java

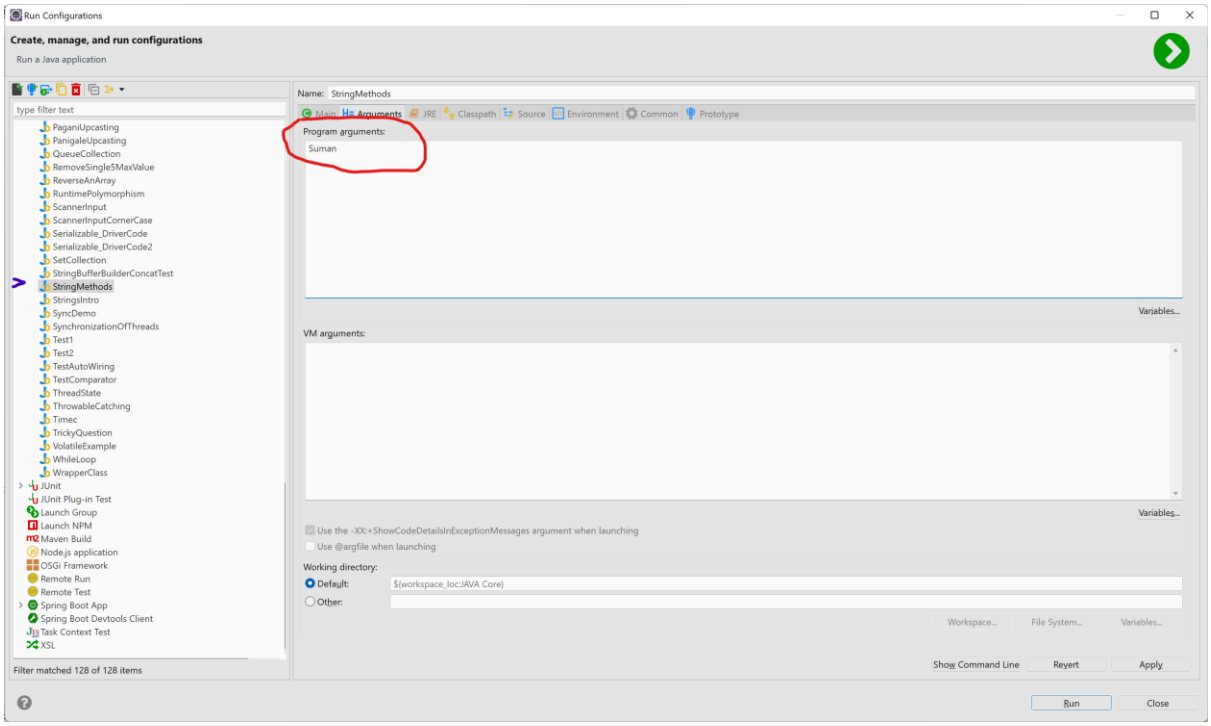
```
import java.util.Scanner;

public class ArgrsArgument {

	public static void main(String[] args) {


	//		Scanner scanner = new Scanner(System.in);
			//String input = scanner.next();
//			System.out.println(input);

String x = args[0];

			System.out.println(args.length);
			System.out.println(x);

	}

}
```

Pushing the argument using Terminal <

```
C:\Users\suman\Downloads\Y Hills\Source Code\Basics>javac ArgrsArgument.java

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java ArgrsArgument
0

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java ArgrsArgument Suman
1

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java ArgrsArgument Suman Shekhar works for
no one
6

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>javac ArgrsArgument.java

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java ArgrsArgument
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for
length 0
        at ArgrsArgument.main(ArgrsArgument.java:14)

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>java ArgrsArgument Suman
1
Suman

C:\Users\suman\Downloads\Y Hills\Source Code\Basics>
```

Using Eclipse





ArrayClass Methods

ArraysClassMethods.java

```
package com.jdk8.arrays;

import java.util.Arrays;

public class ArraysClassMethods {

	public static void main(String[] args) {

		int x[] = { 4, 6, 7, 2, 8, 0 };
		System.out.println(Arrays.toString(x));

		Arrays.sort(x);
		System.out.println(Arrays.toString(x));

		int y[] = x;

		if (y == x)
			System.out.println("Equal1");

		int z[] = { 4, 6, 7, 2, 8, 0 };

		if (z == x)// unequal
			System.out.println("Equal2");

	}
}
```

Strings in JAVA

String is basically an object that represents sequence of char values. An array

of characters works same as Java string.

Strings are a part of java.lang.String and we all know tat java.lang is imported by default, So no need to import explicitly.

Strings are basically of two types

- Mutable
- Immutable

For now we will cover Immutable class, later after Multithreading, we will cover mutable String classes.

That is tricky let me explain you in Object class equals method functionality is to match reference based on objects Sun developers had overridden the equals method in Collection, String, Wrapper classes so except this classes if you apply equals method anywhere it will check reference not content because indirectly every class in java is the child of Object class so if you want to check content in StringBuffer class you can override the object class equals method Thanks.

At the heart of it is the fact that String is immutable and StringBuffer / StringBuilder are mutable.

- If two String objects have the same characters, they will always have the same characters. So it is natural to treat them as equal ... and that is what String::equals(Object) does.
- If two StringBuffer objects can have the same characters now, and different characters a moment later ... due to a mutating operation performed by another thread. An implementation of equals(Object) for StringBuffer that was sensitive to the (changing) content would be problematic. For example:

```
if (buf1.equals(buf2)) {
// Do something to one of the StringBuffer objects.
}
```

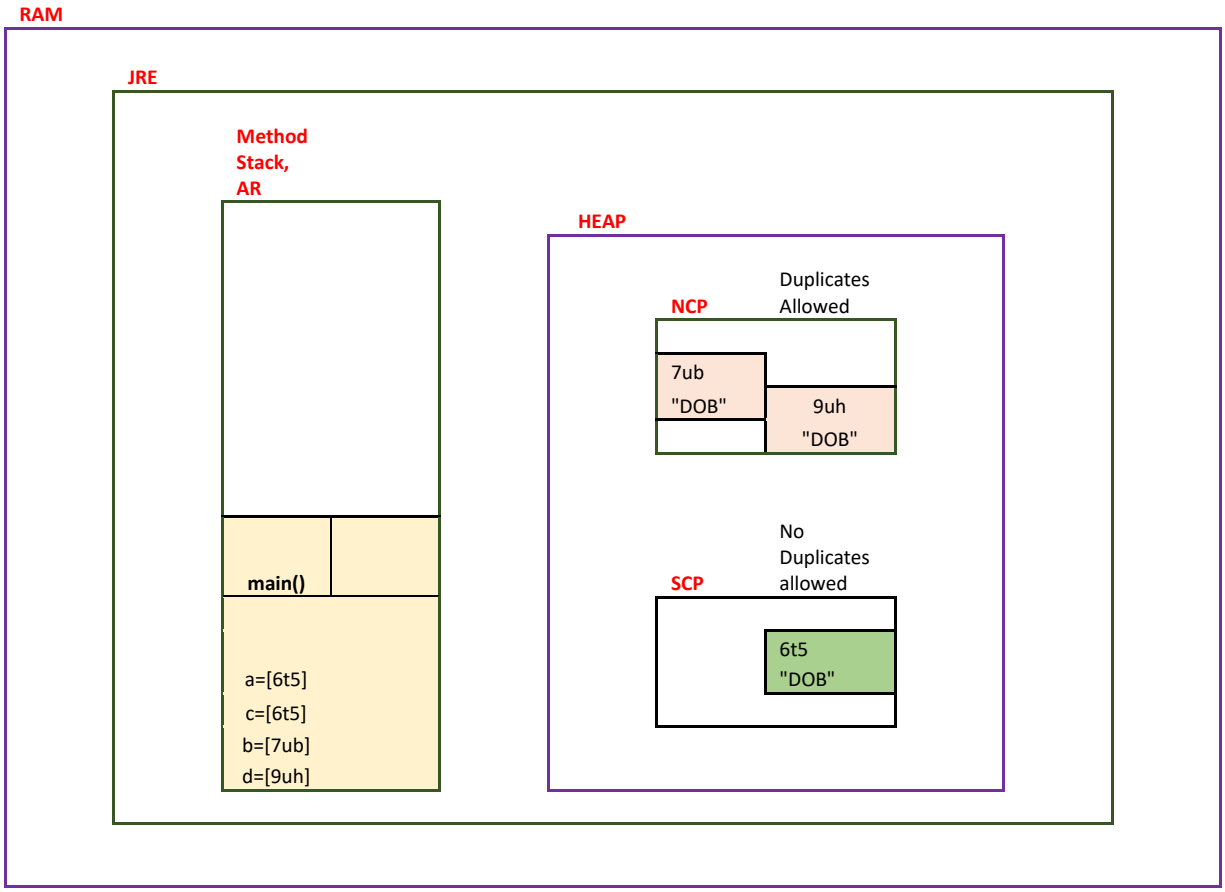
has a potential race condition. Another example is use of StringBuffer instances as keys in HashTable or HashMap.

StringsIntro.java

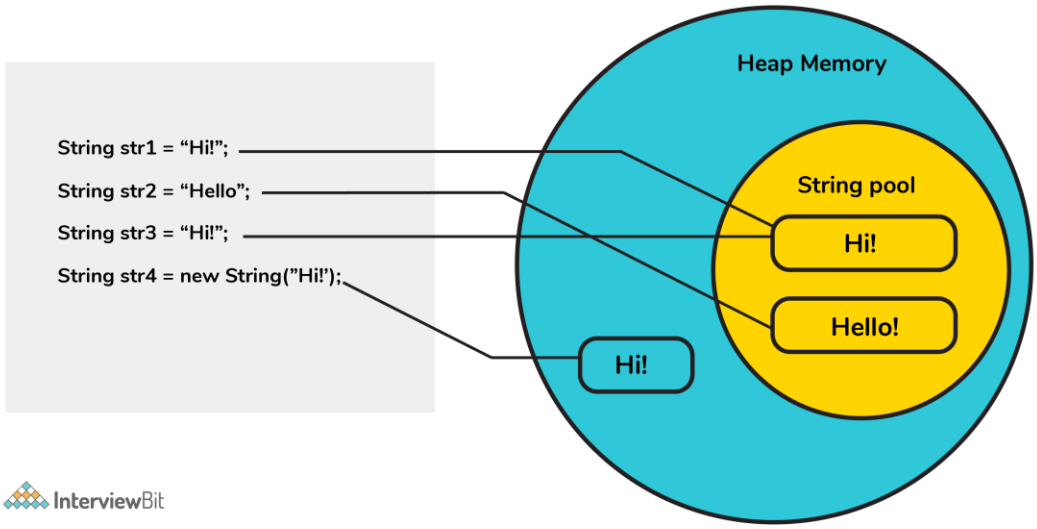
```
package com.jdk8.strings;

public class StringsIntro {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

//		String is just a sequence of characters

		char[] charArray = { 'D', 'O', 'B' };

		String a = "DOB";// String Literal
		String c = "DOB";

		String b = new String(charArray);// Declaration using new keyword
//		string="Shekhar";
		String d = new String(charArray);

		System.out.println(a);
		System.out.println(b);
		System.out.println(c);

		System.out.println(d);

		if (a == c)// == will not compare the contents of String, It will just
compare the memory
					// address
			System.out.println("SAME1");

		if (b == d)
			System.out.println("SAME2");

		if (a.equals(c))// == will not compare the contents of String, It will just
compare the memory
			// address
			System.out.println("SAME3");

		if (b.equals(d))
			System.out.println("SAME4");

	}
```

```
}
```

Memory Discussion for Strings



String Pool



Discussion on Some common methods

StringMethods.java

```
package com.jdk8.strings;

public class StringMethods {

	public static void main(String[] args) {

		String x = "YhilLs";
		String u = "yhills";

		if (x.equals(u)) {// compares contents [Case Sensitive]
			System.out.println("SAME1");
		}

		if (x.equalsIgnoreCase(u)) {// compares contents [Case Insensitive]
			System.out.println("SAME2");
		}

		System.out.println(u.length());// Returns the length
		String a = "Dora";
		String b = "Mon";
		String c = a + b;// + operator is overloaded, it can add numbers, and it can
concatenate string
		System.out.println(c);

//		c=a*b;

		int num = 6;
		boolean bool = false;

		String numString = "" + num;// convert any literal to String
		String boolString = "" + b;

		System.out.println("5" + 6 + "0");// 560
		System.out.println("5" + "6" + "0");// 560
		System.out.println("5" + 6 + 0);// 560
		System.out.println(5 + 6 + "0");// 110
		System.out.println("0" + 7 * 2);// 014
		System.out.println(0 + 7 * 2);// 14
```

```
                    System.out.println(x.contains("ih"));// checks if it contains the supplied
sample or not
                    System.out.println(x.charAt(0));// Returns character at specified index
                    System.out.println(x.indexOf("hi"));// Returns index of specified String if
found else returns -1
                    System.out.println(x.substring(2));// Used to extract a specified piece of
String
                    System.out.println(x.substring(2, 5));

          }

}
```

Day 5

OOP

Java is a **Oject Oriented Programming Language**

The four pillars of object-oriented programming are:

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

Packages

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.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

BenfitsOfPackages.java

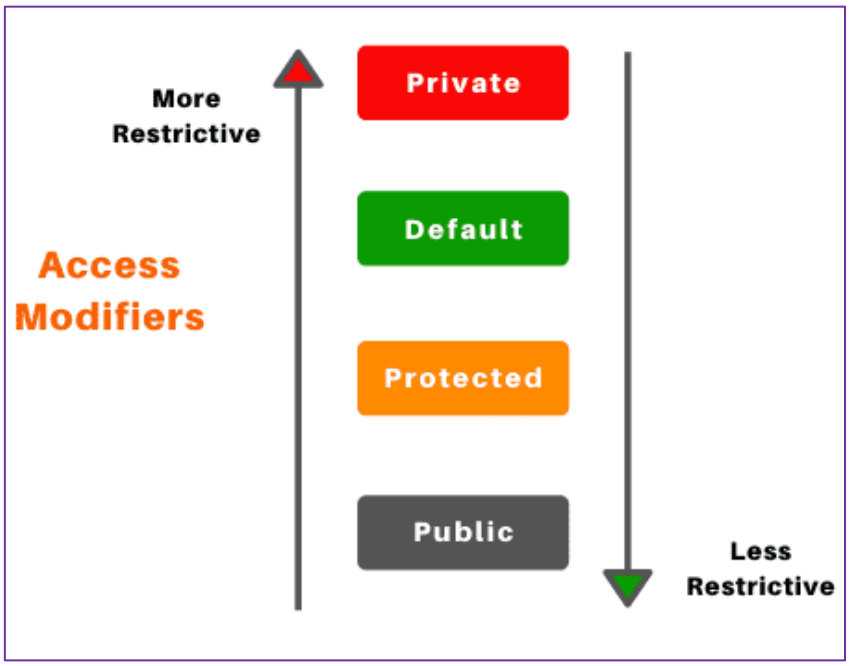
```
package com.jdk8.oop.inheritace.proper_example;

//import com.jdk8.oop.inheritace.IPLPlayer;
public class BenfitsOfPackages {

	public static void main(String[] args) {

		IPLPlayer iplPlayer = new IPLPlayer();
		iplPlayer.age = 88;
		iplPlayer.method1();

		com.jdk8.oop.inheritace.IPLPlayer iplPlayer2 = new
com.jdk8.oop.inheritace.IPLPlayer();
		iplPlayer2.BMI = 77;
		iplPlayer2.method2();
	}

}
```

Access Modifier

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

There are four types of Java access modifiers:

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



Modifier	Class	Class Variables	Methods	Method Variables
public	✓	✓	✓	
private		✓	✓	
protected		✓	✓	
default	✓	✓	✓	
final	✓	✓	✓	✓
abstract	✓		✓	
strictfp	✓		✓	
transient		✓		
synchronized			✓	
native			✓	
volatile		✓		
static		✓	✓	

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

Access Specifiers in Java

		public	private	protected	default
Same Package	Class	YES	YES	YES	YES
	Sub class	YES	NO	YES	YES
	Non sub class	YES	NO	YES	YES
Different Package	Sub class	YES	NO	YES	NO
	Non sub class	YES	NO	NO	NO

Access Specifier ▶ / Item ▼	Default	Public	Protected	Private
Class	Yes	Yes	No	No
Inner Class	Yes	Yes	Yes	Yes
Interface	Yes	Yes	No	No
Interface Inside Class	Yes	Yes	Yes	Yes
enum	Yes	Yes	No	No
enum Inside Class	Yes	Yes	Yes	Yes
enum inside Interface	Yes	No	No	No
Constructor	Yes	Yes	Yes	Yes
methods & data inside class	Yes	Yes	Yes	Yes
methods & data inside Interface	Yes	No	No	No

Access control for members of class and interface in java

Accessibility / Location / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Non Encapsulated Class

Bird.java

```
public class Bird {

	String species;
	String color;
	boolean isEndangered;

	@Override
	public String toString() {
		return "Bird [species=" + species + ", color=" + color + ", isEndangered=" +
isEndangered + "]";
	}

}
```

Driver.java

```
package com.jdk8.oop.encapsulation;

public class Driver {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Bird bird1 = new Bird();
		bird1.color = "blue";
		bird1.isEndangered = false;
		bird1.species = "Eagles";

		System.out.println(bird1);
		bird1.isEndangered=true;
		System.out.println(bird1);

	}

}
```

Encapsulated Class

Bird2.java

```
package com.jdk8.oop.encapsulation;

public class Bird2 {

	private String species;// private is keyword and used as access specifier
	private String color;
	private boolean isEndangered;

	@Override
	public String toString() {
		chirp();
		return "Bird [species=" + species + ", color=" + color + ", isEndangered=" +
isEndangered + "]";
	}

	public String getSpecies() {
		return species;
	}

	public void setSpecies(String s) {
		this.species = species;// Overshadow
	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	public boolean isEndangered() {
		return isEndangered;
	}

	public void setEndangered(boolean isEndangered) {
		this.isEndangered = isEndangered;
	}

	private void chirp() {
		//methods can also be private, SO it will have nmo outside acceess
		System.out.println("CHIRP CHIRPCHIRP CHIRP CHIRP @@@ ");
	}

}
```

Driver2.java

```
package com.jdk8.oop.encapsulation;

public class Driver2 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Bird2 bird1 = new Bird2();

//		bird1.color = "blue";//Direct access will give error for private variables
//		bird1.isEndangered = false;
//		bird1.species = "Eagles";

		bird1.setColor("Blue");
		bird1.setEndangered(false);
		bird1.setSpecies("Eagles");

		System.out.println(bird1);
		System.out.println(bird1.getColor());

//		bird1.chirp();

	}
}
```

Constructors

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

- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules 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

There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor

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

This Keyword

6 usage of java **this** keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

Animal.java

```
package com.jdk8.oop.constructors;

public class Animal {

	private String species;
	private int noAlive;
	private boolean isEndemic;

	@Override
	public String toString() {
		return "Animal [species=" + species + ", noAlive=" + noAlive + ",
isEndemic=" + isEndemic + "]";
	}

	public String getSpecies() {
		return species;
	}

	public void setSpecies(String species) {
		this.species = species;
	}

	public int getNoAlive() {
		return noAlive;
	}

	public void setNoAlive(int noAlive) {
		this.noAlive = noAlive;
	}

	public boolean isEndemic() {
		return isEndemic;
	}

	public void setEndemic(boolean isEndemic) {
		this.isEndemic = isEndemic;
	}

	public Animal(String species, int noAlive, boolean isEndemic) {// Parameterized
Constructor

//		Constructor has the same name as class
//		Constructor does not has an explicit return type
//		Constructors can be parameterized or parameterless
//		A class can have several constructors

		this();// Used for Constructor Chaining, and it should be the 1st statement
//		super();
		this.species = species;
		this.noAlive = noAlive;
		this.isEndemic = isEndemic;
//		this();
	}

	public Animal() {// Default / No Parameter Constructor
		System.out.println("Animal are lovely");

	}
}
```

Driver.java

```
package com.jdk8.oop.constructors;

public class Driver {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Animal animal1 = new Animal();
//		animal1.setSpecies("Lion");
//		animal1.setEndemic(false);
//		animal1.setNoAlive(5000);
		System.out.println(animal1);

		Animal animal2 = new Animal("Buffalo", 6000000, false);
		System.out.println(animal2);

	}

}
```

Constructor can also be private , it serves a special purpose. ASSIGNMENT

Box.java

```
package com.jdk8.oop.constructors;

public class Box {//Object Class

	float lenght;
	float height;
	float breadth;
	String material;


	private Box(float lenght, float height, float breadth, String material) {//Explicit
Constructor
//		super();
		this.lenght = lenght;
		this.height = height;
		this.breadth = breadth;
		this.material = material;
	}

	//What is the use of a private Constructor ?
}
```

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability.

RULES

1. Multiple Inheritance is not possible in JAVA
   ```
   public class IPLPlayer extends Player,SportsMan {
   ```
2. Multi Level Inheritance is allowed
3. Method Overriding
4. Cannot decrease the visibility of the inherirted methods
5. Private methods cannot be overrided
6. Priavte instance variabkles cannot be accessed in the child class

PARENT

Player.java

```
public class Player {

          String name;
          int age;
          int BMI;


}
```

CHILD

IPLPlayer.java

```
package com.jdk8.oop.inheritace.proper_example;

public class IPLPlayer extends Player {

	int matchesPlayed;
	int StrikeRate;

	@Override
	public String toString() {
		return "IPLPlayer [matchesPlayed=" + matchesPlayed + ", StrikeRate=" +
StrikeRate + ", name=" + name + ", age="
					+ age + ", BMI=" + BMI + "]";
	}



}
```

CHILD

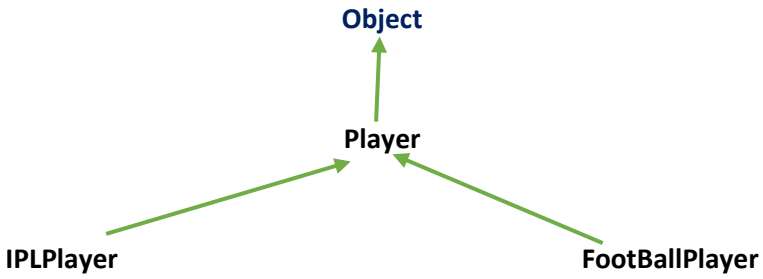
FootBallPlayer.java

```
package com.jdk8.oop.inheritace.proper_example;

public class FootBallPlayer extends Player{

	int noOfGoals;
	int noOfAssisted;


	@Override
	public String toString() {
		return "FootBallPlayer [noOfGoals=" + noOfGoals + ", noOfAssisted=" +
noOfAssisted + ", name=" + name + ", age="
					+ age + ", BMI=" + BMI + "]";
	}



}
```

Tester3.java

```
package com.jdk8.oop.inheritace.proper_example;

public class Tester3 {

	public static void main(String[] args) {

//			Football
			FootBallPlayer[] barcelona = new FootBallPlayer[15];
			FootBallPlayer messi = new FootBallPlayer();
			messi.name = "Messi";
			messi.age = 47;
			System.out.println(messi);
			barcelona[0] = messi;

//			IPL Team
			IPLPlayer[] RCB = new IPLPlayer[15];
			IPLPlayer virat = new IPLPlayer();
			virat.name = "Virat Kohli";
			virat.BMI = 50;
			virat.matchesPlayed = 99;
			virat.StrikeRate = 96;
			virat.age = 45;
			System.out.println(virat);

			RCB[0] = virat;
	}

}
```

Inheritance Diagram



Object Class

The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

The Object class provides many methods. They are as follows:

Methods of Object class

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.

protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Day 6

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

MethodOverloading

Same method name but different functionality.

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

Rules

- No of parameter
- Type of paramerter
- Position of paramerter

MathOperations.java

```
package com.jdk8.oop.polymorphism;

import static java.lang.Math.PI;

public class MathOperations {

	double area(float radius) {
//		return Math.PI * radius * radius;
		return PI * radius * radius;
	}

	double area(int length) {
		return length * length;
	}

	double area(float length, float breadth) {
		return length * breadth;
	}

}
```

Driver.java

```
package com.jdk8.oop.polymorphism;

public class Driver {

	public static void main(String[] args) {
//		Method Overloading
		MathOperations mathOperations = new MathOperations();
		System.out.println(mathOperations.area(10.0f));
		System.out.println(mathOperations.area(10));
		System.out.println(mathOperations.area(10, 8));

	}

}
```

Method Overriding

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

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

Usage of Java Method Overriding

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

Rules for Java Method Overriding

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

A static method cannot be overridden.

Player.java

```
package com.jdk8.oop.inheritace.proper_example;

public class Player {

		String name;
		int age;
		int BMI;
		private long phoneNumber;

		public void stats() {
			System.out.println("STATS --> [ " + name + " " + age + " " + BMI+"]");
		}

		public Player() {
			System.out.println("Constrcutor Triggered of Parent Class --> Player");
		}

}
```

IPLPlayer.java

```
package com.jdk8.oop.inheritace.proper_example;

public class IPLPlayer extends Player {

		int matchesPlayed;
		int StrikeRate;

		@Override
		public String toString() {
			return "IPLPlayer [matchesPlayed=" + matchesPlayed + ", StrikeRate=" + StrikeRate
+ ", name=" + name + ", age="
					+ age + ", BMI=" + BMI + "]";
		}

		void method1() {
			System.out.println("IPLPlayer Class using Inheitance");
		}

		public IPLPlayer() {
			super();
			System.out.println("IPL Player Constructor Triggered");
		}

		@Override
		public void stats() {
			super.stats();
			System.out.println("STRIKE RATE ->" + StrikeRate);
		}

}
```

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Tester3.java

```
package com.jdk8.oop.inheritace.proper_example;

public class Tester3 {

	public static void main(String[] args) {

//			Football
			FootBallPlayer[] barcelona = new FootBallPlayer[15];
			FootBallPlayer messi = new FootBallPlayer();
			messi.name = "Messi";
			messi.age = 47;
			System.out.println(messi);
			barcelona[0] = messi;

//			IPL Team
			IPLPlayer[] RCB = new IPLPlayer[15];
			IPLPlayer virat = new IPLPlayer();
			virat.name = "Virat Kohli";
			virat.BMI = 50;
			virat.matchesPlayed = 99;
			virat.StrikeRate = 96;
			virat.age = 45;
//			System.out.println(virat);
			virat.stats();

			RCB[0] = virat;
	}

}
```

Static Keyword

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

Java static property is shared to all objects of the same class.

Static Variable

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

The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

The static variable gets memory only once in the class area at the time of class loading.

The static can be applied to:

- Variable (also known as a class variable)
- Method (also known as a class method)
- Block
- Nested class

Advantages of static variable

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

Employee.java

```
package com.jdk8.oop.static_examples;

public class Employee {

	String name;
	private long phoneNumber;
	private boolean isMarried;
	String employeeID;
	static int sequence = 1;

	public Employee(String name, long phoneNumber, boolean isMarried, String employeeID) {
		super();
		System.out.println("Employee constructor triggerd");
		this.name = name;
		this.phoneNumber = phoneNumber;
		this.isMarried = isMarried;
		this.employeeID = "EMP" + sequence++;
	}

	public Employee() {
		super();
		// TODO Auto-generated constructor stub
	}

	@Override
	public String toString() {
		return "Employee [name=" + name + ", phoneNumber=" + phoneNumber + ",
isMarried=" + isMarried + ", employeeID="
					+ employeeID + "]";
	}

	static int getCount() {
//		System.out.println(phoneNumber);
		return sequence;
	}

	static {
		sequence = 100;
		System.out.println("Static block called");
	}

}
```

Driver.java

```
package com.jdk8.oop.static_examples;

public class Driver {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Employee emp1 = new Employee("Sam", 98597854, false, "");
		Employee emp2 = new Employee("Amy", 98556854, false, "");
		Employee emp3 = new Employee("John", 945497854, false, "");

		System.out.println(emp1.sequence);
		System.out.println(emp2.sequence);
		System.out.println(emp3.sequence);
		System.out.println(Employee.sequence);//

		System.out.println(emp1);
		System.out.println(emp2);
		System.out.println(emp3);

		System.out.println(Employee.getCount());//



	}

}
```

Final Keyword

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

- class
- variable
- method

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

- A final class cannot be inherited.
- A final variable cannot re initialised
- A final method can be inherited cannot be overridden
- A blank final variable can only b einitialised inside constructor.

FianlKeywordExample.java

```
package com.jdk8.oop.final_keyword_examples;

public final class FianlKeywordExample {
	final static float pi = 3.14179f;

	public static void main(String[] args) {

//			pi=10;

			final int x;
			x=56;
//			x=55;


			float radius = 10;
			System.out.println("AERA --> " + calcAreaOfCircle(radius));

	}

	private static float calcAreaOfCircle(float radius) {
			return pi * radius * radius;
	}

}
```

UseCase.java

```
package com.jdk8.oop.final_keyword_examples;

//public class UseCase extends FianlKeywordExample{
public class UseCase {

	final int x;// Only place where uninitialized final instance variable can exist with
the
					// help of declaration in constructor

	public UseCase(int x) {
			super();
			this.x = x;
	}

}
```

Super Keyword

The super keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

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

Player.java

```
package com.jdk8.oop.inheritace.proper_example;

public class Player {

	String name;
	int age;
	int BMI;
	private long phoneNumber;

	public void stats() {
		System.out.println("STATS --> [ " + name + " " + age + " " + BMI+"]");
	}

	public Player() {
		System.out.println("Constrcutor Triggered of Parent Class --> Player");
	}

}
```

IPLPlayer.java

```
package com.jdk8.oop.inheritace.proper_example;

public class IPLPlayer extends Player {

	int matchesPlayed;
	int StrikeRate;

	@Override
	public String toString() {
		return "IPLPlayer [matchesPlayed=" + matchesPlayed + ", StrikeRate=" +
StrikeRate + ", name=" + name + ", age="
					+ age + ", BMI=" + BMI + "]";
	}

	void method1() {
		System.out.println("IPLPlayer Class using Inheitance");
	}


	public IPLPlayer() {
		super();
		System.out.println("IPL Player Constructor Triggered");
	}

	@Override
	public void stats() {
		super.stats();
		System.out.println("STRIKE RATE ->" + StrikeRate);
	}

}
```

This Keyword

In Java, this is a reference variable that refers to the current object.

- The **this keyword** can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity[Removes Shadow Problem].
- this() can be used to invoke current class constructor and perform constructor chaining.
- this keyword can be used to return current class instance.

Animal.java

```
package com.jdk8.oop.constructors;

public class Animal {

		private String species;
		private int noAlive;
		private boolean isEndemic;

		@Override
		public String toString() {
			return "Animal [species=" + species + ", noAlive=" + noAlive + ", isEndemic=" +
isEndemic + "]";
		}

		public String getSpecies() {
			return species;
		}

		public void setSpecies(String species) {
			this.species = species;
		}

		public int getNoAlive() {
			return noAlive;
		}

		public void setNoAlive(int noAlive) {
			this.noAlive = noAlive;
		}

		public boolean isEndemic() {
			return isEndemic;
		}

		public void setEndemic(boolean isEndemic) {
			this.isEndemic = isEndemic;
		}

		public Animal(String species, int noAlive, boolean isEndemic) {// Parameterized Constructor

//			Constructor has the same name as class
//			Constructor does not has an explicit return type
//			Constructors can be parameterized or parameterless
//			A class can have several constructors

			this();// Used for Constructor Chaining, and it should be the 1st statement
//			super();
			this.species = species;
			this.noAlive = noAlive;
			this.isEndemic = isEndemic;
//			this();
		}

		public Animal() {// Default / No Parameter Constructor
			System.out.println("Animal are lovely");

		}
}
```

Abstraction

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 us 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

- Abstract class (0 to 100%)
- Interface (100%)

Abstract Class

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).

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

Animal.java

```
package com.jdk8.abstractclass;

public abstract class Animal {
//        0-100% Abstraction using Abstract Class
          boolean isExtinct;// Instance Variable

          final void breathe() {// Concrete/Non Abstract Methods/ Non Overidable due to final
                    System.out.println("Every Animal Breathe");
          }

          abstract void speaks();// Abstract Method

}
```

Cow.java

```
package com.jdk8.abstractclass;

public class Cow extends Animal {

	@Override
	void speaks() {// Overrided/Concrete Method
		System.out.println("Cows make sound like MOO~~~~");
	}

	void color() {// Specialized Methods, Concrete Method
		System.out.println("Cows are BROWN, BLACK, WHITE or mix");

	}

}
```

Lion.java

```
package com.jdk8.abstractclass;

public class Lion extends Animal {// extend Inheritance , you can ONLY extend one class

	@Override
	void speaks() {
		System.out.println("LION ROARS LOUDER");

	}

//	@Override
//	void breathe() {
//		System.out.println("Lions breath fatser");
//	}

	void king() {// Specialized Method
		System.out.println("LION IS THE KING OF JUNGLE");
	}

}
```

Tiger.java

```
package com.jdk8.abstractclass;

public class Tiger extends Animal {

	@Override
	void speaks() {// Overrided
		System.out.println("Tiger ROARS####");

	}

	void species() {// Specialized
		System.out.println("The most prestigious species of tigers are Sundarban
Tigers");
	}

}
```

Driver.java

```
package com.jdk8.abstractclass;

public class Driver {

          public static void main(String[] args) {

//                  Animal animal = new Animal();//abstract class cannot be normally
instantiated
                    Cow cow = new Cow();
                    cow.speaks();
                    cow.breathe();// Inherited
                    cow.color();// Specialized

                    Tiger tiger = new Tiger();
                    tiger.breathe();
                    tiger.speaks();// Overided
                    tiger.species();// Specialized

                    // Up casting- Assigning a Parent reference to the CHild Object
                    Animal cowAnimal = new Cow();
                    cowAnimal.breathe();
                    cowAnimal.speaks();

                    // Down casting- DownCasting an already upcasted object
//                  Upcasting and Downcasting does not create new Objects !!!
                    Cow cow3 = (Cow) cowAnimal;// Downcasting
                    // It is ONLY possible when we are down casting an already upcasted objects\

                    Lion lion = new Lion();
                    lion.breathe();
                    lion.speaks();
                    lion.king();

          }

}
```

Interface

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

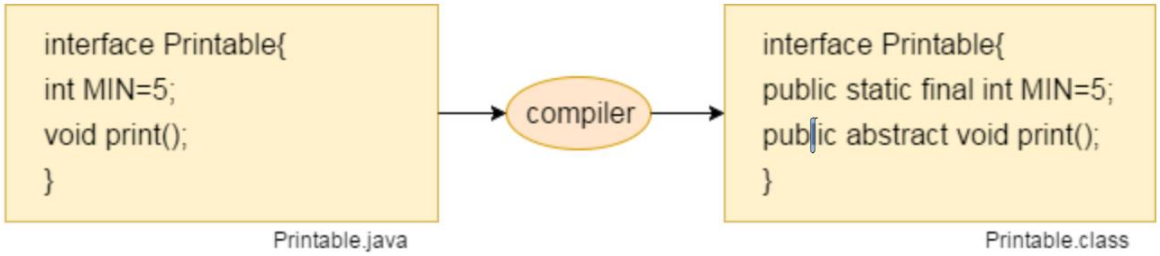
The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. Though JDK 8 allows to add static and default methods.

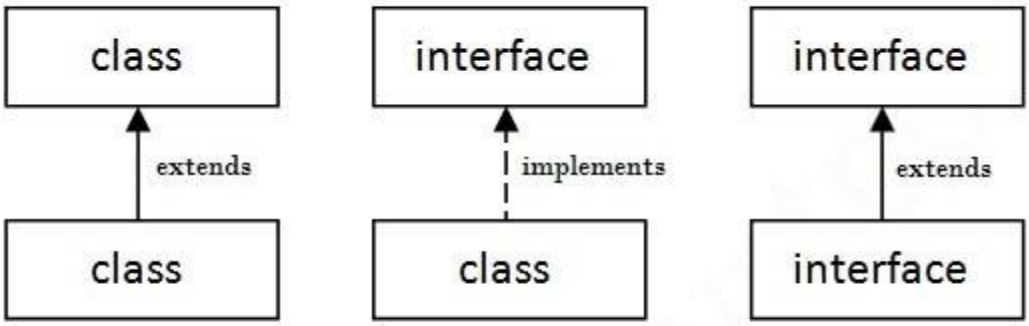
- Java Interface also represents the **IS-A relationship.**
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have default and static methods in an interface.
- Since Java 9, we can have private methods in an interface.

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.

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** implicitly. A class that implements an interface must implement all the methods declared in the interface.





*Functional Interface is an interface with just a **single** method.We can use Lambda function on Functional interfaces only.*

Pure Virtual Function

A virtual function for which we are not required implementation is considered as pure virtual function. For example, Abstract method in Java is a pure virtual function.

Animal.java

```
package com.jdk8.interfaces;

public interface Animal extends CarbonFootprint {
          // JDK ......1.6,1.7,1.8,9,10,.......18
          // Interface can have static methods also since JDK 8
          // Interface can also have default methods since JDK 8
          // Generally methods are implicitly abstract in Interfaces
```

```
	boolean isInstinct = false;// Every Variable in final & static

	void breathe();// Abstract methods | abstrac keywords are optional

	int count();

}
```

CarbonFootprint.java

```
package com.jdk8.interfaces;

@FunctionalInterface
public interface CarbonFootprint {
//iG A INETRFACE IS HAVING JUST A SINGLE ABSTRACT METHODC-> TYHAT INTERFCACE IS CALLED
FUNCTIONAL INTERFACE
	 boolean isInstinct=true;
	boolean isResponsibleForGlobalWarming();

	static void earthCrying() {
		System.out.println("We humans are destroying teh earth, we shiu;ld preservde
it");
	}

}
```

Survey.java

```
package com.jdk8.interfaces;

public class Survey {

	boolean isSurveyed;

}
```

Dog.java

```
package com.jdk8.interfaces;

public class Dog extends Survey implements Animal, CarbonFootprint {// extends keyword will be
always before implements

                                                                        // keyword

	@Override
	public void breathe() {
		System.out.println("Dog breathes faster while running");

	}

	@Override
	public int count() {
		return Integer.MAX_VALUE;
	}

	@Override
	public boolean isResponsibleForGlobalWarming() {
		return false;
	}
```

```
}
```

Driver.java

```
package com.jdk8.interfaces;

public class Driver {

	public static void main(String[] args) {

//		Animal an =		new Animal();//Normally we cannot instantiate an interface
			Animal animal = new Animal() {

				@Override
				public int count() {

					return 10;
				}

				@Override
				public void breathe() {
					System.out.println("Test Breathing");

				}

				@Override
				public boolean isResponsibleForGlobalWarming() {
					// TODO Auto-generated method stub
					return false;
				}
			};// But if we define the body then and there only , then we can instantiate

			int count = animal.count();
			animal.breathe();
			System.out.println(count);
			System.out.println();

			Dog dog = new Dog();
			dog.breathe();
//			System.out.println(dog.isInstinct);
//			System.out.println( Dog.isInstinct);//Multiple implementation can be cause
ambiguity and conflict so care must be taken

			// To remove ambiguity use it Statically from the source
			System.out.println(Animal.isInstinct);
			System.out.println(CarbonFootprint.isInstinct);

			// Static methods dont take part in Inheritance
			CarbonFootprint.earthCrying();

//			Using Lambda over an Functional interface
			CarbonFootprint cft = () -> {
				return true;
			};
			System.out.println(cft.isResponsibleForGlobalWarming());

	}

}
```

Differnce between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

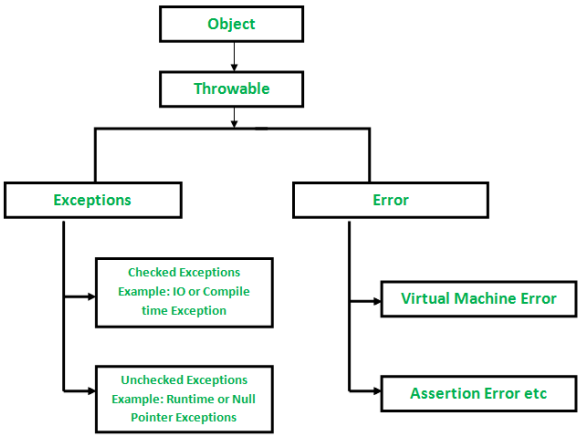
But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also .
2) Abstract class **doesn't support multiple inheritance** .	Interface **supports multiple inheritance** .
3) Abstract class **can have final, non-final, static and non-static variables** .	Interface has **only static and final variables** .
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only .
7) An **abstract class** can be extended using keyword "extends" .	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
`public abstract class Shape{`
`public abstract void draw();`
`}`
	Example:
	`public interface Drawable{`
	`void draw();`
	`}`

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Exception Handling

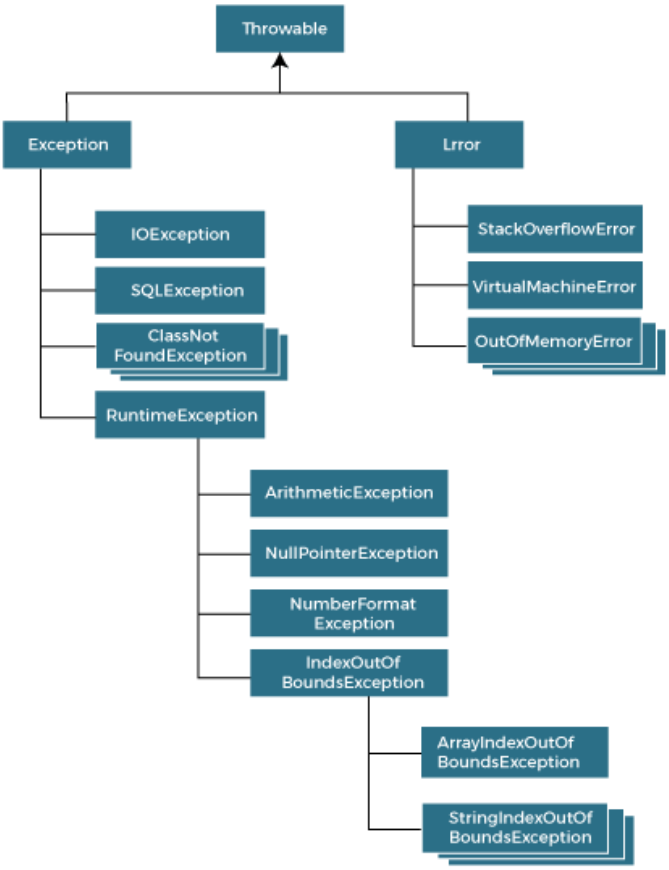


The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

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

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

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.



Checked Exception

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

Unchecked Exception

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

Error

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

Exception Basic

ExceptionExample.java

```
package com.jdk8.oop.exceptions;

public class ExceptionExample {

	public static void main(String[] args) {
		System.out.println("START");
		float x = (float) (9 / 0.0);// Infinity

		// Unchecked Exceptions
		System.out.println(6 / 0);// Arithmetic Exception

		int[] fibonachi = { 1, 1, 2, 3, 5 };// ArrayIndexOutOfBoundsException:
		for (int i = 0; i <= fibonachi.length; i++) {
			System.out.println(fibonachi[i]);
		}

		System.out.println("END");

	}
}
```

Exception Handling Basic

ExceptionHandlingBasics.java

```
package com.jdk8.oop.exceptions;

public class ExceptionHandlingBasics {

	public static void main(String[] args) {

		System.out.println("START");

//		NumberFormatException
// Arithmetic Exception
		int y = 9;
		String z = "" + y;
		try {
//			int x = 6 / 0;// Unchecked
			String x = "6O";
			int parseInt = Integer.parseInt(x);
			System.out.println(parseInt);
		} catch (ArithmeticException e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		} catch (Exception e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		}

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

ExceptionHandlingBasics2.java

```
package com.jdk8.oop.exceptions;

public class ExceptionHandlingBasics2 {

	public static void main(String[] args) {

		System.out.println("START");

		int[] x = { 2, 6, 7, 8 };

		try {
			x[4] = 8 / 0;
		} catch (ArithmeticException e) {// It will be caught here because
evaluation happens from left to right
			System.out.println(e.getMessage());
			e.printStackTrace();
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		} catch (Exception e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		}

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

CheckedExceptionExample.java

```
package com.jdk8.oop.exceptions;

public class CheckedExceptionExample {

	public static void main(String[] args) {

		for (int i = 0; i < 10; i++) {
			try {
				Thread.sleep(500);// Will be checked at compile time , so
must to handle this
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println("HELLO");
		}

	}

}
```

MultipleCatchBlock3.java

```
package com.jdk8.oop.exceptions;

public class MultipleCatchBlock3 {

	public static void main(String[] args) {

		try {
			int a[] = new int[5];
			a[6] = 30 / 0;
			System.out.println(a[10]);
		} catch (ArithmeticException e) {
			System.out.println("Arithmetic Exception occurs");
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("ArrayIndexOutOfBounds Exception occurs");
		} catch (Exception e) {
			System.out.println("Parent Exception occurs");
		}
		System.out.println("END");
	}
}
```

This much combo can exist

- **try catch …**
- **try catch … finally**
- **try finally**

1. If multiple catch blocks, the ExceptionClasses should be child to parent as in top to bottom.
2. Exception class should be at last if added.
3. Its not recommended to catch throwable Exception, as it may catch **Errors**

Java finally block

- Java **finally** block is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- The finally block follows the try-catch block.

TryCatchFinally.java

```
package com.jdk8.oop.exceptions;

import java.util.Scanner;

public class TryCatchFinally {

	public static void main(String[] args) {
		System.out.println("START");
		Scanner scanner = new Scanner(System.in);
		System.out.println("Enter you age");
		int age = scanner.nextInt();
		int m;
		try {
			int y = 100 / age;
			System.out.println(y);
		} catch (Exception e) {
			System.exit(1);
			m = 7 / 0;
			e.printStackTrace();
		} finally {
			scanner.close();
			System.out.println("CLOSED");
		}

		System.out.println("END");

	}

}
```

TryFinally.java

```
package com.jdk8.oop.exceptions;

public class TryFinally {

	public static void main(String[] args) {

		try {
			int x = 7 / 0;
		} finally {
			System.out.println("Finally Exceuted");
		}

	}
}
```

Try with Resource

In Java, the Try-with-resources statement is a try statement that declares one or more resources in it. A resource is an object that must be closed once your program is done using it. For example, a File resource or a Socket connection resource. The try-with-resources statement ensures that each resource is closed at the end of the statement execution. If we don't close the resources, it may constitute a resource leak and also the program could exhaust the resources available to it.

You can pass any object as a resource that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable.

By this, now we don't need to add an extra finally block for just passing the closing statements of the resources. The resources will be closed as soon as the try-catch block is executed.

Syntax: Try-with-resources

```
try(declare resources here) {
    // use resources
}
catch(FileNotFoundException e) {
    // exception handling
}
```

When it comes to exceptions, there is a difference in try-catch-finally block and try-with-resources block. If an exception is thrown in both try block and finally block, the method returns the exception thrown in finally block.

For try-with-resources, if an exception is thrown in a try block and in a try-with-resources statement, then the method returns the exception thrown in the try block. The exceptions thrown by try-with-resources are suppressed, i.e. we can say that try-with-resources block throws suppressed exceptions.

TryWithResource.java

```
package com.geekster.exceptions;

import java.io.FileOutputStream;
import java.util.Scanner;

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

		Scanner sc;

		// Creating an object of FileOutputStream to write stream or raw data
		try (FileOutputStream fos = new FileOutputStream("docs/sample.txt");
//				sc = new Scanner(System.in);
				Scanner scan = new Scanner(System.in);) {// Adding
resource
			System.out.println("Give the input to write");
			String text = "GeekSter took input" + scan.nextLine();
			byte arr[] = text.getBytes();
			fos.write(arr);
		} catch (Exception e) {
			System.out.println(e);
		}

		System.out.println("END");

	}

}
```

Custom Exceptions | throw | throws

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

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

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

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

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

Following is few of the reasons to use custom exceptions:

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

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

VotingEligiblity.java

```
package com.jdk8.oop.exceptions;

import java.util.Scanner;

public class VotingEligiblity {

	public static void main(String[] args) {

		System.out.println("START");
		Scanner scanner = new Scanner(System.in);
		System.out.println("Enter you age");
		int age = scanner.nextInt();

		try {
			validate(age);
		} catch (InvalidAgeException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} finally {
			scanner.close();
		}
		System.out.println("END");

	}

	private static void validate(int age) throws InvalidAgeException {

		if (age >= 18)
			System.out.println("ELIGIBLE");
		else
			throw new InvalidAgeException("Invalid Age for Voting");// Creating
an Exception

	}
}

class InvalidAgeException extends Exception {

	private static final long serialVersionUID = 1L;

	InvalidAgeException(String msg) {
		super(msg);
	}
}
```

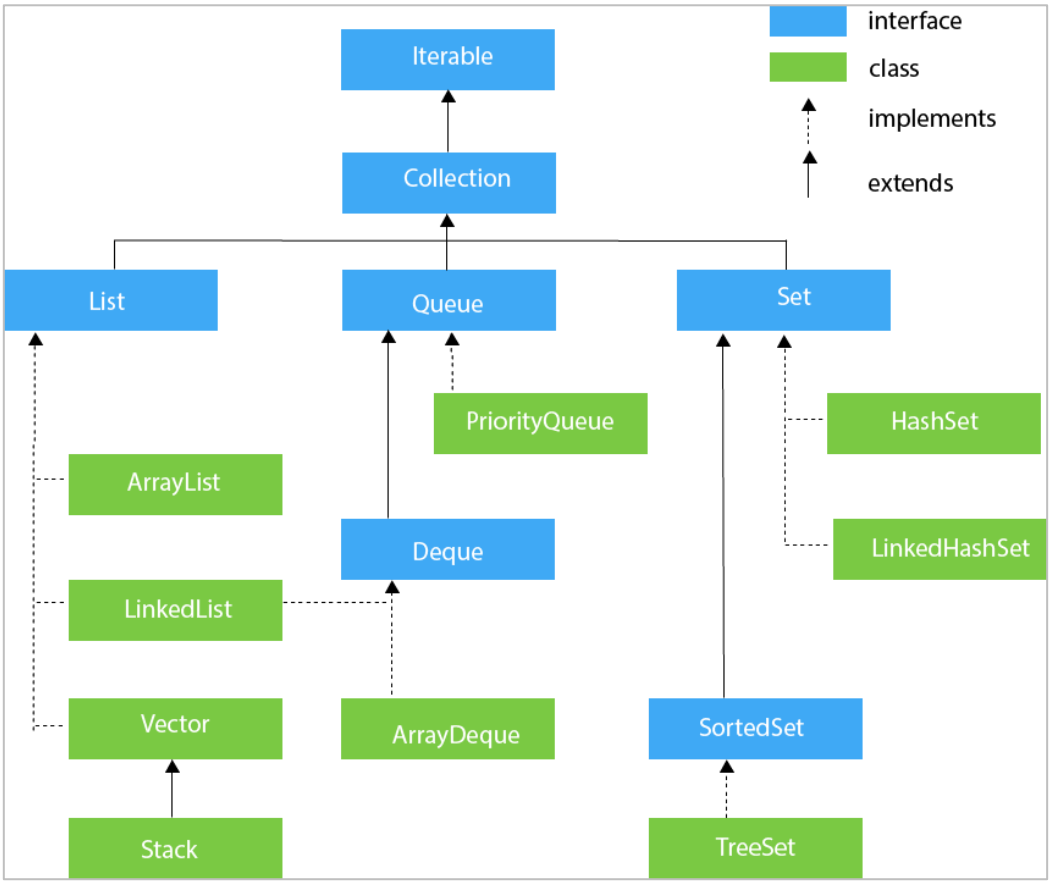
Day 7

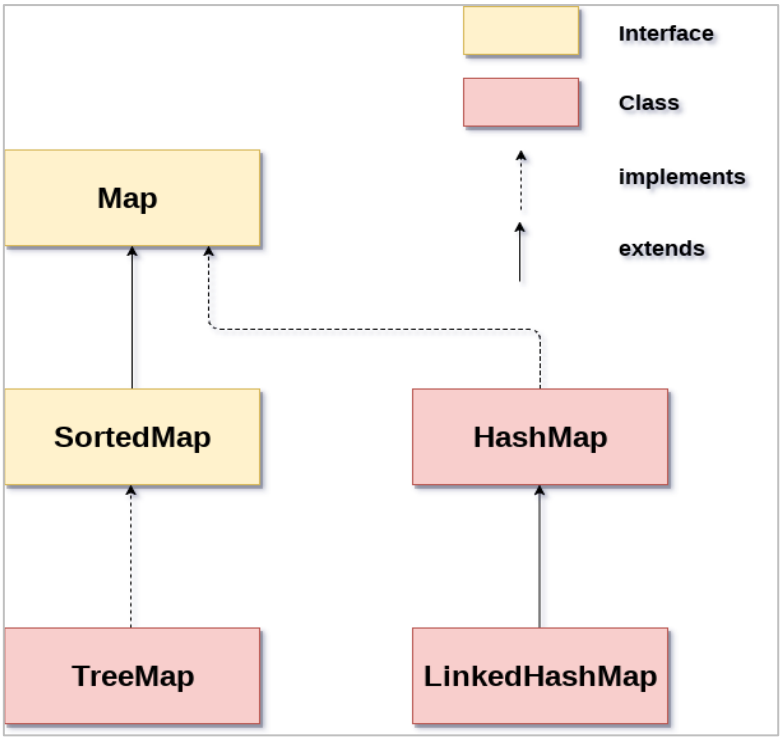
Collections Overview

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

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

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





Generics

Generics means parameterized types. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics?

The Object is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature.

Bird2.java

```
package com.jdk8.oop.encapsulation;

public class Bird2 {

	private int number;
	public int getNumber() {
		return number;
	}

	public void setNumber(int number) {
		this.number = number;
	}

	private String species;// private is keyword and used as access specifier
	private String color;
	private boolean isEndangered;



	@Override
	public String toString() {
		return "Bird2 [number=" + number + ", species=" + species + ", color=" +
color + ", isEndangered="
					+ isEndangered + "]";
	}

	public String getSpecies() {
		return species;
	}

	public void setSpecies(String s) {
		this.species = species;// Overshadow
	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	public boolean isEndangered() {
		return isEndangered;
	}

	public Bird2(int number, String species, String color, boolean isEndangered) {
		super();
		this.number = number;
		this.species = species;
		this.color = color;
		this.isEndangered = isEndangered;
	}

	public void setEndangered(boolean isEndangered) {
		this.isEndangered = isEndangered;
	}

	private void chirp() {
		//methods can also be private, SO it will have nmo outside acceess
		System.out.println("CHIRP CHIRPCHIRP CHIRP CHIRP @@@ ");
	}

}
```

GericsExample.java

```
package com.jdk8.coolections;

import java.util.ArrayList;
import java.util.Arrays;

import com.jdk8.oop.encapsulation.Bird2;

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

		int[] x = { 5, 7, 7 };
		System.out.println(Arrays.toString(x));// Inheriting

		ArrayList arrayList = new ArrayList();
		arrayList.add("Suman");
		arrayList.add(1);
		arrayList.add(true);
		arrayList.add(new Bird2(0, "Toucan", "Colorfull", true));

		ArrayList<Bird2> arrayListWithGenerics = new ArrayList<Bird2>();
		arrayListWithGenerics.add(new Bird2(1, "Parrot", "Green", false));
//		arrayListWithGenerics.add(1);//this will give error as we are using Generics

	}

}
```

ArrayList

Properties

1. Java ArrayList class can contain duplicate elements.
2. Java ArrayList class maintains insertion order.
3. Java ArrayList class is non synchronized.
4. Java ArrayList allows random access because the array works on an index basis.
5. In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
6. We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases

- Java ArrayList class uses a dynamic array for storing the elements.
- ArrayList expands dynamically, ensuring that there is always room for more elements to be added. An array of the Object class serves as the ArrayList's underpinning data structure.
- `private static final int DEFAULT_CAPACITY = 10;`
- Object Array in ArrayList is transient.
- Constructors

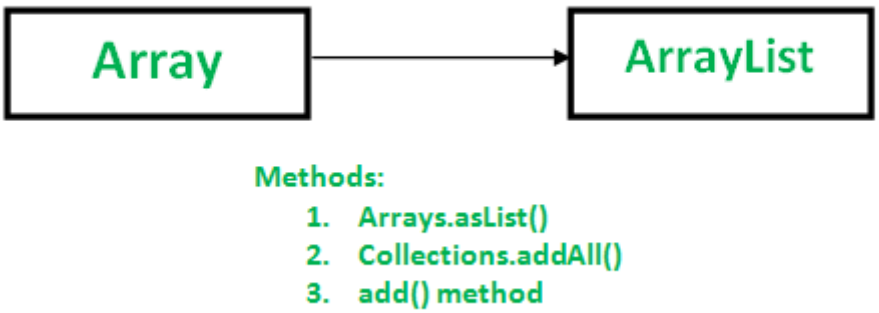
- **ArrayList()**: This constructor is to initialize an empty List.
- **ArrayList(int capacity):** In this constructor, we can pass capacity as a parameter, used to initialize the capacity by the user.
- **ArrayList(Collection c):** In this we pass a collection

-
-
-

It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.

Arrays to ArraList



ArrayList Example With Comparator

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

ArrayListExample.java

```
package com.jdk8.coolections.list;
```

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import com.jdk8.oop.encapsulation.Bird2;

public class ArrayListExample {

	public static void main(String[] args) {

		ArrayList<Integer> arrayList = new ArrayList<Integer>();
		arrayList.add(5);
		arrayList.add(-5);
		arrayList.add(7);
		System.out.println(arrayList);
		arrayList.add(1, 9);
		System.out.println(arrayList);
		arrayList.remove(0);
		System.out.println(arrayList);
//		arrayList.sort();
		Collections.sort(arrayList);
		System.out.println(arrayList);
		boolean contains = arrayList.contains(9);
		System.out.println(contains);

//		1st Normal For Loop
		for (int i = 0; i < arrayList.size(); i++) {
			System.out.println(arrayList.get(i));
		}

		ArrayList<Integer> arrayListCloned = (ArrayList<Integer>) arrayList.clone();
		System.out.println("CLONE " + arrayListCloned);

//		2nd for each loop
		for (Integer integer : arrayList) {// enhanced for loop
			arrayListCloned.add(7);
			System.out.println(integer);
		}

		System.out.println("CLONE " + arrayListCloned);
//		3rd
		Iterator<Integer> iterator = arrayList.iterator();
		while (iterator.hasNext()) {
			Integer integer = (Integer) iterator.next();
			System.out.println(integer);

		}
		ArrayList<Bird2> arrayListOfBird = new ArrayList<Bird2>();
		arrayListOfBird.add(new Bird2(1, "Crow", "Black", false));
		arrayListOfBird.add(new Bird2(9, "Pigeon", "Grey", false));
		arrayListOfBird.add(new Bird2(7, "Peacock", "Rainbow", true));
		arrayListOfBird.add(new Bird2(0, "Vultures", "Brown", true));
		arrayListOfBird.add(new Bird2(0, "Raven", "Deep Black", true));
//		Custom Sorting
//		BEFORE
		System.out.println("Before Sorting\n\n\n");
		for (Bird2 bird2 : arrayListOfBird) {
			System.out.println(bird2);
		}

		System.out.println("After Sorting\n\n\n");
		arrayListOfBird.sort(new BirdSorterViaNumber());
		for (Bird2 bird2 : arrayListOfBird) {
			System.out.println(bird2);
		}
	}
}

class BirdSorterViaNumber implements Comparator<Bird2> {
```

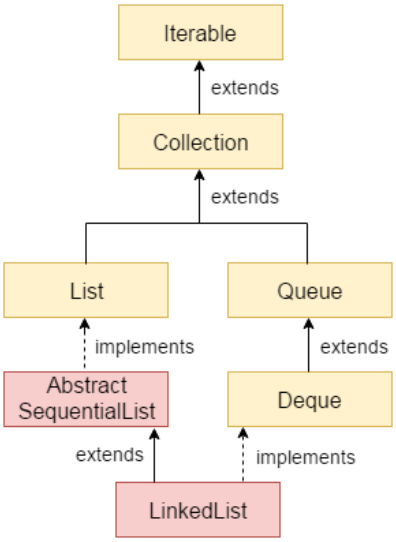
```
        @Override
        public int compare(Bird2 b1, Bird2 b2) {
                if (b1.getNumber() == b2.getNumber())
                        return 0;
                else if (b1.getNumber() > b2.getNumber())
                        return 1;
                else
                        return -1;
        }
}
```

Linked List

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.



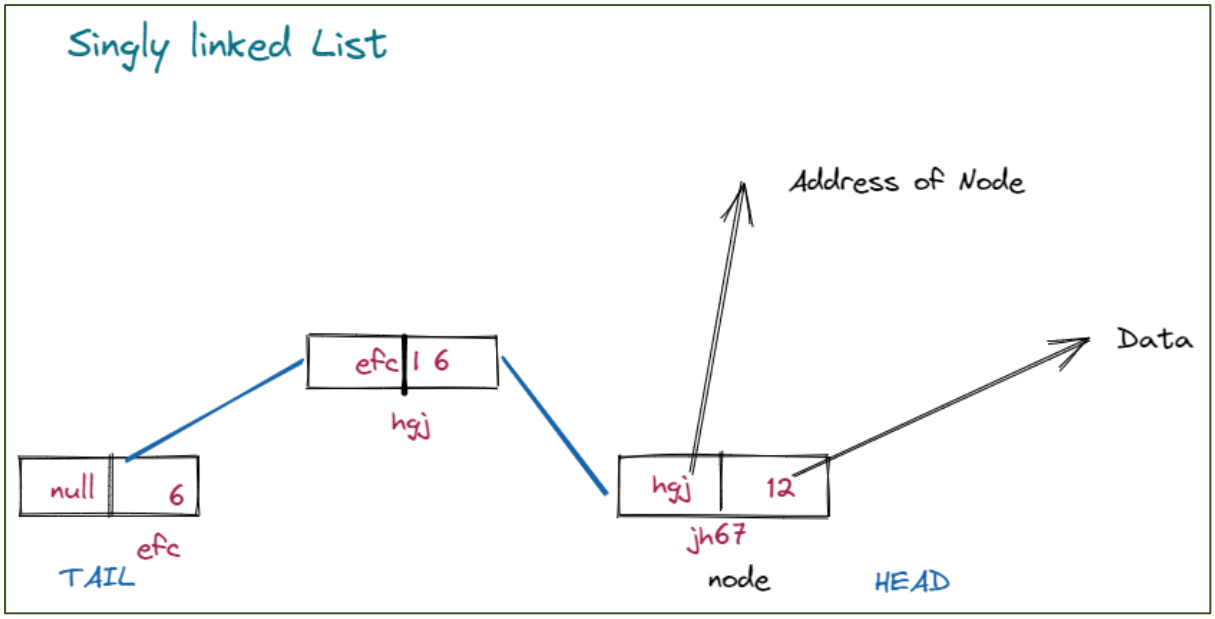


Figure 3 Singly LinkedList DataStructure

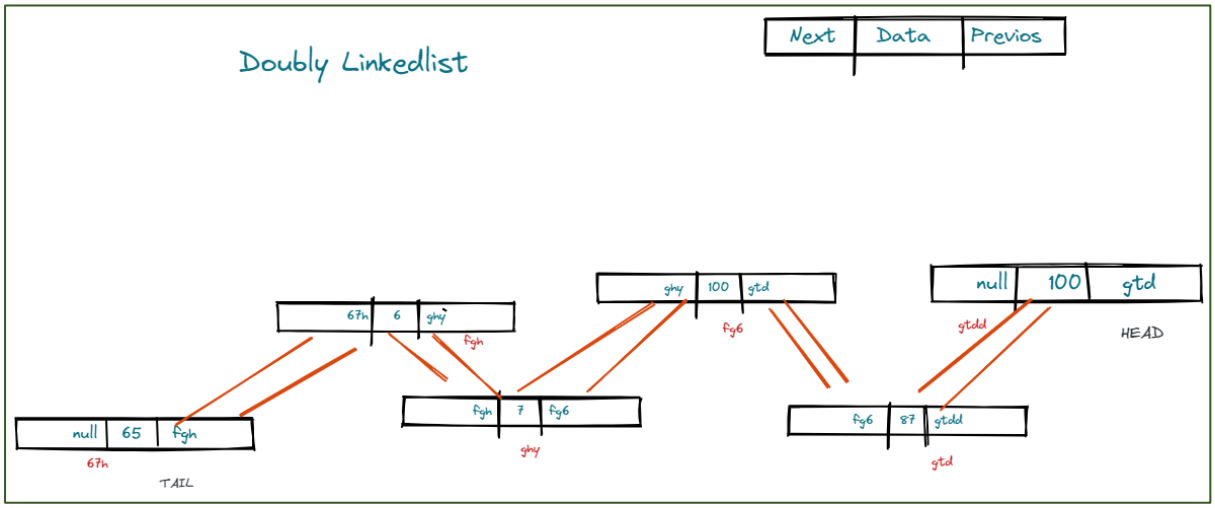


Figure 4 Doubly LinkedList DataStructure

LinkedListExample.java

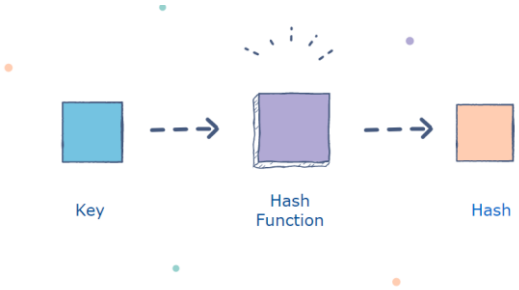
```
package com.jdk8.collections.linkedlist;
```

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.ListIterator;

public class LinkedListExample {

	public static void main(String[] args) {

		// PROS: Insertion , Deletion very fast
		// CONS: Traversal

		LinkedList<Integer> integerLinkedList = new LinkedList<>();
		integerLinkedList.add(5);
		integerLinkedList.add(6);
		integerLinkedList.add(66);
		integerLinkedList.add(76);
		integerLinkedList.add(68);
		integerLinkedList.add(2, 10);
		System.out.println(integerLinkedList.get(0));
		integerLinkedList.remove();
		System.out.println(integerLinkedList);

		for (Integer integer : integerLinkedList) {
			System.out.print(integer + "  ");
		}
		System.out.println();

//		Using Iterator for traversal
		Iterator<Integer> iterator = integerLinkedList.iterator();

		while (iterator.hasNext()) {
			System.out.print(iterator.next() + " ");
		}
		System.out.println();

		ArrayList<Integer> arrayList = new <Integer>ArrayList();
		arrayList.add(4);
		arrayList.add(14);
		arrayList.add(44);
		arrayList.add(45);
		arrayList.add(49);
		arrayList.add(433);

//		Iterator on ArrayList
		Iterator<Integer> iteratorForArrayList = arrayList.iterator();
		while (iteratorForArrayList.hasNext()) {
			Integer integer = (Integer) iteratorForArrayList.next();
			System.out.print(integer + " ");
		}
		System.out.println();

//		Custom Iterator on ArrayList
		ListIterator<Integer> listIterator =
arrayList.listIterator(arrayList.size());
		System.out.println("ARRAYLIST REVERSED ITERATOR");
		while (listIterator.hasPrevious()) {
			Integer integer = (Integer) listIterator.previous();
			System.out.print(integer + " ");
		}
		System.out.println();

//		Custom DescendingIterator onLinkedList
		Iterator<Integer> descendingIterator =
integerLinkedList.descendingIterator();
		while (descendingIterator.hasNext()) {// hasNext() will work because its a
doubly linkedlist
			Integer integer = (Integer) descendingIterator.next();
			System.out.print(integer + " ");
```

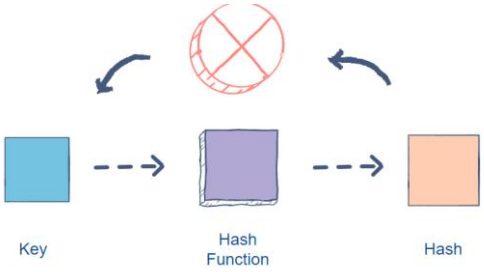
```
                    }
                    System.out.println();

//          -1(I) [5, 6, 10, 66, 76, 68]
          }
}
```

What is hashing?

Hashing is the process of converting a given key into another value. A **hash function** is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a **hash value** or simply, a **hash**.

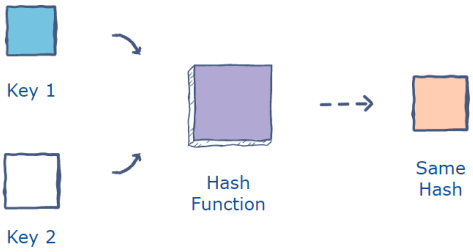


A good hash function uses a **one-way** hashing algorithm, or in other words, the hash cannot be converted back into the original key.



Collisions

Keep in mind that two keys can generate the same hash. This phenomenon is known as a collision. There are several ways to handle collisions, but that's a topic for another time.

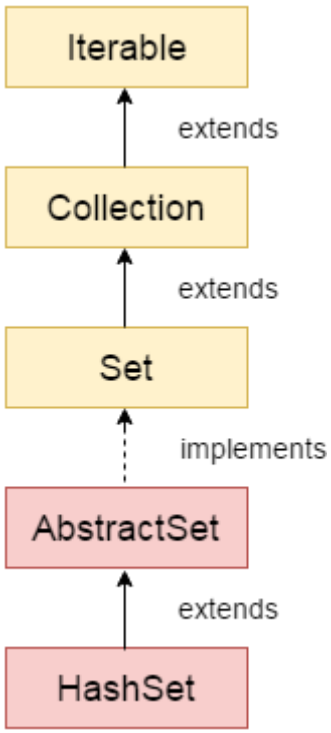


HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called hashing.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.



HashSetExample.java

```
package com.jdk8.collections.hashset;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;

public class HashSetExample {
	// PROS: Searching , No Duplicates
	// CONS: Takes More Space, prohibits Random Access via Indexing
	public static void main(String[] args) {

		HashSet<Integer> hashSetInteger = new HashSet();
		hashSetInteger.add(null);
		hashSetInteger.add(1);
		hashSetInteger.add(2);
		hashSetInteger.add(4);
		hashSetInteger.add(6);
		hashSetInteger.add(9);
		hashSetInteger.add(4);
		hashSetInteger.add(4);
		hashSetInteger.add(4);
		hashSetInteger.add(4);
		hashSetInteger.add(40);
		hashSetInteger.add(45);
		hashSetInteger.add(46);
		hashSetInteger.add(47);
		hashSetInteger.add(48);
		hashSetInteger.add(499);
		hashSetInteger.add(49);
		System.out.println(hashSetInteger);

		HashSet<String> hashSetStrings = new HashSet();
		hashSetStrings.add("Jan");
		hashSetStrings.add("Feb");
		hashSetStrings.add("Mar");
		hashSetStrings.add("Apr");
		hashSetStrings.add("May");
		hashSetStrings.add("June");
		hashSetStrings.remove("Mar");
		System.out.println(hashSetStrings);

//		Iterator on HashSet
		Iterator<String> iterator = hashSetStrings.iterator();
		while (iterator.hasNext()) {
			String string = (String) iterator.next();
			System.out.println(string);
		}

		ArrayList<Integer> arrayList = new <Integer>ArrayList();
		arrayList.add(4);
		arrayList.add(14);
		arrayList.add(44);
		arrayList.add(44);
		arrayList.add(44);
		arrayList.add(45);
		arrayList.add(49);
		arrayList.add(4);
		arrayList.add(433);
		Collections.sort(arrayList);
		System.out.println(arrayList);

		ArrayList<Integer> arrayList2 = new <Integer>ArrayList();
		arrayList2.add(3);
		arrayList2.add(34);
		arrayList2.add(35);
		arrayList2.add(3);
```

```
//                  Converting an ArrayList to HashSet
                    HashSet<Integer> hashSet = new HashSet<>(arrayList);
                    System.out.println(hashSet);

//                  Searching in HashSet
                    boolean contains = hashSet.contains(434);// fastest among all the
collections
                    System.out.println(contains);

//                  Extending my existing HashSet
                    hashSet.addAll(arrayList2);
                    System.out.println(hashSet);

          }

}
```

Queue

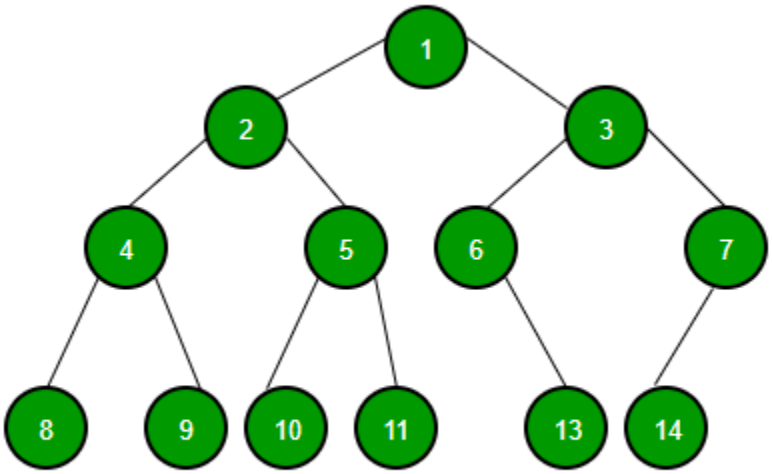
Java Queue

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

Heap

Binary Tree

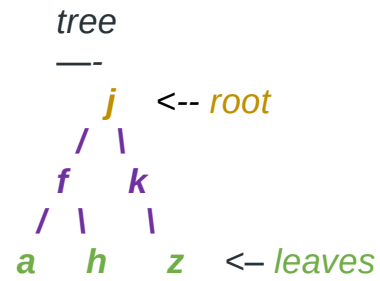
A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A **tree** is a popular data structure that is non-linear in nature. Unlike other data structures like array, stack, queue, and linked list which are linear in nature, a tree represents a hierarchical structure. The ordering information of a tree is not important. A tree contains nodes and 2 pointers. These two pointers are the left child and the right child of the parent node. Let us understand the terms of tree in detail.

- **Root:** The root of a tree is the topmost node of the tree that has no parent node. There is only one root node in every tree.

- **Edge:** Edge acts as a link between the parent node and the child node.
- **Leaf:** A node that has no child is known as the leaf node. It is the last node of the tree. There can be multiple leaf nodes in a tree.
- **Depth:** The depth of the node is the distance from the root node to that particular node.
- **Height:** The height of the node is the distance from that node to the deepest node of the tree.
- **Height of tree:** The Height of the tree is the maximum height of any node.



Basic Operation on Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing an element. There are three types of traversals in a binary tree which will be discussed ahead.

Auxiliary Operation on Binary Tree:

- Finding the height of the tree
- Find the level of the tree
- Finding the size of the entire tree.

Applications of Binary Tree:

- In compilers, Expression Trees are used which is an application of binary tree.
- Huffman coding trees are used in data compression algorithms.
- Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(logN) time complexity.

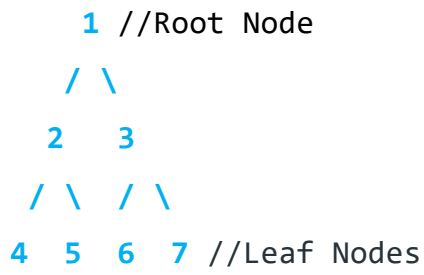
Binary Tree Traversals:

- **PreOrder Traversal:** Here, the traversal is: root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
- **InOrder Traversal:** Here, the traversal is: left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.
- **PostOrder Traversal:** Here, the traversal is: left child – right child – root. It means that the left child is traversed first then the right child and finally its root node.

Let us traverse the following tree with all the three traversal methods:

Tree

_____



PreOrder Traversal of the above tree: 1-2-4-5-3-6-7
InOrder Traversal of the above tree: 4-2-5-1-6-3-7
PostOrder Traversal of the above tree: 4-5-2-6-7-3-1

Why Use Trees?
1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:
file system

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of multi-stage decision-making (see business chess).

Binary Tree: A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.
Binary Tree Representation: A tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.
A Tree node contains the following parts.

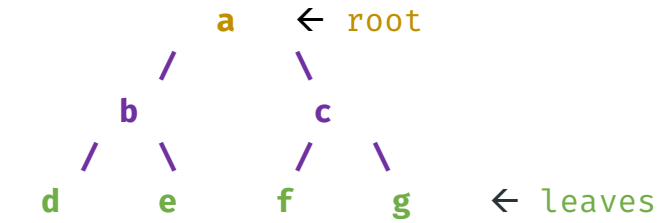
1. Data
2. Pointer to the left child

3. Pointer to the right child

```
// Class containing left and right child
// of current node and key value
class Node {
    int key;
    Node left, right;

    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}

// A Java program to introduce Binary Tree
class BinaryTree {

    // Root of Binary Tree
    Node root;

    // Constructors
    BinaryTree(int key) { root = new Node(key); }

    BinaryTree() { root = null; }

    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();

        // create root
        tree.root = new Node(1);

        /* following is the tree after above statement

              1
            /   \
          null  null     */

        tree.root.left = new Node(2);
        tree.root.right = new Node(3);

        /* 2 and 3 become left and right children of 1
               1
             /     \
           2        3
         /   \     /  \
      null null null null  */

        tree.root.left.left = new Node(4);
        /* 4 becomes left child of 2
                    1
                  /      \
                2          3
              /   \       /  \
             4    null null  null
           /  \
         null null
         */
    }
}
```

Representing a BST in an Array

Case :1



The below array **Z** is filled using level traversal

0	1	2	3	4	5	6	7
	a	b	c	d	e	f	g

Usually, it is studied with index starting with 1

While representing a tree in an array, we see the elements and their relationship and how they are preserved.

If a node is at index ***i***

- Its left child is at ***2*i***
- The right child is at ***2*i + 1***
- The parent is at *floor* ***[i/2]***

Let's check **b** (2)

L(f) = 2*2 = 4 Z[4] = d

R(f) = 2*2 = 1 = 5 Z[5] = e

P(f) = [2/2] = 1 Z[1] = a

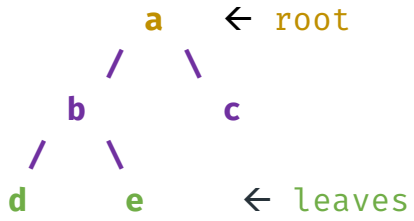
Let's check f (6)

L(f) = 2*6 = 12 Out of bounds, therefore its null

R(f) = 2*6 = 1 = 13 Out of bounds, therefore its null

P(f) =[6/2] = Z[3] = c

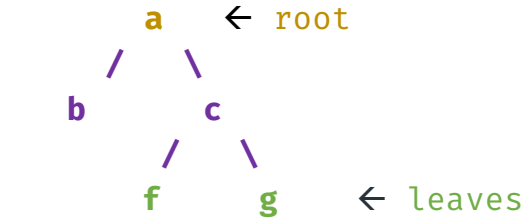
Case :2



The below array **Z** is filled using level traversal

0	1	2	3	4	5
	a	b	c	d	e

Case :3*



The below array **Z** is filled using level traversal

0	1	2	3	4	5	6	7
	a	b	c	-	-	f	g

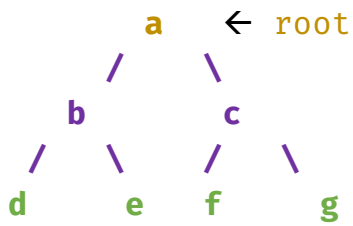
Binary tree

Full Binary Tree

A binary tree with maximum number of nodes for a given height is called full binary tree.

Total number of elements for a full binary tree will $2^{(h+1)} + 1$

Every full binary tree is a compete binary tree



Complete Binary Tree

While representing in an array, if there is any empty element in the array, it's not a complete Binary Tree and *vice versa.*

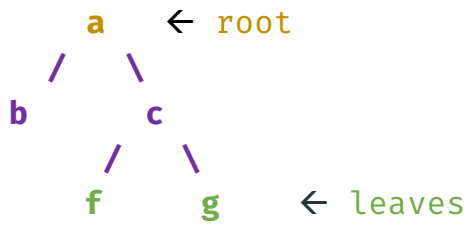
Any incomplete binary tree is not a full binary tree.

For a complete binary tree, we **must** have node from left to right.

For a given level You cannot have a node in right all of a sudden without left nodes filled.

For achieving a complete binary tree, we are not going to next level without filling the prior levels from top to bottom. That's why *the height of a complete binary tree will be* ***log(n)***

Also compete binary tree is a full binary tree to level ***h-1***



The below array **Z** is filled using level traversal is an **incomplete** binary tree

0	1	2	3	4	5	6	7
	a	b	c	-	-	f	g

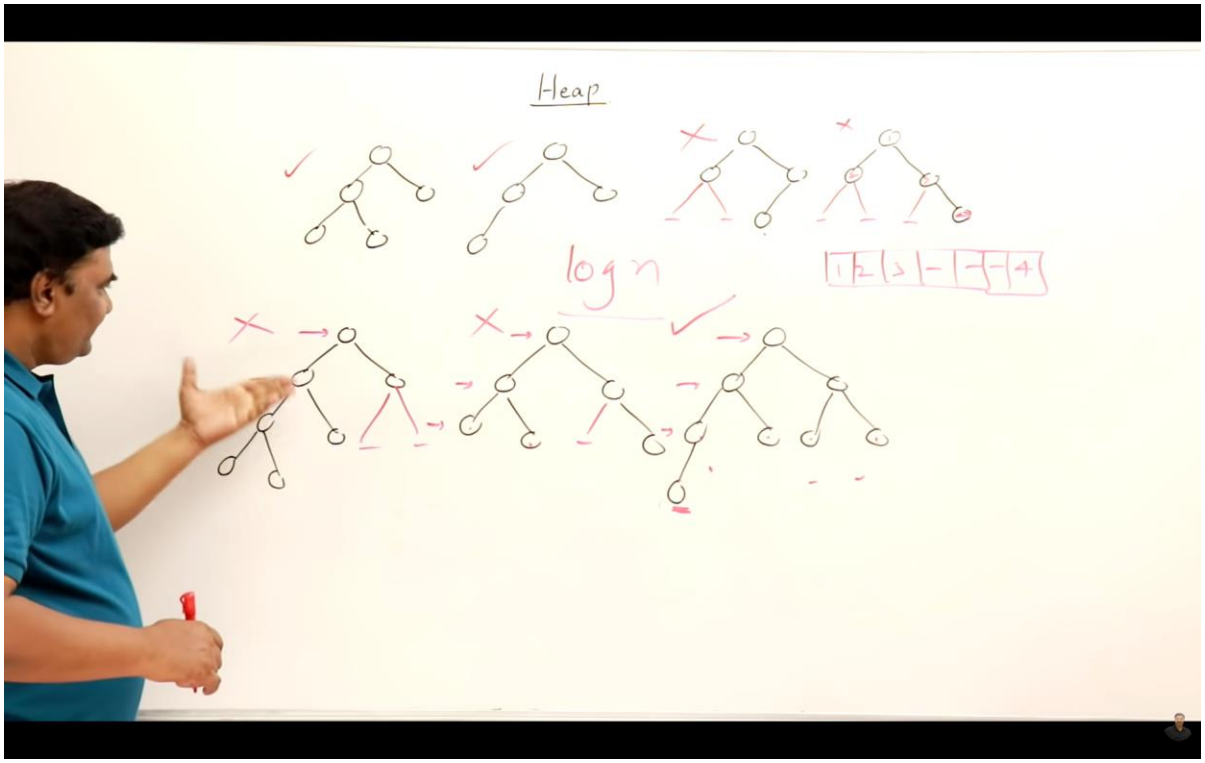


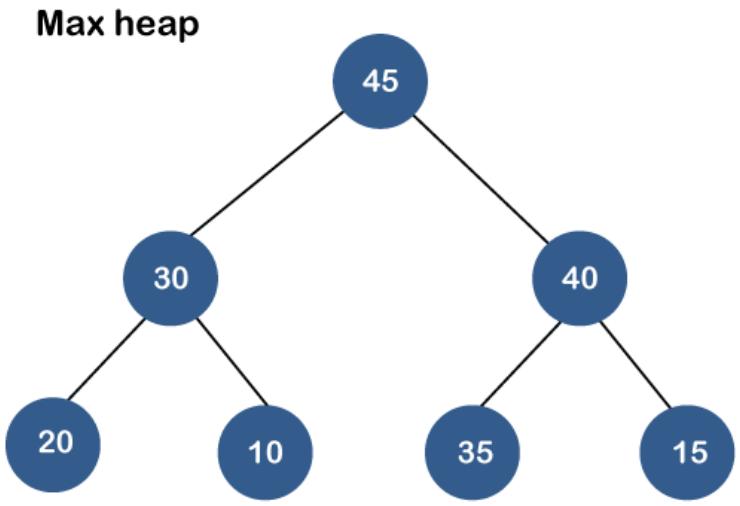
Figure 5: Checking trees for their complete binary tree

https://www.youtube.com/watch?v=HqPJF2L5h9U

What is Heap?

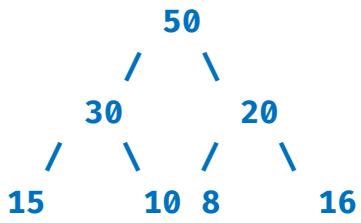
A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of every parent node is **greater** than or **equal** the value of the child nodes. So duplicates are allowed.



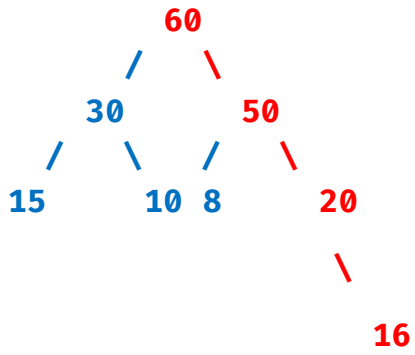
Insertion of a new element in a MAX HEAP

Take a MAX heap like below



0	1	2	3	4	5	6	7	9
	50	30	20	15	10	8	16	

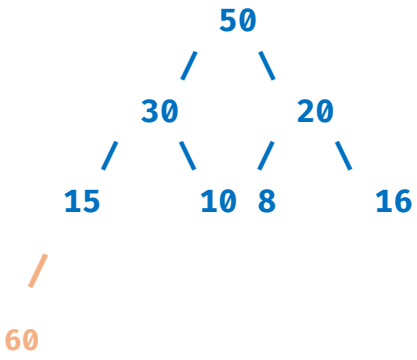
Wrong notion of people how its added



This is totally wrong as now its not a complete binary tree.

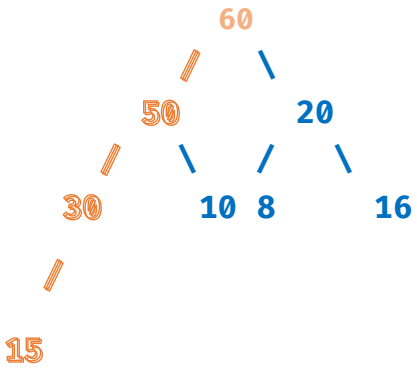
The correct way is to add in last free space in an array. So, 60 will be added in the last free space in an array

0	1	2	3	4	5	6	7	9
	50	30	20	15	10	8	16	60



But its not a complete HEAP satisfied, so rearrangement will happen.

1. 60 is compared with parent ancestor nodes and it will reach it right place, if its greater than parent, it will go up.



0	1	2	3	4	5	6	7	9
	50	30	20	15	10	8	16	60

So, in array to compare with parent

Index of (60) = 9

P (60) = 9/2 = 4 = (15)

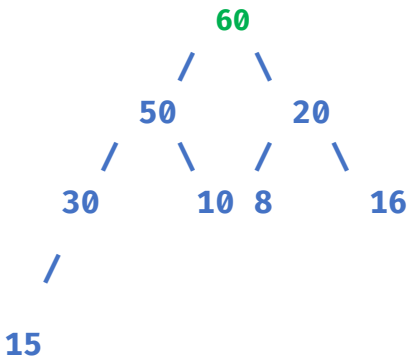
60 > 15 So swap

0	1	2	3	4	5	6	7	9
	50	30	20	60	10	8	16	15

Similarly, it will compare and swap all the way to its correct position.

0	1	2	3	4	5	6	7	9
	50	60	20	30	10	8	16	15

0	1	2	3	4	5	6	7	9
	60	50	20	30	10	8	16	15



So maximum swaps take can be **h** so it depends on the **h** of the tree and max **h** of a tree can be **log(n)**

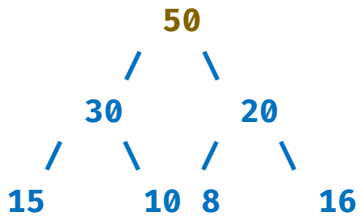
So, **O (1) to O (log n)** for insertion is the time complexity.

So, while inserting we add the element as a leaf then we compare with the ancestors, so element moves from leaf to root. So, the direction of adjustment is upwards.

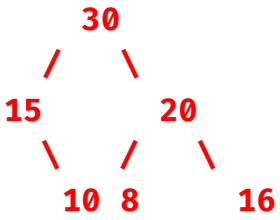
Deletion of an element in a MAX HEAP

In a mall, we see an apple tower, the top most apple is the shiniest one and that is removed first.

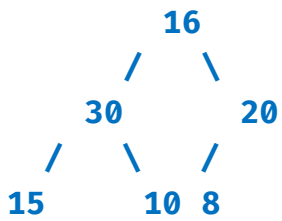
So similarly, here root will be removed 1st.



So, if we just remove 50 and try to adjust the tree, chances are that it will not remain a complete binary tree.

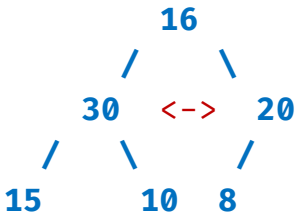


So, the correct way is when the **root** element



Now after doing the deletion and swapping ops, it's a complete binary tree but it not satisfies as a max HEAP. So, we will readjust it preserving the COMPLETE binary tree.

We will start from ROOT to leaves. And we will compare the parent (**ROOT**)with the children.



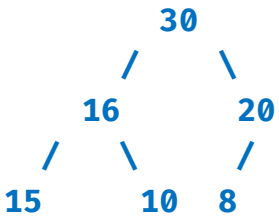
Compare 16 with its children.

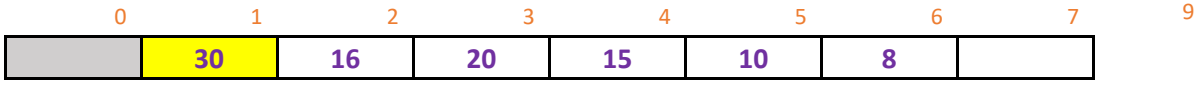
And swap the bigger child with 16

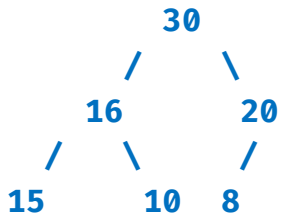
Index of (16) = 1

L (16) = 2*i = 2*1 =2 = (30) ~~~~> Swap the 30 as it is the bigger child

R (16) = 2*i + 1 = 2*1 + 1=3 (20)







0	1	2	3	4	5	6	7	9
	30	16	20	15	10	8		

Comparing the children of 16, we find it is bigger than all of its child so now swaps and our heap is ready.

1. So, while deletion we 1st remove the root.
2. The last element (*last element in an array or the last leaf) in* CBT will take its place
3. And we push the element downwards towards the leaves to readjust to form the MAX HEAP.
4. For deletion the maximum adjustment will depend on the height, So **O (log n)**
5. While deletion we get the Maximum element of the MAX Heap. Again, deleting will give the 2nd largest and so on.

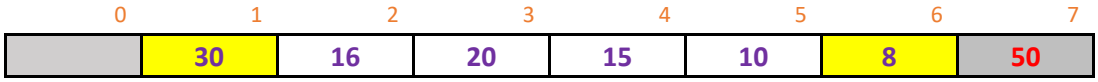
Insertion ~~~~> Adjustment leaf to root

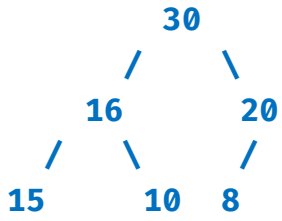
Deletion ~~~~~> adjustment root to leaf

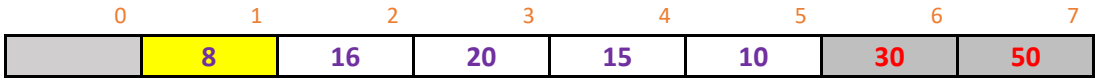
0	1	2	3	4	5	6	7
	30	16	20	15	10	8	50

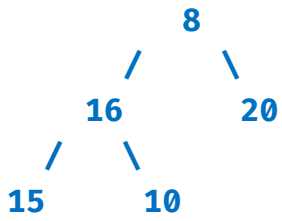
We can keep the deleted element in the vacant space away from the HEAP.

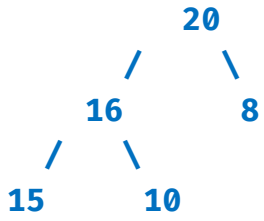
If we delete the next element from the HEAP, our array will look like









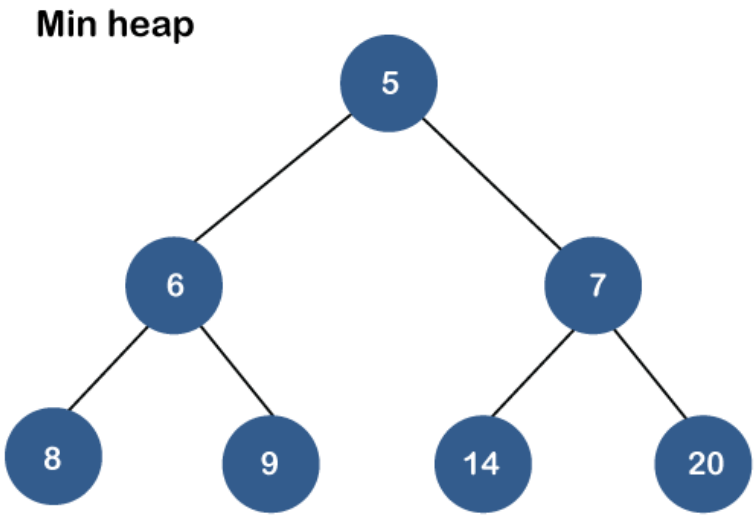


So stored deleted part is in Descending order and it gets sorted automatically using Heap Sort.

HEAP Sort

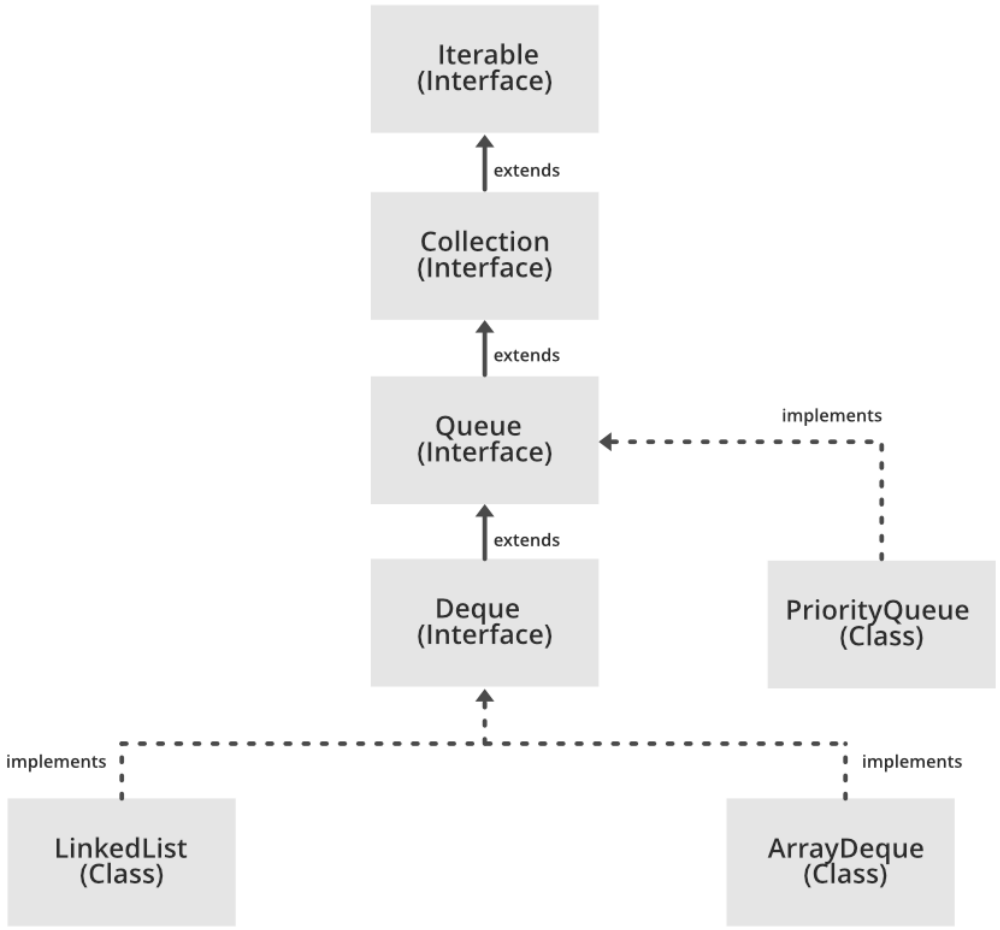
1. Create a HEAP
2. Delete all the elements from the HEAP.

- **Min heap:** The min heap is a heap in which the value of every parent node is less than or equal to the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

Priority Queue



Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java. Implementations done by these classes are not thread safe**. If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option**.

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a Queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play.

The priority Queue doesn't follow FIFO rule like normal queue, It arrange sthe element based on priorities. Priorities queues can store comparable objects to arrange them in priorities.

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come

first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

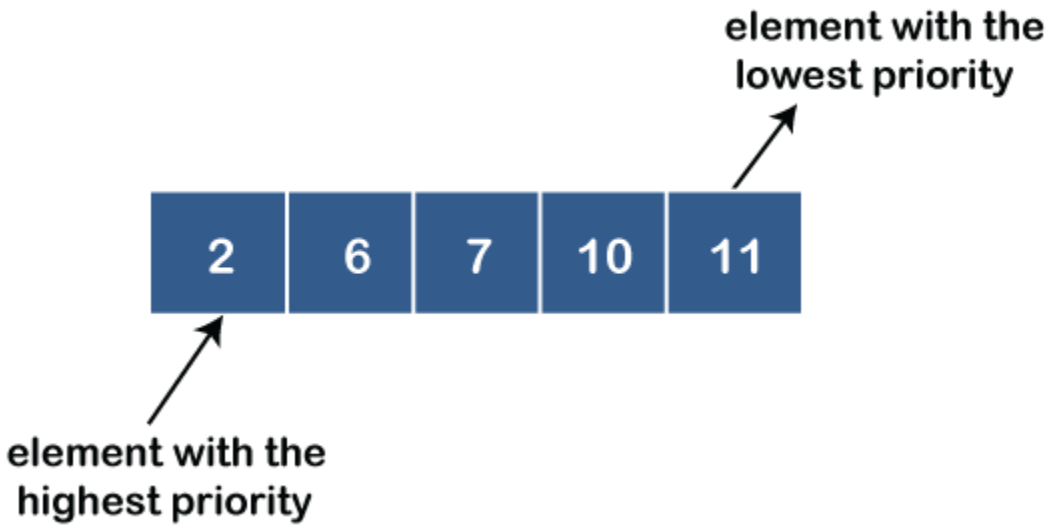
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.

- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

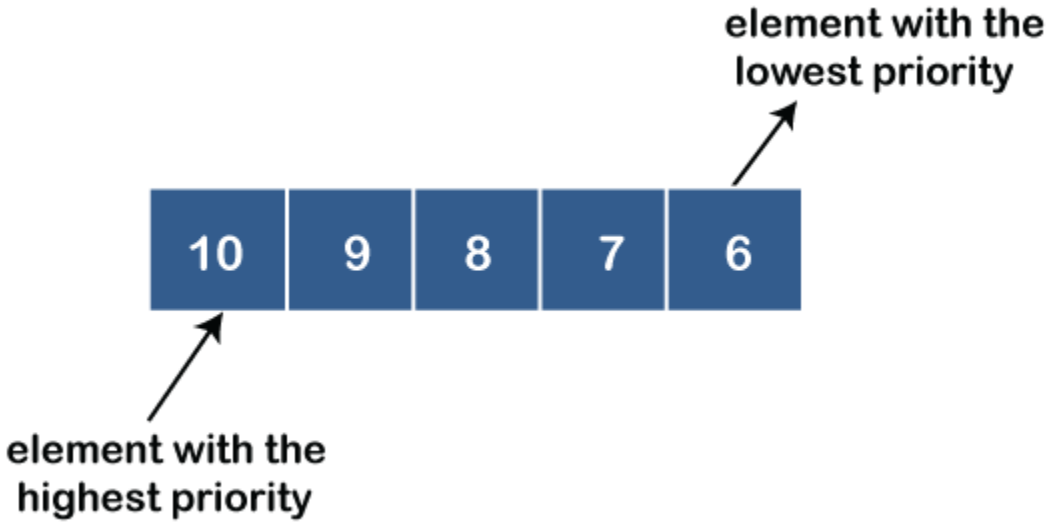
There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest

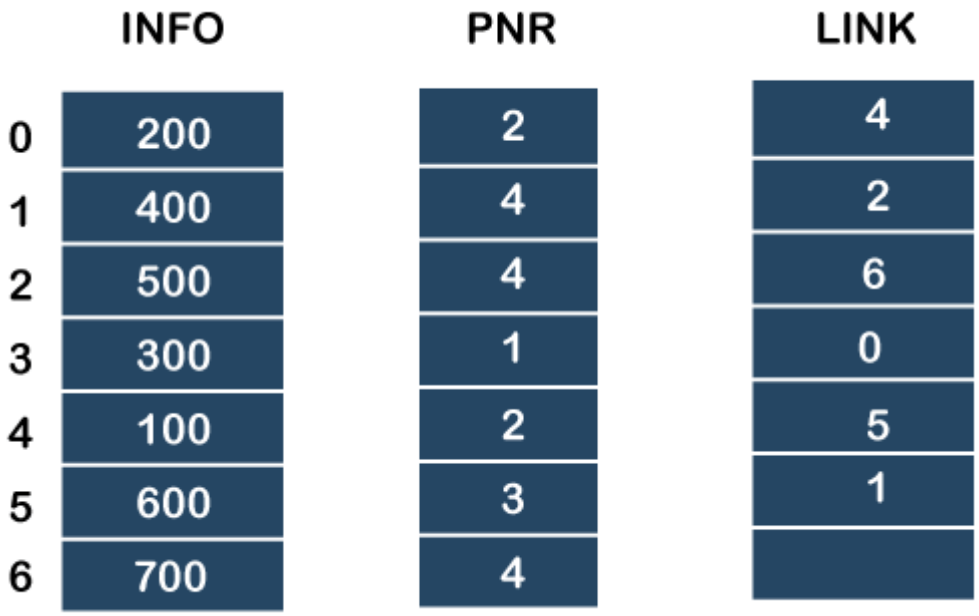
number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and **LINK** basically contains the address of the next node.



Let's create the priority queue step by step.

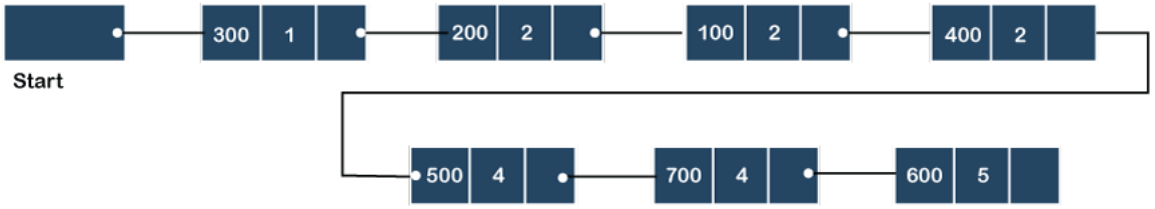
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 300, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 300, priority number 2 is having a higher priority, and data values associated with this priority are 200 and 100. So, this data will be inserted based on the FIFO principle; therefore 200 will be added first and then 100.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 400, 500, 700. In this case, elements would be inserted based on the FIFO principle; therefore, 400 will be added first, then 500, and then 700.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 600, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

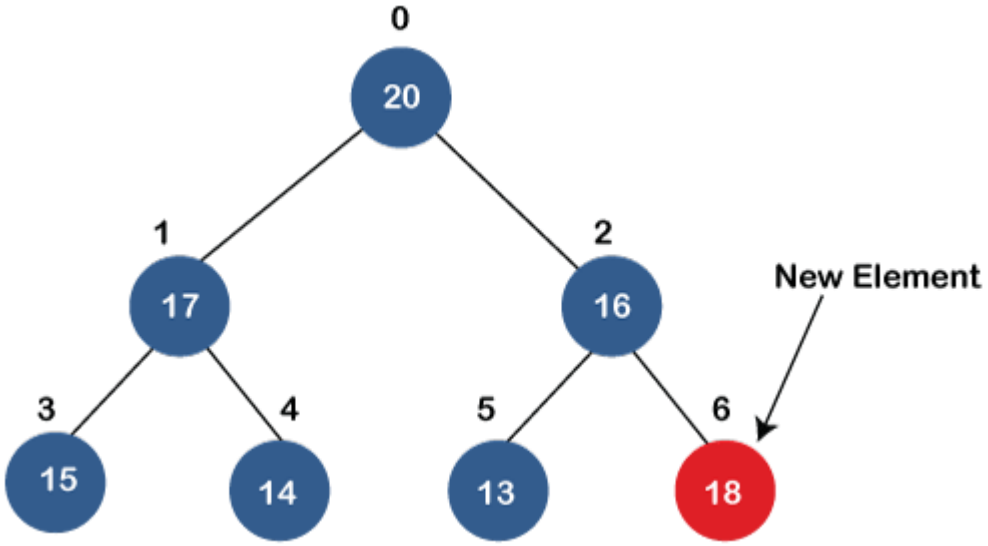
Priority Queue Operations

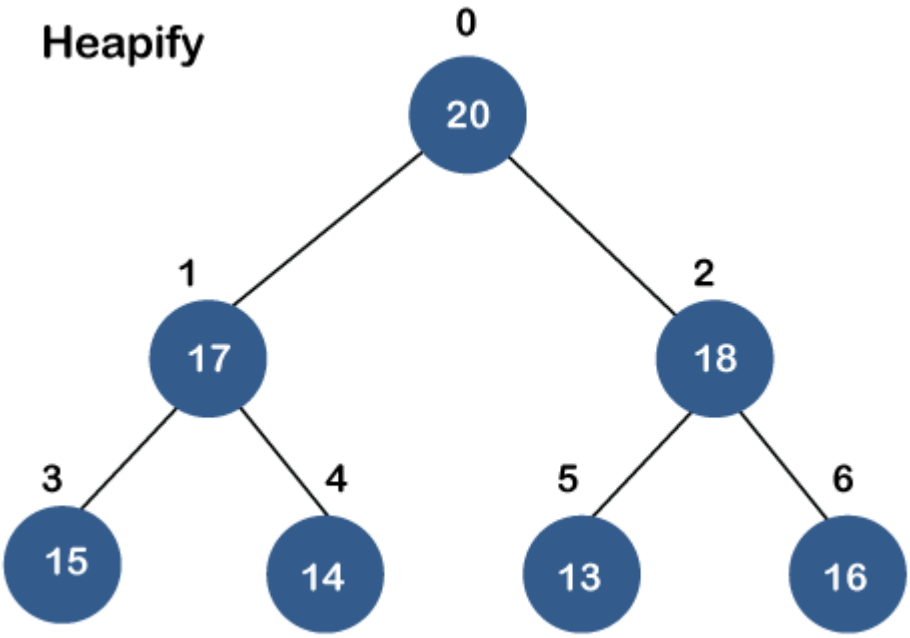
The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.





- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

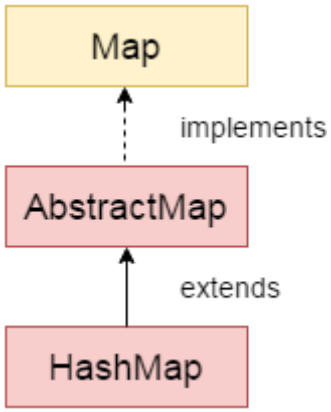
The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

HashMap

Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc.

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.



java.util.Properties Vs java.util.Map<String, String>

There is different purpose of these two-utility class. Map, or in your case, HashMap is general purpose storage object where you have unique key, and values to which they point. HashMap can have any object type as their key and any object type as their values.

java.util.Properties is, however, a special purpose map. It is developed to read/write from/to properties files. It has special methods to do so [see load(..)]. Map does not.

So, you have different situations to use them. Places where you need properties to read, you better go with Properties. And places where you want to have lookup values stored off some logic, you go with HashMap<String, String>.

There is no hard and fast rule, you can use HashMap<String, String> and Properties interchangeably. But as an engineer, use *right tool for the task*.

https://stackoverflow.com/questions/9463268/java-util-properties-vs-java-util-mapstring-string

HashMapExample.java

```
package com.jdk8.collections.hashmap;

import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class HashMapExample {
          // PROS: k:v storage, No Duplicate Keys, Faster searching
          // CONS: Takes More Space, prohibits Random Access via Indexing
          public static void main(String[] args) {

                    HashMap<Integer, String> studentListRollNumberWise = new HashMap<>();
                    studentListRollNumberWise.put(1, "Sam");// Entry
                    studentListRollNumberWise.put(2, "Ronnie");
                    studentListRollNumberWise.put(6, "Christopher");
                    studentListRollNumberWise.put(9, "Jack");
                    studentListRollNumberWise.put(10, "Jane");
                    studentListRollNumberWise.put(10, "Jim");
                    studentListRollNumberWise.remove(9);// removes the whole k:v entry
                    studentListRollNumberWise.replace(1, "Samntha");
                    System.out.println(studentListRollNumberWise);

                    boolean containsKey = studentListRollNumberWise.containsKey(1);// searching
                    System.out.println(containsKey);
                    boolean containsValue = studentListRollNumberWise.containsValue("1");
                    System.out.println(containsValue);
```

```
                    String string = studentListRollNumberWise.get(1);// gets the value for
corresponding keys
                    System.out.println(string);

//                  keys can only be Immutable Values; String, WrapperClasses are immutable
//                  [String Buffer, Builder...] are mutable
                    HashMap<String, Integer> fibonacci = new HashMap<>();
                    fibonacci.put("1", 1);

//                  Iterating a HashMap
                    for (Entry<Integer, String> entry : studentListRollNumberWise.entrySet()) {
                              System.out.println(entry);
                              Integer key = entry.getKey();
                              String val = entry.getValue();
                              System.out.println(key + "--> " + val);
                    }

                    System.out.println("KEYS");
//                  Iterating a HashMap Keys only
                    for (Integer key : studentListRollNumberWise.keySet()) {
                              System.out.print(key + " ");
                    }

//                  Iterating a HashMap Values only
                    for (String value : studentListRollNumberWise.values()) {
                              System.out.println(value);
                    }

                    // Exporting the EntrySet of a HashMap
                    Set<Entry<Integer, String>> entrySet = studentListRollNumberWise.entrySet();
          }
}
```

Sorting using sort() | Comparable | Comparator

Java **Comparable** interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

Java **Comparator** interface is used to order the objects of a user-defined class.This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Student.java

```
package com.jdk8.customsorting;

public class Student implements Comparable<Student> {
	int roll;
	String name;
	int age;

	public Student(int roll, String name, int age) {
		super();
		this.roll = roll;
		this.name = name;
		this.age = age;
	}

	@Override
	public int compareTo(Student s) {
		if (age == s.age)
			return 0;
		else if (age > s.age)
			return 1;
		else
			return -1;
	}

	@Override
	public String toString() {
		return "Student [roll=" + roll + ", name=" + name + ", age=" + age + "]";
	}

}
```

StudentPlain.java

```
package com.jdk8.customsorting;

public class StudentPlain {
	int roll;
	String name;
	int age;

	public StudentPlain(int roll, String name, int age) {
		super();
		this.roll = roll;
		this.name = name;
		this.age = age;
	}

	@Override
	public String toString() {
		return "Student [roll=" + roll + ", name=" + name + ", age=" + age + "]";
	}

}
```

StudentPlain.java

```
package com.jdk8.customsorting;

public class StudentPlain {
	int roll;
	String name;
	int age;

	public StudentPlain(int roll, String name, int age) {
		super();
		this.roll = roll;
		this.name = name;
		this.age = age;
	}

	@Override
	public String toString() {
		return "Student [roll=" + roll + ", name=" + name + ", age=" + age + "]";
	}

}
```

CustomSortingExample.java

```
package com.jdk8.customsorting;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class CustomSortingExample {

	public static void main(String[] args) {

		ArrayList<Student> arrayList = new ArrayList();
		arrayList.add(new Student(1, "Sam", 55));
		arrayList.add(new Student(7, "Samantha", 45));
		arrayList.add(new Student(9, "Rocky", 39));
		arrayList.add(new Student(8, "Christopher", 6));
		arrayList.add(new Student(67, "Tamy", 67));

		System.out.println(arrayList);

		Collections.sort(arrayList);// Custom Sorting using Comparable
		System.out.println(arrayList);

		ArrayList<StudentPlain> arrayListPlain = new ArrayList<StudentPlain>();
		arrayListPlain.add(new StudentPlain(1, "Sam", 55));
		arrayListPlain.add(new StudentPlain(7, "Samantha", 45));
		arrayListPlain.add(new StudentPlain(9, "Rocky", 39));
		arrayListPlain.add(new StudentPlain(8, "Christopher", 6));
		arrayListPlain.add(new StudentPlain(67, "Tamy", 67));
		System.out.println(arrayListPlain);

		// Custom Sorting using Comparator
		Collections.sort(arrayListPlain, new RollComparator());// Sorting based on
Roll
		System.out.println(arrayListPlain);

//		Using Lambda | Sorting further based on Age
		Collections.sort(arrayListPlain, (Comparator<StudentPlain>) (StudentPlain
s1, StudentPlain s2) -> {

			if (s1.age == s2.age)
				return 0;
			else if (s1.age > s2.age)
				return 1;
			else
				return -1;
		});
		System.out.println(arrayListPlain);

	}
}
```

*Comparable vs **Comparator***

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Day 8

Multithreading

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

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Advantages of Java Multithreading

1. It doesn't block the user because threads are independent, and you can perform multiple operations at the same time.
2. You can perform many operations together, so it saves time.
3. Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

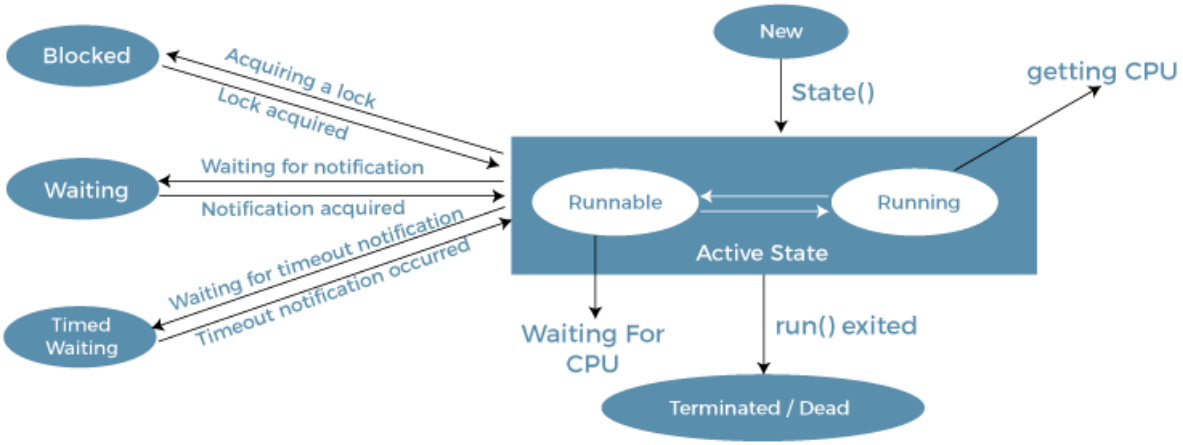
- A process is an active program i.e. a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.
- A thread is a lightweight process that can be managed independently by a scheduler. It improves the application performance using parallelism. A thread shares information like data segment, code segment, files etc. with its peer threads while it contains its own registers, stack, counter etc.
- The major differences between a process and a thread are given as follows –

Comparison Basis	Process	Thread
Definition	A process is a program under execution i.e an active program.	A thread is a lightweight process that can be managed independently by a scheduler.
Context switching time	Processes require more time for context switching as they are more heavy.	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.

Comparison Basis	Process	Thread
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.

ThreadScheduler

Life Cycle of a thread



Life Cycle of a Thread

1. **New Thread:** When a new thread is created, it is in the **new** state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and has not started to execute.
2. **Runnable State:** A thread that is **ready to run** is moved to a runnable state. In this state, a thread might be running, or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked/Waiting state:** When a thread is temporarily inactive, then it's in one of the following states:
   - Blocked
   - Waiting
4. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
5. **Terminated State:** A thread terminates because of either of the following reasons:
   - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

ThreadTwo.java

```
package com.fewcodes.multithreading;

class ThreadTwo implements Runnable {
          public void run() {

                    try {
// moving thread t2 to the state timed waiting
                              Thread.sleep(100);
                    } catch (InterruptedException ie) {
                              ie.printStackTrace();
                    }

                    System.out.println(
                                        "The state of thread t1 while it invoked the method
join() on thread t2 -" + ThreadState.t1.getState());

// try-catch block
                    try {
                              Thread.sleep(200);
                    } catch (InterruptedException ie) {
                              ie.printStackTrace();
                    }
          }
}

// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable {
          public static Thread t1;
          public static ThreadState obj;

          public static void main(String argvs[]) {
                    obj = new ThreadState();
                    t1 = new Thread(obj);

// thread t1 is spawned
// The thread t1 is currently in the NEW state.
                    System.out.println("The state of thread t1 after spawning it - " +
t1.getState());

// invoking the start() method on   the thread t1
                    t1.start();
                    // thread t1 is moved to the Runnable state

                    System.out.println("The state of thread t1 after invoking the method start()
on it - " + t1.getState());
          }

          public void run() {
                    ThreadTwo myObj = new ThreadTwo();
                    Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
                    System.out.println("The state of thread t2 after spawning it - " +
t2.getState());
                    t2.start();

// thread t2 is moved to the runnable state
                    System.out.println("the state of thread t2 after calling the method start()
on it - " + t2.getState());

                    try {
// moving the thread t1 to the state timed waiting
                              Thread.sleep(200);
                    } catch (InterruptedException ie) {
                              ie.printStackTrace();
                    }
```

```
                    System.out.println("The state of thread t2 after invoking the method sleep()
on it - " + t2.getState());

                    try {
// waiting for thread t2 to complete its execution
                              t2.join();
                    } catch (InterruptedException ie) {
                              ie.printStackTrace();
                    }
                    System.out.println("The state of thread t2 when it has completed it's
execution - " + t2.getState());
          }

}
```

Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.

Achieving Multithreading in Java

Multithreading in Java can be achieved in Java in two ways

- Runnable interface
- Thread class

By Using Runnable interface

RunnerOne.java

```
package com.jdk8.multithreading;

public class RunnerOne implements Runnable {

	@Override
	public void run() {

		for (int i = 0; i < 20; i++) {
			try {
				Thread.sleep(40);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			System.out.println("Iteration -> " + i + " ; [" +
Thread.currentThread().getName() + " "
					+ Thread.currentThread().getState()+ "]");

		}

	}
}
```

DriverOne.java

```
package com.jdk8.multithreading;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class DriverOne {

	public static void main(String[] args) {

		System.out.println("START");
		RunnerOne runnerOne = new RunnerOne();// task

		Thread thread1 = new Thread(runnerOne);// thread to execute task
		String name = thread1.getName();
		System.out.println(name);

		Thread thread2 = new Thread(new RunnerOne());
		Thread thread3 = new Thread(new RunnerOne());
		thread3.getState();
		thread1.setPriority(Thread.MAX_PRIORITY);
		thread2.setPriority(Thread.NORM_PRIORITY);
		thread3.setPriority(Thread.MIN_PRIORITY);// [1-10]
		thread1.getState();
		thread1.start();
		thread2.start();
		thread3.start();

		System.out.println("END");

//		ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(10);
//		CompletableFuture<T>

	}

}
```

By Using Thread Class

RunnerTwo.java

```
package com.jdk8.multithreading;

public class RunnerTwo extends Thread {

	@Override
	public void run() {

		for (int i = 0; i < 20; i++) {
			try {
				Thread.sleep(5);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			System.out.println("Iteration -> " + i + " ; [" +
Thread.currentThread().getName() + "]");

		}

	}

}
```

DriverTwo.java

```
package com.jdk8.multithreading;

public class DriverTwo {

	public static void main(String[] args) {

		System.out.println("START");
		RunnerTwo r1 = new RunnerTwo();
		r1.start();
		try {
			r1.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

//		RunnerTwo r2 = new RunnerTwo();
//		r2.start();
//
//		RunnerTwo r3 = new RunnerTwo();
//		r3.start();
		System.out.println("END");

	}
}
```

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

Priority: Priority of each thread lies between **1 to 10** If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler. We can set thread prority via setPriority() method.

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

String Buffer vs String Builder

StringBuffer	StringBuilder
StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is *little less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

StringBufferBuilderLiteral.java

```
package com.jdk8.strings;

public class StringBufferBuilderLiteral {

	public static void main(String[] args) {

		// Immutable STrings
		String s1 = "hello";// Literal
		String s2 = new String(s1);

		String s1new = s1.concat(" world");
		System.out.println(s1new);
		System.out.println(s1);

		if (s1 == s2)
			System.out.println(true);
		else
			System.out.println(false);

		if (s1.equals(s2))
			System.out.println(true);
		else
			System.out.println(false);

		// Mutable STring Classes
		StringBuffer sb1 = new StringBuffer("Hello");// MT - Safely
		sb1.append("World");
		System.out.println(sb1);

		StringBuffer sb2 = new StringBuffer("Hello");// MT - Safely
		sb2.append("World");
		System.out.println(sb2);

		if (sb1 == sb2)// comparing ref
			System.out.println(true);
		else
			System.out.println(false);

		if (sb1.equals(sb2))// equals() is no overrided for Buffer or builder
comparing ref
			System.out.println(true);
		else
			System.out.println(false);

		if (sb1.toString().equals(sb2.toString()))// comparing value trick
			System.out.println(true);
		else
			System.out.println(false);

	}
}
```

File Handling Glimpse

FileHandlingDemo.java

```
package com.jdk8.filehandling;

import java.io.File;
import java.io.IOException;

public class FileHandlingDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		String path = "C:sample.txt";

		File file = new File(path);
//		FileInputStream
//		Scanner
		if (file.exists())
			System.out.println(true);
		else
			System.out.println(false);
		try {
			file.createNewFile();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}

}
```