Dec Batch 2024 Notes

Important Download Links

JDK Download URL:	https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html (Please sign up with oracle first and then download the JDK)
Eclipse Download URL:	https://www.eclipse.org/downloads/packages/
Setup java in Windows OS Environment variable :	https://mkyong.com/java/how-to-set-java_home-on-windows-10/

What is Java?

Topic Name	Details
Java &JR Eclipse Installation **(JDK/JRE/JVM Basics)**	Eclipse download link: https://www.eclipse.org/downloads/ (Any version of Eclipse will be fine) Or https://www.eclipse.org/downloads/packages/ Eclipse is an IDE(Integrated Development Environment) to develop applications with the help of Java, Python, C/C++, Ruby etc. Note: You need to download JDk first then after eclipse because eclipse is going to check if jdk installed or not. - You can check which jdk version is installed in your system by running the below command in terminal. (for mac/windows users) ---> java -version compiler converts entire code to byte code which is runnable in any machines (Windows, Linux, Mac, Unix) ☐ JVM - Java Virtual Machine (Converts byte code to machine code) JRE - Java Run Time Environment (Just runs java programs) Java is not fully object oriented because it supports primitive data types like **char, byte, long, int, double** etc., which are not objects. Because in JAVA we use data types like int, float, double etc which are not object oriented, and of course is what opposite of OOP is. That is why **JAVA is not 100% object oriented**. Java - Platform Independent) JDK - Java Development Kit (JRE, JVM, Debugging, java docs)
	JDK - Java Development Kit comprises of = Tools + Compiler + Libs + JRE(Run Time env) + JVM(Platform Dependent) In a system, **JDK** helps in the **design and execution** of code. In contrast, **JRE** in a system simply helps in the **execution** of code, not in the design of it. And JVM is always platform dependent.
Java Docs Link	Java Docs is the official Link to view the Interfaces , Classes etc in a given Java package Ex: Java.util - package If we want to know all the classes, interfaces of this package then java docs is helpful

How Java Source Code will be converted to .class file.

JDK (Java Development Kit) is a software development kit that contains the **JRE (Java Runtime Environment)**, compiler, and tools for developing Java applications. JRE (Java Runtime Environment) is a set of tools and libraries that allow Java applications to run on a computer. JVM (Java Virtual Machine) is a virtual machine that executes Java code and provides runtime environment for Java applications.

In summary:

- JVM: runtime environment for executing Java code.
- JRE: includes JVM and libraries to run Java applications.
- JDK: includes JRE, compiler, and development tools for Java applications.

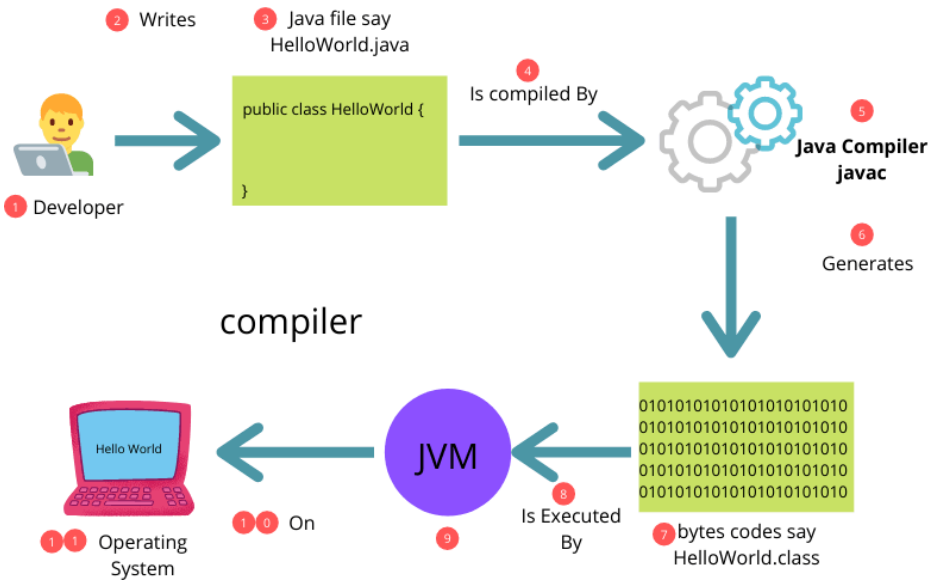
- [ ] The Java source code file (.java) is first compiled into Java bytecode (.class) using the Java compiler (javac). The bytecode is a platform-independent code that can be executed on any system with a Java Virtual Machine (JVM).

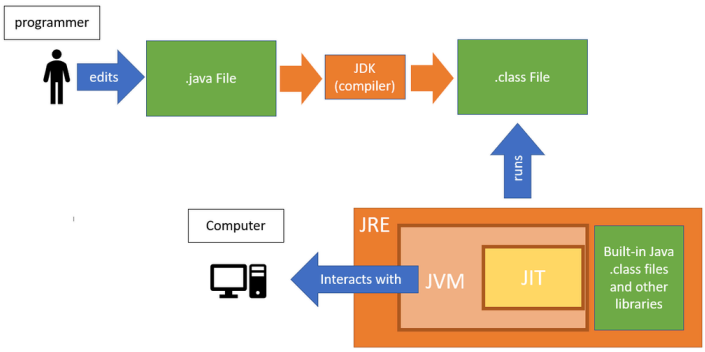
- When a Java program is executed, the JVM loads the bytecode into memory and runs it. The JVM interprets the bytecode and executes the corresponding machine code instructions on the underlying system. The JVM also provides a runtime environment for the program, which includes memory management, security, and other services.

- The JRE (Java Runtime Environment) contains the JVM and the Java class libraries, which provide a set of basic services and tools required to run a Java program. The JRE also includes the Java Plug-in, which enables Java applets to run in web browsers, and the Java Web Start, which enables Java applications to be launched directly from the web.

In summary:

- Java compiler converts .java to .class bytecode.
- JVM runs .class bytecode using JRE.
- JRE provides runtime environment and class libraries for Java programs.
- JVM is platform dependent.



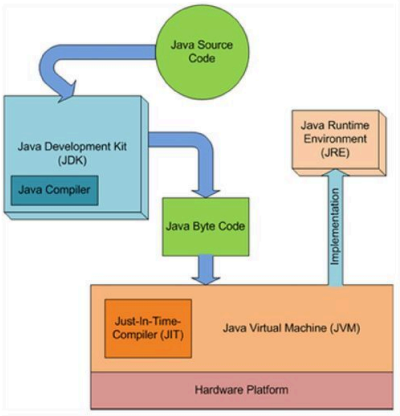


Difference between JDK/JRE/JVM/JIT

JVM Internals

Like a real computing machine, JVM has an instruction set and manipulates various memory areas at run time. Thus for different hardware platforms one has corresponding implementation of JVM available as vendor supplied JREs. It is common to implement a programming language using a virtual machine.

A Java virtual machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. From the point of view of a compiler, the Java Virtual Machine (JVM)is just another processor with an instruction set, Java bytecode, for which code can be generated. Life cycle is as follows, source code to byte code to be interpreted by the JRE and gets converted to the platform specific executable ones.



Jdk/jre/jit/jvm– Unit-I 4

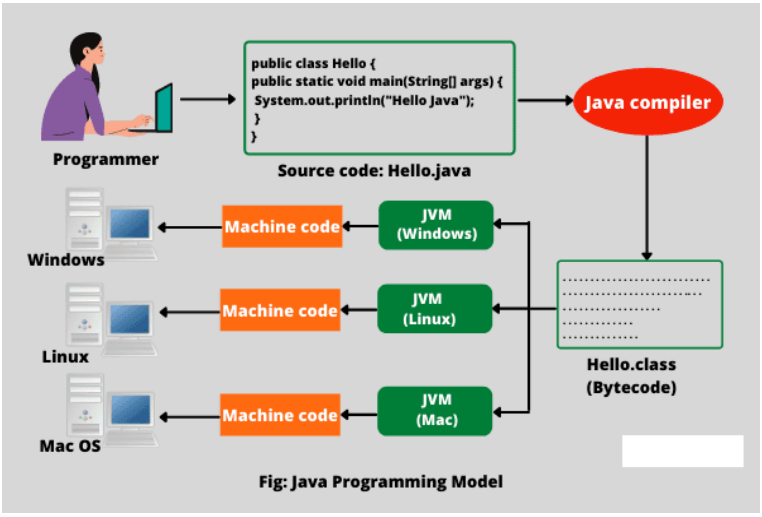
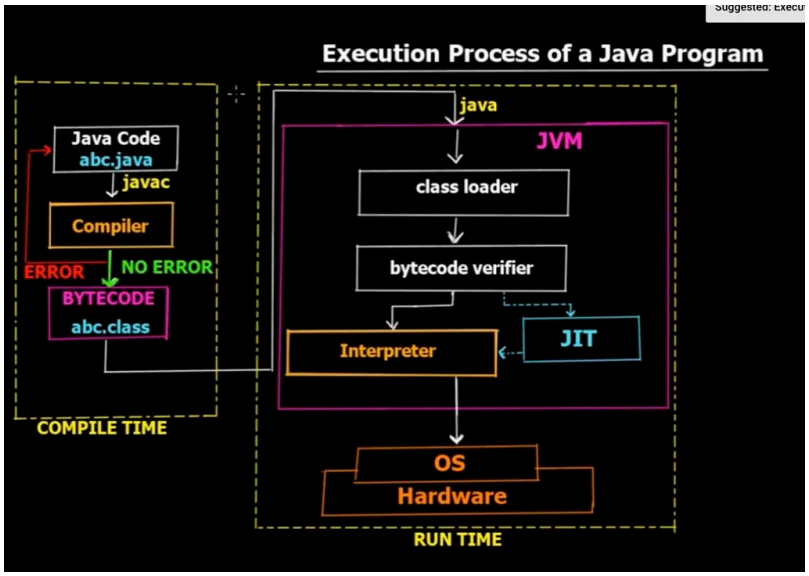


Fig: Java Programming Model



What is JIT (Just In Time)?

Just-In-Time (JIT) compilation is a part of the Java Runtime Environment (JRE) that improves the performance of Java applications by compiling bytecodes into native machine code at runtime.
Here's a breakdown of how it works and why it's beneficial:

1. **Bytecode**: Java programs are compiled into bytecode, which is a platform-independent code that can be executed by any Java Virtual Machine (JVM), regardless of the underlying hardware and operating system.
2. **JVM and Interpretation**: When a Java program is run, the JVM interprets the bytecode, converting it into machine code line by line as it is executed. This process is straightforward but not very efficient in terms of execution speed.
3. **JIT Compilation**: To improve performance, the JIT compiler kicks in. It compiles the entire bytecode into native machine code that a computer's processor can execute directly. This compilation happens on the fly, just in time for execution.
4. **Performance Improvement**: The native machine code runs much faster than interpreted bytecode. JIT compilation thus improves the performance of Java programs after an initial warm-up period.
5. **HotSpot**: This is a term often associated with JIT compilation in Java.
6. The JIT compiler in the JVM focuses on the "hot spots" of the code—sections that are executed frequently—and compiles them into native code to maximize performance benefits.

Byte Code vs Machine Code

- Bytecode and machine code are two different forms of code that a computer can execute.
- Bytecode is a platform-independent code that is generated by the Java compiler from Java source code. It is designed to be executed by the Java Virtual Machine (JVM) and can run on any system with a JVM installed. Bytecode is an intermediate form of code that is not directly executable by the computer's hardware, but it is optimized for efficient execution by the JVM.
- Machine code, on the other hand, is code that is directly executable by the computer's hardware. It is a low-level code that consists of binary instructions that correspond to specific operations the computer can perform. Machine code is platform-specific, meaning it is written for a specific type of processor and operating system and cannot run on other systems without modification.

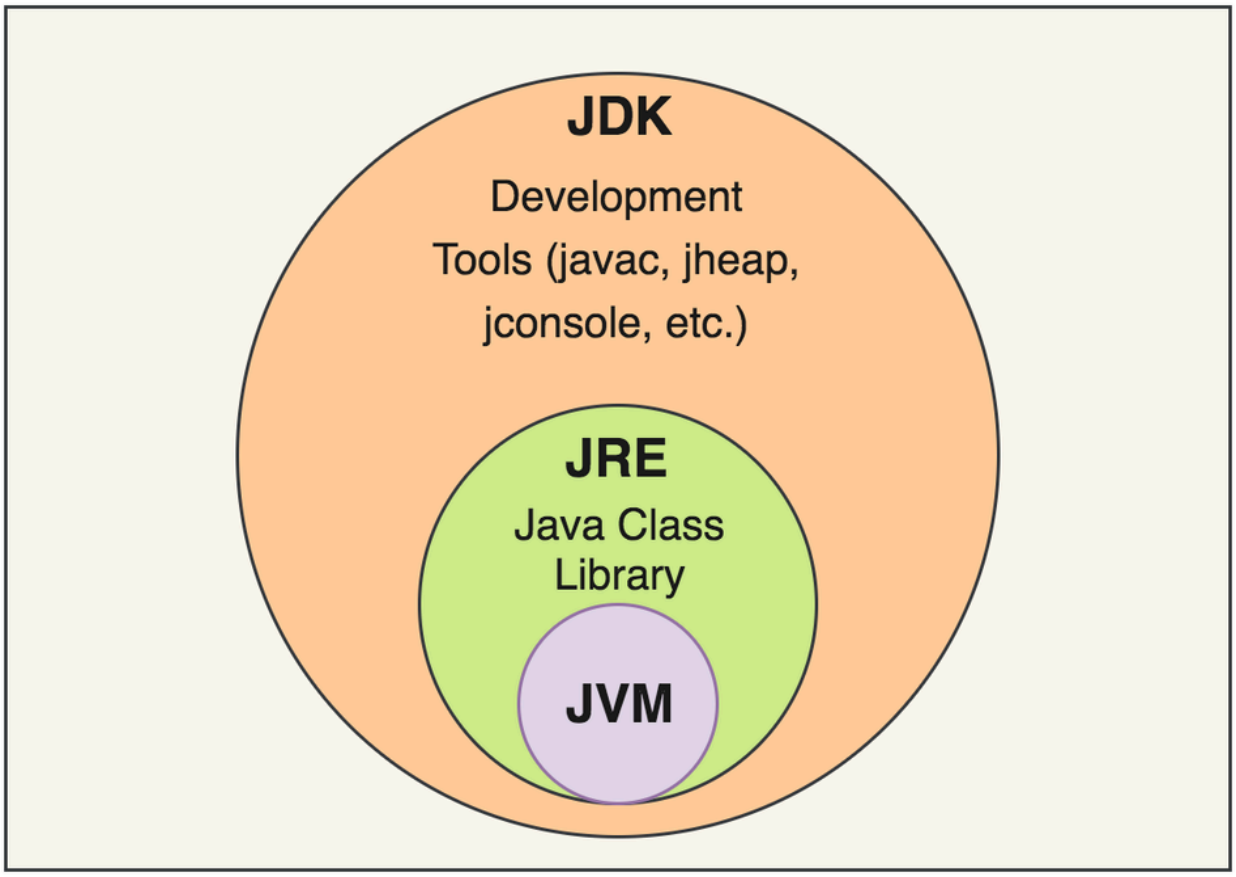
In summary:

- Bytecode is platform-independent, optimized for JVM, and executed by JVM.
- Machine code is platform-specific, low-level, and executed directly by the computer's hardware.

When a program is compiled in languages like C or C++, it's translated directly into machine code specific to the target hardware and operating system.
This makes these languages less portable than Java, as code compiled for one platform might not work on another without recompilation.

What is "WORA"	***Write once, run anywhere*** (***WORA***), or sometimes Write once, run everywhere (WORE) "Write once, run anywhere" (WORA) is a concept often associated with the Java programming language. It refers to the ability of Java to allow developers to write code on one platform and have it run on any platform that has a compatible Java Virtual Machine (JVM) implementation. This is made possible by compiling Java source code into bytecode, which is a platform-independent intermediate representation of the code. The bytecode is then executed by the JVM, which translates it into machine code specific to the underlying hardware and operating system.

Differences between JDK, JRE, and JVM:



Aspect	JDK (Java Development Kit)	JRE (Java Runtime Environment)	JVM (Java Virtual Machine)
Purpose	Development of Java applications, includes compiler and tools	Running Java applications	Execution of Java bytecode
Components	Compiler, debugger, libraries, development tools	Libraries and tools required to run Java applications	Runtime engine to execute Java bytecode
Includes JRE	Yes	Yes	No
Includes Compiler	Yes	No	No
Development Tools	Yes	No	No

Platform Independence	Yes	Yes	No (JVM is platform-specific)
Example Usage	Writing, compiling, and testing Java applications	Running standalone Java applications	Interpreting and/or compiling bytecode

Topic Name	Details
JDK Path Setup in Windows	**To set up the JDK path in a Windows machine, follow these steps:** 1. Locate the JDK installation directory on your system. It is usually in the form of "C:\Program Files\Java\jdk-version". 2. Right-click on "This PC" or "My Computer" and select "Properties". 3. In the System Properties window, click on the "Advanced" tab and then click on the "Environment Variables" button. 4. Create a new variable : JAVA_HOME and its value: C:\Program Files\Java\jdk-version 5. In the Environment Variables window, under "System Variables", scroll down to find the "Path" variable and click on it to edit. 6. Click on the "Edit" and add this value: %JAVA_HOME%\bin 7. Click "OK" to close all windows and save the changes. 8. Open a new Command Prompt window and run the command "java -version" to confirm that the JDK path has been set up correctly. This will set up the JDK path for the current user.
	You can follow this article to setup the JDK path : https://mkyong.com/java/how-to-set-java_home-on-windows-10/

Topic Name	Details
create a Java project in Eclipse	**To create a Java project in Eclipse:** 1. Open Eclipse. 2. Click on File > New > Java Project. 3. Enter a project name and click Finish. 4. Right-click on the src folder in the Package Explorer and create a new package 5. Right-click on the package, go to New > Class. 6. Enter a class name and select public static void main(String[] args) in the Methods section. 7. Click Finish. 8. Write your Java code in the main method. 9. Save and run the project by clicking Run > Run As > Java Application.
create a Java project in IntelliJ IDEA **Download InteiiJ community version:** https://www.jetbrains.com/idea/download	**To create a Java project in IntelliJ IDEA:** 1. Open IntelliJ IDEA. 2. Click on Create New Project. 3. Select Java from the list of project types. 4. Choose a project SDK or click New to add a JDK. 5. Enter a project name and click Finish. 6. Right-click on the src directory and select New > Java Class. 7. Enter a class name and select public static void main(String[] args) in the Methods section. 8. Click OK. 9. Write your Java code in the main method. 10. Save and run the project by clicking Run > Run.

JavaSession: Topic : Data_Types

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

--

Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

Startertutorials.com

Data Types:

There are two data types:

1. Primitive Data Type: These datatypes don't need any object creation and they occupy the pre-defined memory (Memory Size is fixed)
2. Non Primitive Data Type: Where memory size is not fixed and object creation is needed

Ex: String, Array, Interface, Class

Difference between primitive data types and non primitive data types

Topic Name	Details

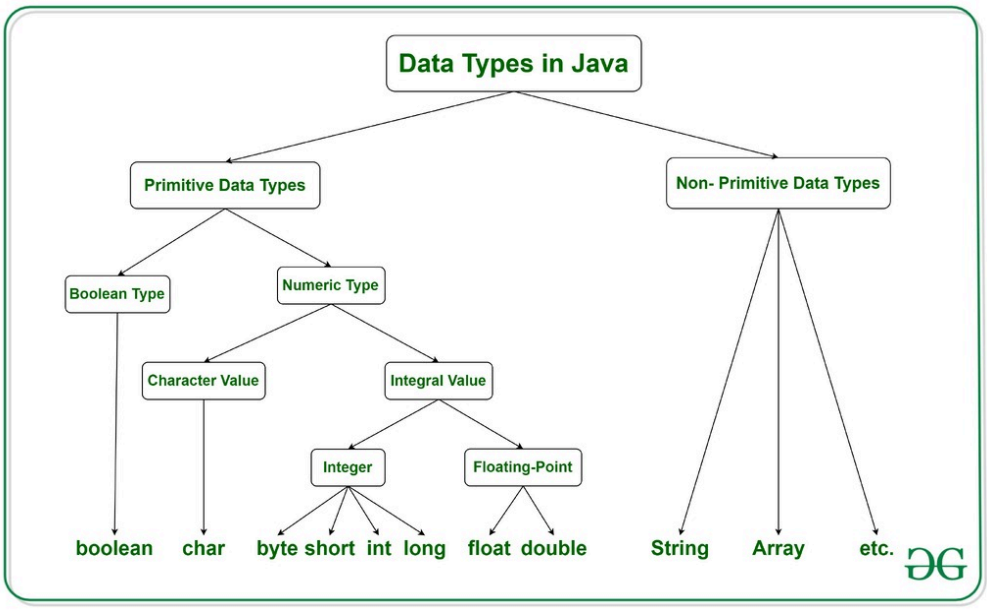
Difference between primitive data types and non primitive data types

- Primitive data types in Java are the basic data types that are built-in to the language and represent simple values. Examples of primitive data types are int, float, double, char, and boolean. They are stored directly in memory, have a fixed size and a range of values, and can only be assigned values directly.
- Non-primitive data types, also known as reference data types, are data types that are constructed from primitive data types. Examples of non-primitive data types are arrays, classes, and strings. They are stored as references in memory and have a variable size, meaning they can hold an unlimited amount of data.
- They cannot be assigned values directly, but instead hold a reference to the object in memory where the value is stored.
- In summary, the main difference between primitive and non-primitive data types is that primitive data types represent simple values directly, while non-primitive data types represent objects that are stored in memory and hold references to the data.

Difference between primitive data types and non primitive data types

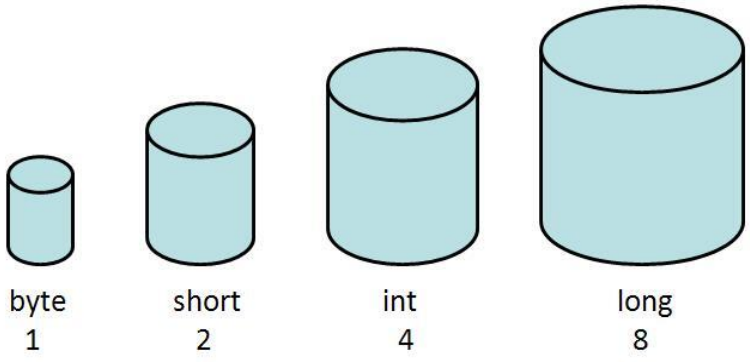
Properties	Primitive data types	Objects
Origin	Pre-defined data types	User-defined data types
Stored structure	Stored in a stack	Reference variable is stored in stack and the original object is stored in heap
When copied	Two different variables is created along with different assignment(only values are same)	Two reference variable is created but both are pointing to the same object on the heap
When changes are made in the copied variable	Change does not reflect in the original ones.	Changes reflected in the original ones.
Default value	Primitive datatypes do not have null as default value	The default value for the reference variable is null
Example	byte, short, int, long, float, double, char, boolean	array, string class, interface etc.

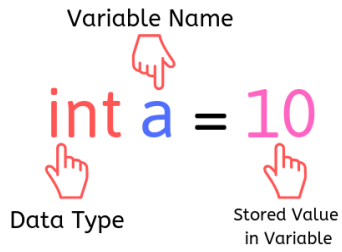
Java Data Types:

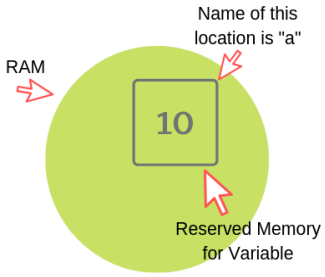


- 1 byte = 8 bits

How Memory allocated for a primitive data type:
ex: int a = 10;







Topic Name	Details
Why do we need to use L/l as a suffix for long numbers??	long myLong = 123456789L; The L suffix is required to write when the values to be stored is out of range of integer to tell compiler its long value because by default treated as int. We can write L for small values also but its not mandatory. Why do we need to use L/l as a suffix for long numbers?? Ans: This notation is used to make it explicit to the compiler and to other developers that the value should be treated as a **long** and not as an **int**. This is important when the value being assigned is outside the range of an **int** data type. Using the **L** notation for **long** data types is considered a best practice to ensure that the code is clear and readable.
Why do we need to use f as a suffix for float numbers??	float myFloat = 1.23f; This notation is used to make it explicit to the compiler and to other developers that the value should be treated as a **float** and not as a **double**. This is important when memory usage and precision need to be optimized, and when a smaller range of values is acceptable. Using the **f** notation for **float** data types is considered a best practice to ensure that the code is clear and readable.

Java naming conventions

Topic Name	Details

Java naming conventions

Java naming conventions are a set of rules for naming identifiers (such as variables, methods, classes, etc.) in the Java programming language.
The conventions help to keep the code consistent and easy to read, understand, and maintain. Some common Java naming conventions are:

1. Class names start with an uppercase letter and use PascalCase (e.g. EmployeeRecord, PaymentProcessor).
2. Method names start with a lowercase letter and use CamelCase (e.g. getEmployeeData, processPayment).
3. Variable names start with a lowercase letter and use CamelCase (e.g. employeeName, paymentAmount).
4. Constants are written in uppercase letters with words separated by an underscore (e.g. MAX_SIZE, MIN_WAGE).
5. Package names are all lowercase (e.g. java.util, com.example.payroll).
6. Interfaces start with an uppercase "I" followed by CamelCase (e.g. IEmployee, IPayment).
7. Acronyms are treated as a single word in names (e.g. HttpServlet, JdbcTemplate).
8. Enum names use CamelCase (e.g. Month, PaymentMethod).
9. Annotation names end with "Annotation" (e.g. OverrideAnnotation, DeprecatedAnnotation).
10. Use meaningful and descriptive names for identifiers, avoiding abbreviations and short names (e.g. "empName" instead of "en").yy,
11. Avoid using reserved words as identifier names (e.g. "class", "int").
12. Do not use Hungarian Notation, where the first few letters of an identifier indicate its type (e.g. "strName", "intAge").
13. Avoid using single-letter names except for temporary variables used within a small scope.
14. Try to keep the length of names reasonable, a
15. voiding excessively long or short names.
16. Use camelCase for local variables and method parameters.
17. Avoid using abbreviations unless they are widely recognized and well-established (e.g. "URL").
18. Prefix instance variables with "m" or "m_" (e.g. m_employeeName).
19. Prefix static variables with "s" or "s_" (e.g. s_maxCount).
20. Prefix boolean variables with "is" or "has" (e.g. isEmployee, hasData).
21. Use "get" and "set" prefixes for getter and setter methods (e.g. getEmployeeName, setEmployeeName).
22. Use the "toString" method to return a human-readable representation of an object.
23. Use a consistent naming convention throughout the project to maintain consistency and readability.

ASCII Table

Dec	Oct	Hex	C	Dec	Oct	Hex	C	Dec	Oct	Hex	C	Dec	Oct	Hex	C
0	0	0	^@	32	40	20		64	100	40	@	96	140	60	`
1	1	1	^A	33	41	21	!	65	101	41	A	97	141	61	a
2	2	2	^B	34	42	22	"	66	102	42	B	98	142	62	b
3	3	3	^C	35	43	23	#	67	103	43	C	99	143	63	c
4	4	4	^D	36	44	24	$	68	104	44	D	100	144	64	d
5	5	5	^E	37	45	25	%	69	105	45	E	101	145	65	e
6	6	6	^F	38	46	26	&	70	106	46	F	102	146	66	f
7	7	7	^G	39	47	27	'	71	107	47	G	103	147	67	g
8	10	8	^H	40	50	28	(	72	110	48	H	104	150	68	h
9	11	9	^I	41	51	29	)	73	111	49	I	105	151	69	i
10	12	a	^J	42	52	2a	*	74	112	4a	J	106	152	6a	j
11	13	b	^K	43	53	2b	+	75	113	4b	K	107	153	6b	k
12	14	c	^L	44	54	2c	,	76	114	4c	L	108	154	6c	l
13	15	d	^M	45	55	2d	-	77	115	4d	M	109	155	6d	m
14	16	e	^N	46	56	2e	.	78	116	4e	N	110	156	6e	n
15	17	f	^O	47	57	2f	/	79	117	4f	O	111	157	6f	o
16	20	10	^P	48	60	30	0	80	120	50	P	112	160	70	p
17	21	11	^Q	49	61	31	1	81	121	51	Q	113	161	71	q
18	22	12	^R	50	62	32	2	82	122	52	R	114	162	72	r
19	23	13	^S	51	63	33	3	83	123	53	S	115	163	73	s
20	24	14	^T	52	64	34	4	84	124	54	T	116	164	74	t
21	25	15	^U	53	65	35	5	85	125	55	U	117	165	75	u
22	26	16	^V	54	66	36	6	86	126	56	V	118	166	76	v
23	27	17	^W	55	67	37	7	87	127	57	W	119	167	77	w
24	30	18	^X	56	70	38	8	88	130	58	X	120	170	78	x
25	31	19	^Y	57	71	39	9	89	131	59	Y	121	171	79	y
26	32	1a	^Z	58	72	3a	:	90	132	5a	Z	122	172	7a	z
27	33	1b	^[	59	73	3b	;	91	133	5b	[	123	173	7b	{
28	34	1c	^\	60	74	3c	<	92	134	5c	\	124	174	7c	|
29	35	1d	^]	61	75	3d	=	93	135	5d	]	125	175	7d	}
30	36	1e	^^	62	76	3e	>	94	136	5e	^	126	176	7e	~
31	37	1f	^_	63	77	3f	?	95	137	5f	_	127	177	7f	

ASCII Range for Letters and Numbers:	For lowercase letters (a-z), the ASCII range is 97 to 122. For uppercase letters(A-Z), the ASCII range is 65 to 90. For numerals(0-9), the ASCII range is 48 to 57.
Unicode and ASCII Characters	Unicode and ASCII are character encoding standards used to represent text characters as numerical values. They provide a way for computers to understand and exchange textual information by assigning a unique number (code point) to each character. **ASCII (American Standard Code for Information Interchange):** ASCII is one of the earliest character encoding standards and is primarily used for representing English characters and control characters in computers. It uses 7 bits to represent 128 different characters, including letters, numbers, punctuation, and control characters. Here are a few ASCII character examples along with their decimal and binary representations: • Character: 'A', Decimal: 65, Binary: 01000001 • Character: 'a', Decimal: 97, Binary: 01100001 • Character: '0', Decimal: 48, Binary: 00110000 • Character: '!', Decimal: 33, Binary: 00100001 **Unicode:** Unicode is a more comprehensive character encoding standard that aims to cover characters from all writing systems in the world, including various languages, symbols, and emojis. It uses a larger number of bits (usually 16 or 32 bits) to represent a much wider range of characters compared to ASCII. Here are a few Unicode character examples along with their hexadecimal and binary representations: • Character: 'A', Unicode Hex: U+0041, Binary: 01000001 (Equivalent to ASCII for ASCII characters) • Character: '€', Unicode Hex: U+20AC, Binary: 00100000 10101100 (Euro sign) • Character: 'न', Unicode Hex: U+0928, Binary: 10010010 10001000 (Devanagari letter 'na' in Hindi) • Character: '😊', Unicode Hex: U+1F60A, Binary: 00011111 01100000 00101000 00101010 (Smiling face emoji) Unicode provides a way to represent a vast array of characters from various languages and cultural contexts, making it more suitable for global communication and content.

All Divide By Zero cases

Cases	Results	Example	Tricky examples
Any integer divided by 0	Arithmetic Exception	9/0.	0/0 is also exception
0 divided by Any Integer.	0	0/9=0	
Any integer or Float divided by 0 or 0.0	Infinity	2.5/0,2.5/0.0	9/0.0 = infinity 9.5/0.0=infinity
int divided by int	integer	5/2=2	
Int divided by float or float divide by integer	float	5.0/2=2.5 , 5/2.0=2.5	
Int 0 divided by float 0 or float 0 divided by int 0	NaN- Not a number	0/0.0=naN 0.0/0=naN 0.0/0.0=naN	

```
public class
ArithMeticOperations {

public static void
main(String[] args) {

System.out.println(10 + 20);

System.out.println(20 / 10);

System.out.println(9 / 2);// 4
System.out.println(9.0 / 2);//
4.5
System.out.println(9 / 2.0);//
4.5
System.out.println(9.0 /
2.0);// 4.5

System.out.println(9.0 /
3.0);// 3.0

// System.out.println(9/0);//AE
System.out.println(0 / 9);// 0

System.out.println(9.0 / 0);//
Infinity
System.out.println(9 / 0.0);//
Infinity
System.out.println(9.0 /
0.0);// Infinity

// System.out.println(0/0);//AE

System.out.println(0.0 / 0);//
NaN - not a number
System.out.println(0 / 0.0);//
NaN
System.out.println(0.0 /
0.0);// NaN

//
System.out.println('a'/0);//AE

System.out.println(3.4f / 0);//
Infinity

}

}
```

// Notes on Arithmetic Operations in Java:

1. Basic Integer Division:
   - Integer division (when both operands are int):
     * 9/2 = 4 (decimal part is truncated)
     * 20/10 = 2 (exact division)

2. Floating Point Division Rules:
   a) When at least one operand is floating-point:
     * 9.0/2 = 4.5
     * 9/2.0 = 4.5
     * 9.0/2.0 = 4.5
     * 9.0/3.0 = 3.0

   The result maintains decimal precision when any operand is floating-point

3. Division by Zero Cases:
   a) Integer Division by Zero:
     * 9/0 - Throws ArithmeticException (commented in code)
     * 0/9 = 0 (valid operation)

   b) Floating Point Division by Zero:
     * 9.0/0 = Infinity
     * 9/0.0 = Infinity
     * 9.0/0.0 = Infinity

   c) Zero divided by Zero:
     * 0/0 - Throws ArithmeticException (for integers)
     * 0.0/0 = NaN (Not a Number)
     * 0/0.0 = NaN
     * 0.0/0.0 = NaN

4. Special Cases:
   - Float division by zero:
     * 3.4f/0 = Infinity

   - Character division by zero:
     * 'a'/0 - Throws ArithmeticException (commented in code)

Key Rules:
1. Any division by zero with integers throws ArithmeticException
2. Division by zero with floating-point numbers results in Infinity
3. Zero divided by zero with floating-point numbers results in NaN
4. Including any floating-point number in division preserves decimal precision

String Concatenation	In Java, a **String** is a in-built class that represents a sequence of characters. It is one of the most widely used classes in Java and is used to represent text data. Ex: **String s = "Hello World";** **String msg = "This is my java code";** ***String Concatenation:*** String concatenation in Java refers to the operation of combining two or more strings into a single string. This can be done using the "+" operator. The "+" operator can be used to concatenate not just strings, but also other data types such as numbers, which will be automatically converted to strings before concatenation. **Examples of string concatenation using the "+" operator:** ***int*** **a** = 10, int **b** = 20; • *System.out.println* (a+b); - Output= **30** • Here '+' sign is behave as Arithmetic Operator • *System.out.println* (a+b+" test data"); Output= "**30test data**" • Here 1st '+' sign is behave as Arithmetic Operator, while 2nd '+' sign Concatenation Operator • *System.out.println* ("test data" + a); Output= "**test data10**" • Here '+' sign is behave as Concatenation Operator • *System.out.println* ("test data" + a+b) Output= "**test data1020**" • Here '+' sign is behave as Concatenation Operator • **Note** - Since execution happens left to right , the placement of the string matters while printing to get the correct data • **System** is a class ***out*** is variable and **println()** is method.

String Concatenation:

```java
public class StringConcatenation {

public static void main(String[] args) {

int a = 10;
int b = 20;

System.out.println("the value of a is :" + a);
System.out.println("the value of b is :" + b);
System.out.println("the sum is : " + (a + b));

double c = 12.33;
double d = 12.44;

String x = "hello";
String y = "world";

// helloworld989930

char ch = 'a';
char th = 'b';
System.out.println(ch + th);// 97+98=195
char yh = 99;// c

System.out.println(x + y + (byte) th + (byte) yh + (a + b));// helloworld989930

System.out.println(a + b);
System.out.println(x + y);// helloworld
System.out.println(x + a);// hello10
System.out.println(a + b + x + y);// 30helloworld
System.out.println(x + y + a + b);// helloworld1020
System.out.println(x + y + (a + b));// helloworld30

System.out.println(a + b + x + y + a + b);// 30helloworld1020

System.out.println(c + d + x + y);// 24.77helloworld
System.out.println("------");
System.out.println(x + y + c + d);// helloworld12.3312.44
System.out.println("------");

System.out.println(x + y + c + d + a + b);// helloworld12.3312.441020
System.out.println(a + b + c + d + x + y + a + b);// 54.77helloworld1020

System.out.println(ch + th + x + y);// 195helloworld
System.out.println(x + y + ch + th + a + b);// helloworldab1020

System.out.println(x + y + (ch + th) + a + b);// helloworld1951020
System.out.println(x + y + (ch + th + a + b));// helloworld225

System.out.println(a + b + yh);// 30+99=129

System.out.println(ch + th + yh);// 294

System.out.println(ch + th + yh + x);// 294hello

char naveen = 100;// 979899100
System.out.println(naveen);// d
```

//Notes on String Concatenation in Java:

1. Variable Declarations:
   - Integers: a = 10, b = 20
   - Doubles: c = 12.33, d = 12.44
   - Strings: x = "hello", y = "world"
   - Characters: ch = 'a', th = 'b', yh = 99 (represents 'c')

2. Character Operations:
   - Characters are stored as ASCII values internally
   - ch ('a') has ASCII value 97
   - th ('b') has ASCII value 98
   - When chars are added: ch + th = 97 + 98 = 195

3. String Concatenation Rules:
   a) Left to Right Evaluation:
      - If string appears first: treats everything after as string concatenation
        Example: x + y + a + b = "helloworld1020"

      - If numbers appear first: performs arithmetic until it hits a string
        Example: a + b + x + y = "30helloworld"

   b) Parentheses Override:
      - Operations inside parentheses are evaluated first
      Example: x + y + (a + b) = "helloworld30"

4. Mixed Data Type Operations:
   - With doubles:
     * c + d + x + y = "24.77helloworld"
     * x + y + c + d = "helloworld12.3312.44"

   - With characters:
     * When chars are used with strings: treated as characters
       Example: x + y + ch + th = "helloworldab"

     * When chars are added before string: treated as numbers
       Example: ch + th + x + y = "195helloworld"

5. Special Cases:
   - Character assignment with numbers:
     * char naveen = 100 results in 'd' (ASCII value 100)

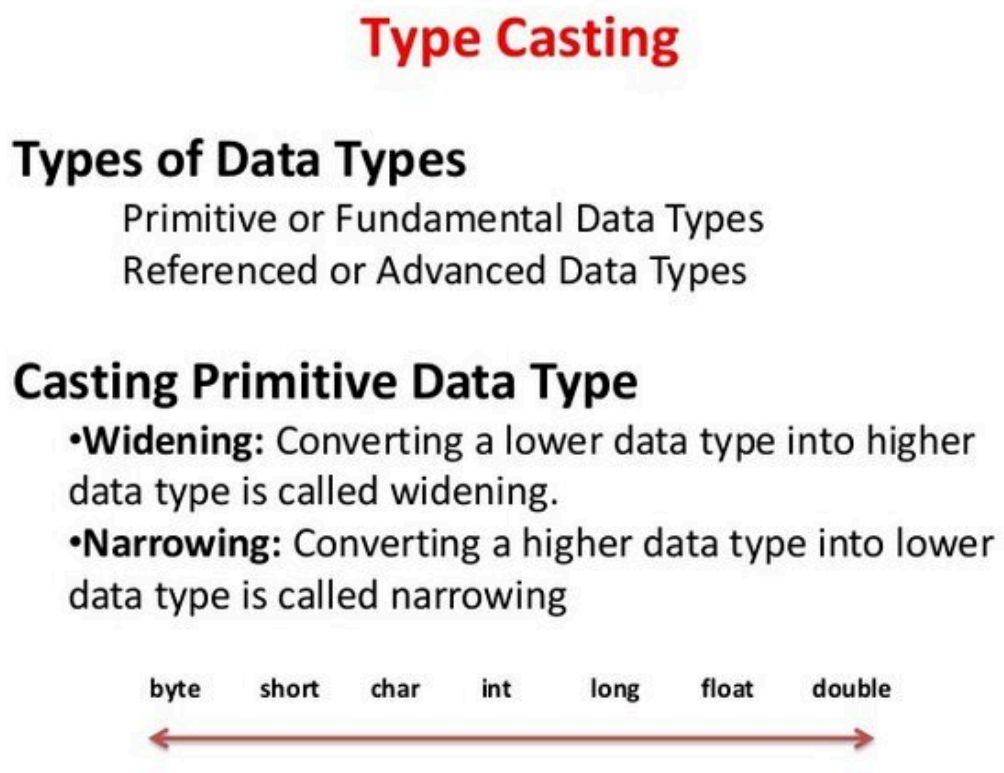
   - Division with concatenation:
     * x + y + b/a = "helloworld2" (20/10 = 2)

6. Type Casting:
   - (byte)th converts character to its byte value
   - Used in complex concatenations to ensure proper numeric representation

Key Concept: The position of strings in concatenation operations determines whether numeric values are added or concatenated as strings.

```
}
```

Type Casting:

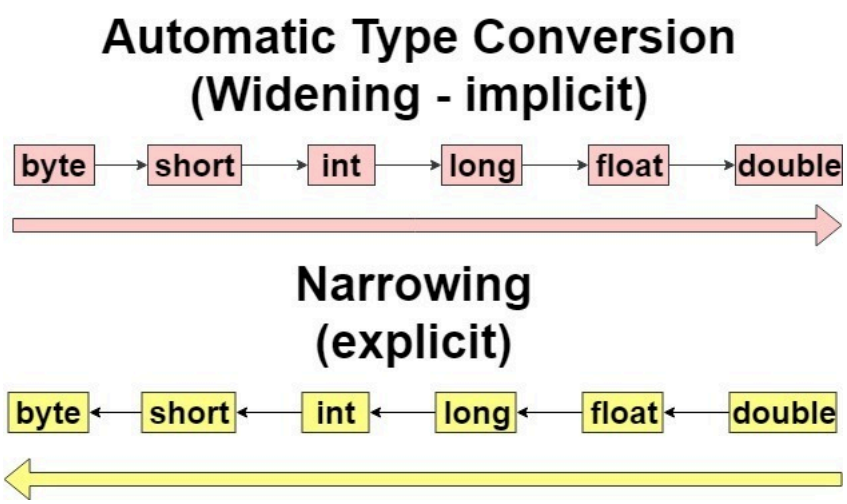


Type Casting:		

Widening or Automatic Type Conversion

In Java, when assigning a value of a smaller data type to a larger data type, Java automatically converts the smaller data type into the larger one. This is known as automatic or widening type conversion and it doesn't require explicit casting. The data types are arranged from smallest to largest as follows:

1. **byte** - This is an 8-bit signed integer.
2. **short** - This is a 16-bit signed integer.
3. **int** - This is a 32-bit signed integer.
4. **long** - This is a 64-bit signed integer.
5. **float** - This is a 32-bit floating point.
6. **double** - This is a 64-bit floating point.



For example, here's how automatic conversion works:

```
byte byteValue = 42;
short shortValue = byteValue; // automatic conversion from byte to short
int intValue = shortValue; // automatic conversion from short to int
long longValue = intValue; // automatic conversion from int to long
float floatValue = longValue; // automatic conversion from long to float
double doubleValue = floatValue; // automatic conversion from float to double
```

Narrowing or Explicit Type Conversion

Conversely, when assigning a value of a larger data type to a smaller data type, Java requires explicit casting. This is because narrowing conversion could potentially lead to data loss, and Java needs you to confirm you understand this risk by writing a cast. The order for narrowing conversion is the reverse of widening conversion.

Here's an example of explicit casting:

```
double doubleValue = 42.0;
float floatValue = (float) doubleValue; // explicit cast from double to float
long longValue = (long) floatValue; // explicit cast from float to long
int intValue = (int) longValue; // explicit cast from long to int
short shortValue = (short) intValue; // explicit cast from int to short
byte byteValue = (byte) shortValue; // explicit cast from short to byte
```

Rules and Considerations

- Widening conversions do not lose information about the overall magnitude of a numeric value.
- Narrowing conversions may lose information about the overall magnitude of a numeric value and may also lose precision.
- When converting from floating-point to integral types, the fractional part is truncated.
- Casting in the opposite direction (narrowing conversion) requires explicit casts.
- Java does not allow casting boolean values to any numeric type and vice versa.

Remember that while widening conversions are generally safe, narrowing conversions

		carefully to avoid data loss.

Widening Conversion Examples	// Example 1: From integer to floating-point types
	int myInt = 100;
	long myLong = myInt; // No casting required
	float myFloat = myLong; // No casting required
	double myDouble = myInt; // No casting required - int to double is also widening
	
	System.out.println("Int value: " + myInt);
	System.out.println("Long value: " + myLong);
	System.out.println("Float value: " + myFloat);
	System.out.println("Double value: " + myDouble);
	
	// Example 2: From char to integral types
	char myChar = 'A';
	int charToInt = myChar; // char to int is widening since char is 16-bit and int is 32-bit
	
	System.out.println("Char value: " + myChar);
	System.out.println("Int value from char: " + charToInt); // Outputs 65, which is ASCII value of 'A'	

Narrowing Conversion Examples	// Example 1: From floating-point to integral types
	double myDoubleValue = 9.78;
	int myIntValue = (int) myDoubleValue; // Explicit casting is required
	
	System.out.println("Double value: " + myDoubleValue);
	System.out.println("Converted Integer value: " + myIntValue); // Outputs 9, fractional part is lost
	
	// Example 2: From long to smaller types
	long myLongValue = 123456789L;
	int myIntFromLong = (int) myLongValue; // Explicit casting is required
	short myShortFromLong = (short) myLongValue; // Explicit casting is required
	byte myByteFromLong = (byte) myLongValue; // Explicit casting is required
	
	System.out.println("Long value: " + myLongValue);
	System.out.println("Converted Int value: " + myIntFromLong); // Potential loss of magnitude
	System.out.println("Converted Short value: " + myShortFromLong); // Potential loss of magnitude
	System.out.println("Converted Byte value: " + myByteFromLong); // Potential loss of magnitude
	
	// Example 3: From char to byte
	char myCharacter = 'F';
	byte myByteFromChar = (byte) myCharacter; // Explicit casting is required
	
	System.out.println("Character value: " + myCharacter);
	System.out.println("Converted Byte value: " + myByteFromChar); // Outputs 70, which is ASCII value of 'F'	

Assignments:

Data Types:

1. Write a Java program to add two strings:

String a = “Hello“;
String b = “Naveen K“
Expected Output :
Hello Naveen K

2. Write a Java program to print the sum of two numbers.
Test Data:
74 + 36
Expected Output:
110

3. Write a Java program to print the division of two numbers.

k = 50/3

- Expected Output :

43
16 — correct one
19
13

4. Write a Java program to compute the specified expressions and print the output. Go to the editor.
Test Data:

- ((25.5 * 3.5 - 3.5 * 3.5) / (40.5 - 4.5))

Expected Output
2.138888888888889

5. Try to concat "Hello Selenium" with a character 't'.

6. Create three int variables having value
s like : 100, 200, 3400. Add them and concatenate and generate this output String :
"Your Total amount is. 3700".

7. Print the ASCII value of the character 'h'.
8. Write a program to add 3 to the ASCII value of the character 'd' and print the equivalent character.
9. Write a program to find the square of the number 3.9.

Incremental/Decremental Operators:

1) What will be the output of the following program?

2) Guess the output of the following program?

3) What will be the output of the below program?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      int i=0;

      i = i++ - --i + ++i - i--;

      System.out.println(i);

   }
}
```

4) Is the below program written correctly?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
        {
        boolean b = true;

        b++;

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

5) What will be the output of the below program?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      int i=1, j=2, k=3;

      int m = i-- - j-- - k--;

      System.out.println("i="+i);
      System.out.println("j="+j);
      System.out.println("k="+k);
      System.out.println("m="+m);
   }
}
```

6) What will be the output of the following program?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      int a=1, b=2;
      System.out.println(--b - ++a + ++b - --a);

   }
```

7) What will be the value of i, j and k in the below program?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
          int i=19, j=29, k;

         int m = i-- - j-- - k--;

         System.out.println("i="+i);
         System.out.println("j="+j);
         System.out.println("k="+k);
   }
}
```

8) What is wrong with the below program? Why it is showing a compilation error?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
         int I = 11;
      int j = --(i++);
   }
}
```

9) Guess the value of *p* in the below program?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      int m = 0, n = 0;

      int p = --m * --n * n-- * m--;

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

10) What will be the output of the following program?

```
public class IncrementDecrementQuiz
{
     public static void main(String[] args)
   {
      int a=1;

      a = a++ + ++a * --a - a--;

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

11) What will be the outcome of the below program?

```
public class IncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      int a = 11++;
```

```
    }
}
```

12) What will be the outcome of the below program?

```
public class JavaIncrementDecrementQuiz
{
  public static void main(String[] args)
  {
        int ch = 'A';//65

        System.out.println(ch++);//65
  }
}
```

13) **What will be the outcome of the below program?**

```
public class JavaIncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      char ch = 'A';

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

14) **What will be the outcome of the below program?**

```
public class JavaIncrementDecrementQuiz
{
   public static void main(String[] args)
   {
      double d = 1.5, D = 2.5;

      System.out.println(d++ + ++D);
   }
}
```

Comparison operators in Java

Topic Name	Details

Comparison operators in Java	Comparison operators in Java. Following are symbols used for comparison operators: 1. == equals to 2. > greater than 3. >= greater than equal to 4. < less than 5. <= less than equal to 6. != not equal String comparison can be done by using method *equals:* String 8 = "chrome"; if(browser.***equals***("chrome")) //here we are comparing browser variable with value in the brackets (). Conditions - decision making: 1. if 2. if else 3. if else if 4. switch case

Switch case	1- Data type allowed in Switch - **Char**, **Byte , short , Int and string only**
	2- under default BREAK statement is not mandatory but as a good practice we should keep it .
	Real time examples -
	 
	//
	//real time of switch case use cases:
	//1. cross browser logic
	//2. ecomm - multiple countries - langauge
	//3. multi environment: DEV, QA, STAGE, UAT, PROD
	//4. API methods: GET, POST, PUT, DELETE
	//5. DropDown: Single, Multiple, All
	//6. Payment Methods: CC, UPI, PAYPAL, Xoom, WU, BANK
	
	//use cases of switch cases:
	//cross browser logic: ch/ff/sf/edge/ie
	//Ecomm App: Product Categories
	//RBAC: User roles: admin, customer, partner, seller, vendor, distributor
	//Cross OS/Platforms: MAc,Windows, Linux
	//Multi Environment Setup: dev, qa, stage, uat, prod
	//Payment: UPI, CC, Paypal, NetBanking, other
	//Month: 12 month: Jan to Dec
	//Uber Booking:
	//Web Page: Link, Image, text field, button, checkbox, radio button

switch case	1. Runtime execution time it is go to directly into case where have to jumped. 2. Compiler decides that where to jumped. 3. If break is not there it will go all the case statements . Note:- We can use switch case statement only in integer (byte, short and int) , char and String but not long, float, double and boolean.

if else if condition	1. Check all the condition until not satisfied 2. We can not use break keyword in if condition. 3. This condition is only significant for early conditions are satisfied. 4. If last condition satisfied it will check all condition which is waste of time.

Switch Case Statements:

Switch Case - ***Note - works with Char/Int, short, byte/String*** ***Not Applicable: long, float, double, boolean***	Details

Example:
 	
`public class SwitchCaseConcept {`
`public static void main(String[] args) {`
`String browser = "chrome";`
`switch (browser) {`
`case "chrome":`
`System.out.println("launch chrome");`
`break;`
`case "firefox":`
`System.out.println("launch firefox");`
`break;`
`case "safari":`
`System.out.println("launch safari");`
`break;`
`case "ie":`
`System.out.println("launch ie");`
`break;`
`case "opera":`
`System.out.println("launch opera");`
`break;`
`default:`
`System.out.println("please pass the right browser.....");`
`break;`
`}`
`}`
`}`
	This is a Java program which demonstrates the usage of switch-case statement.
	1. **public class SwitchCaseConcept**: The class definition. The class name is "SwitchCaseConcept".
	2. **public static void main(String[] args)**: The main method is the starting point of the Java program.
	3. **String browser = "chrome";**: A string variable named "browser" is declared and assigned the value "chrome".
	4. **switch (browser) {**: The switch statement checks the value of the "browser" variable.
	5. **case "chrome":**: If the value of the "browser" variable is "chrome", the code inside this case block is executed.
	6. **System.out.println("launch chrome");**: This line prints "launch chrome" to the console.
	7. **break;**: The "break" statement terminates the switch statement, and no other case block is executed.
	8. **case "firefox":**, **case "safari":**, **case "ie":**, **case "opera":**: These are similar to case "chrome". If the value of "browser" matches any of these, the corresponding print statement is executed.
	9. **default:**: The "default" block is executed if none of the cases match the value of the "browser" variable.
	10. **System.out.println("please pass the right browser.....");**: This line prints "please pass the right browser....." to the console.
	11. **break;**: The "break" statement terminates the switch statement.

1. Limitation of Switch case is, it is only applicable for integer/String/Char/(byte,short,int), string and Character values and cannot be used for Boolean, Float, Double values. 2. Switch Case logic can be best use for Cross browsers, environment selection, user access-based role, Choosing payment options etc 3. Switch case - does not work with Float / Double 4. Switch case does not work with integer type-Long 5. Switch case does not work with boolean 6. Whenever there is no break statement involved in a switch case block. All the statements are executed even if the test expression is not satisfied 7. Does not work with variable conditions. 8. We can not use a continue with the switch statement 9. Java is case sensitive language ,every keyword in java should be lower case only.	

Examples of Switch- Case and if-else	Details

Check the given char is vowel or consonant.	**using if-else:**
	char c = 'i';
	if(c=='a' || c=='e' || c=='i' || c=='o' || c=='u') {
	System.***out***.println("Vowel");
	}
	else {
	System.***out***.println("consonant");
	}
	
	==========
	using switch-case:
	char alphabet = 'z';
	switch (alphabet) {
	case 'a':
	System.***out***.println("this is vowel");
	break;
	case 'e':
	System.***out***.println("this is vowel");
	break;
	case 'i':
	System.***out***.println("this is vowel");
	break;
	case 'o':
	System.***out***.println("this is vowel");
	break;
	case 'u':
	System.***out***.println("this is vowel");
	break;
	
	default:
	System.***out***.println("this is a consonant");
	break;
	}

if-else/if-else if/nested if-else

1. if Statement:

- The **if** statement is used to execute a block of code only if a certain condition is true.
- If the condition is false, the code block is skipped.

Example:

```
int num = 10;
if (num > 5) {
   System.out.println("Number is greater than 5.");
}
```

2. if-else Statement:

- The **if-else** statement allows you to execute one block of code when a condition is true and another block when the condition is false.
- Only one of the code blocks will be executed.

Example:

```
int num = 3;
```

```
} else {
    System.out.println("Number is not greater than 5.");
}
```

3. if-else if Statement:

- The **if-else if** statement allows you to check multiple conditions in sequence and execute the code block corresponding to the first true condition.
- If none of the conditions are true, the else block (if present) will be executed.

Example:

```
int num = 7;
if (num < 5) {
    System.out.println("Number is less than 5.");
} else if (num < 10) {
    System.out.println("Number is between 5 and 10.");
} else {
    System.out.println("Number is greater than or equal to 10.");
}
```

4. Nested if Statements:

- You can have an **if** statement within another **if** statement. This is called nested **if** statements.
- Nested **if** statements allow you to check conditions in a hierarchical manner.

Example:

```
int x = 10;
int y = 5;
if (x > 0) {
    if (y > 0) {
        System.out.println("Both x and y are positive.");
    } else {
        System.out.println("x is positive, but y is not.");
    }
} else {
    System.out.println("x is not positive.");
}
```

Remember these points when using conditional statements in Java:

- Conditions should be boolean expressions (**true** or **false**).
- Use curly braces {} to define the scope of the code blocks.
- **else if** and **else** parts are optional.

Remember these points when using conditional statements in Java:

- Conditions should be boolean expressions (**true** or **false**).
- Use curly braces {} to define the scope of the code blocks.
- **else if** and **else** parts are optional.

==========

Concept of && and & - AND Operator

Explanation:
how this works:

- The **if** statement starts evaluating from left to right.
- It first checks **c1**, which is **true**.
- Then it evaluates **c2**. Since **c2** is **false**, the overall condition of the **if** statement cannot be **true** anymore (because for an AND operation to be true, all operands must be true).
- At this point, the evaluation stops (short-circuits) without checking **c3** and **c4**.
- Since the overall condition evaluates to **false**, the code inside the **if** block (**System.out.println("selenium");**) will not be executed.

In summary, with the && operator, as soon as it encounters a false condition, it stops evaluating further because the final result will definitely be false.

==========

Explanation:
In this case:

- The **if** statement evaluates **c1**, which is **true**.
- It then evaluates **c2**, which is **false**. However, unlike with **&&**, it doesn't stop here.
- It continues to evaluate **c3** and **c4** as well.
- After evaluating all conditions, the final result is determined to be **false** (since one **false** operand in a series of **&** operations results in a **false** outcome).
- Consequently, the statement within the **if** block (**System.out.println("selenium");**) is not executed.

==========

Operand 1	Operand 2	Result with &	Result with &&	Notes
true	**true**	**true**	**true**	Both operands are **true**, so the result is **true** for both **&** and **&&**.
true	**false**	**false**	**false**	**&** evaluates both operands. **&&** stops after evaluating the first operand since the result cannot be **true** anymore.
false	**true**	**false**	**false**	**&** evaluates both operands. **&&** stops after evaluating the first operand.
false	**false**	**false**	**false**	Both operands are **false**, so the result is **false** for both **&** and **&&**. **&&** stops after evaluating the first operand.

Concept of || and | - OR Operator

Explanation:
In this case, **c1** is **true**. When the **if** statement is evaluated, it checks **c1** first. Since **c1** is **true**, the condition for the **if** statement is immediately satisfied, and

true.

- This behavior is known as "**short-circuit evaluation**."
- In an OR operation (||), if the left-hand operand is **true**, the right-hand operand is not evaluated because the overall expression is already true regardless of the value of the right-hand operand.
- This is an optimization technique to improve performance, especially in cases where evaluating all conditions might be resource-intensive or unnecessary.

==========

Explanation:
In this case:

- Even though **c1** is **true** and thus the overall condition of the **if** statement is **true**, the evaluation doesn't stop there.
- **c2**, **c3**, and **c4** will also be evaluated, which is different from using **||** where the evaluation would stop as soon as **c1** is found to be **true**.
- The output will still be "selenium" because the overall condition is **true**.
- **Useful for the logging purpose.**

Operand 1	Operand 2	Result with |	Result with ||	Notes
true	**true**	**true**	**true**	Both operands are true, so the result is true for both **|** and **||**.
true	**false**	**true**	**true**	**|** evaluates both operands. **||** stops after evaluating the first operand since the result is already true.
false	**true**	**true**	**true**	**|** evaluates both operands. **||** continues and evaluates the second operand since the first is false.
false	**false**	**false**	**false**	Both operands are false, so the result is false for both **|** and **||**. **||** evaluates both operands.

==========

Interview question:

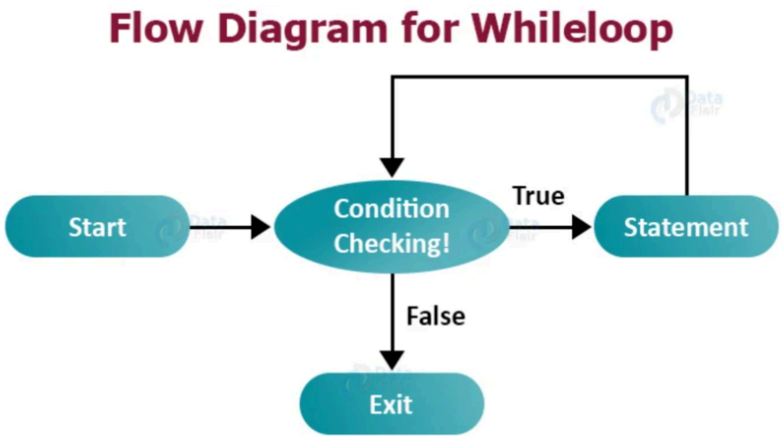
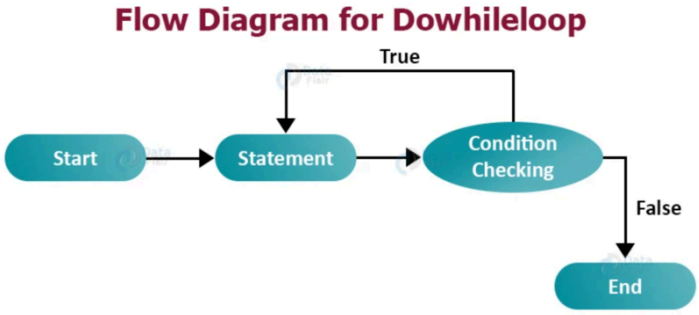
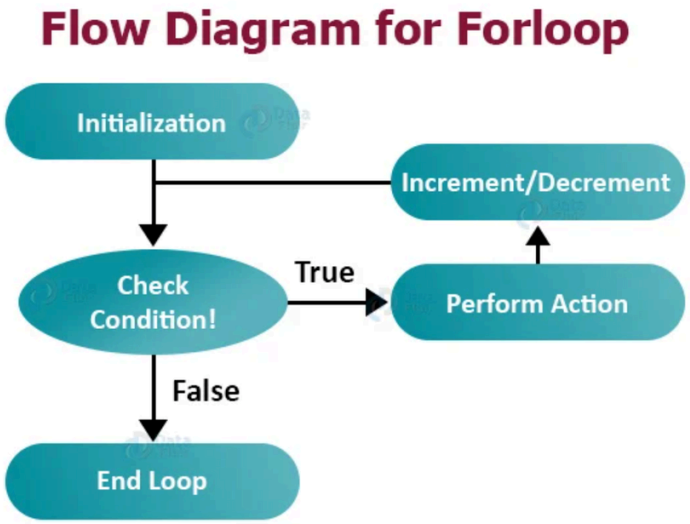
> Find the greatest number among the four variables.

LoopsConcepts_For_While_DoWhile

Loops	Notes

Why do we use loops?
Types of loops
Elements involved in loops
Syntax
	Loops:
	• Loops are used to execute a certain set of statements repeatedly until it meets the given condition.
	
	• for loop
	• while loop
	• do while loop
	
	Different components involved:
	• Initialization
	• Condition
	• Expression (Increment / decrement)
	• Body of the loop
	
	Syntax:
	for (initialization; condition; increment / decrement)
	{
	//statement to be executed
	}+
	
	eg:
	for(int i=10;i<1;i++)
	{
	System.out.println(i);
	}
	o/p : no o/p since the condition is validated and its false so nothing will be printed.
	--- 	
	while (condition)
	{
	//statement to be executed
	increment / decrement ;
	}
	--- 	
	do
	{
	//statement to be executed
	increment / decrement ;
	}**while** (condition);.0
	--- 	
	☐ **Important: While writing loop make sure exit condition must be present otherwise it will exhaust the memory and application**

Key difference between For loop versus While loop	Use **For** loop when number of iterations is fixed and use **While** loop when number of iterations is unknown. Example - For loop - Menu items on the page - calendar handling while loop - waiting for the element - waiting for the page to load

*Flow Diagram for **While loop***	
*Flow Diagram for **Do While loop***	
*Flow Diagram for **For loop***	

For Loop without initialization/condition	**for (; ;){**
	system.out.println("Hello"); }
	In such case, ***Hello* will be printed infinite times *because by default it will assume condition is true.***
	`
	for (; ;){
	system.out.println("Hello");
	break;
	}
	o/p:-Hello will be printed once
	Infinity is the property of loop.

Use Cases for while Loop	1. waiting for element on the page 2. waiting for the page to be loaded 3. pagination - Loop until the element is found (exp- Finding a person name in multiple pages of a web table) 4. For increment operators we can use i++, ++i or i=i+1 5. For Decrement operators we can use i--,--i or i=i-1
Use Cases for for Loop	1. drop-down traversing 2. menu items 3. calendar handling 4. We use for loop when number of iterations are fixed
Use Cases for do-while loop	1. Java do-while loop is used to iterate a part of the program several times. 2. do-while loop will check condition at the end of the block so it will be executed minimum 1 time.

Infinite loop-> while, for & do-while and use cases

while

Ex: 1

```
int i=1;
while(i<=10)
{
System.out.println(i);
}
```

o/p:111111111111.....

infinite times loop will be executed.

Ex:2

```
while(true)
{
System.out.println("Welcome to taj hotel");
}
```

Welcome to taj hotelWelcome to taj hotelWelcome to taj hotelWelcome to taj hotel.....

UseCase:

24/7 display hotel name

---

do while:

```
int p=1;
do {
System.out.println(p)
}
while(p<=10);
```

o/p- 111111...

---

for:

Example 1:

```
for(int i=10;i>1;i++)
{
System.out.println(i);
}
```

Example 2:

```
for(;;)
{
 System.out.println("Test");
}
```

By default the condition is taken as true,so enters into infinite loop

Example 3: Print even numbers from 1 to 100

```
String evenNumberList = "";
       for(int c=1;c<=100;c++) {
          if(c%2==0) {
        evenNumberList = (evenNumberList + c +",");
          }

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

o/p:-

2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86

```
// a to z with ASCII
for (char ch = 'a'; ch <= 'z'; ch++) {
```

	}
	
	// A to Z with ASCII
	for (**char** ch = 'A'; ch <= 'Z'; ch++) {
	System.***out***.println(ch + " = " + (**byte**) ch);
	}

Why we use for() loop?	1. For loops are used in java to execute statements repeatedly for a given number of times. 2. For loops are used when number of times to execute the statements is known to programmer.
What is the distinction between for\while & do while loop?	• In a do while the statement or body of the loop will be executed at least once before the condition is validated.
Symbol of %	◦ **%** is known as **modulus operator** . it gives us the remainder when we divide 2 numbers. • for example : 10%2 = Remainder is 0. 5%2 = Remainder is 1. • More examples: ▪ **10%1=0** ▪ **1%10 (any number)=1** ▪ **10/1=10** ▪ **1/10 (any number)=0**

Use cases of loops	//use cases of while loop:
	//when number of iterations are not fixed -- while loop
	//total links/images on the page
	//webtable pagination 1 2 3 4...next
	//webelement is loading on the page
	//page load time out
	//calendar:
	//build is running : 10, 1 hr , 2 hr, 30 mins
	//read data from DB: SQL -- rows/colns
	
	//use cases of for loop:
	//when number of iterations are fixed -- for loop
	//month/days drop down --> 1 to 12
	//category options -->
	//Arrays, Arraylist
	//excel file -- test data -- rows = 1 to rowSize()
	//country drop down --> 1 to 230 ---> if name = brazil -- break;
	//Read data : JSON/XML
	//read data from DB: SQL -- rows/colns
	
	//use cases of do-while :
	//1. webtable paginaton: check if element is already present in the table , click on it and end the loop
	//2. go and check the element first and then start the while loop
	//3. calendar:

Print A-Z , a-z, 0-9 with the respective ASCII numbers the console one using while, do-while and for loop.

for loop:

```
public class CharacterAndASCII {
    public static void main(String[] args) {
        // Print A-Z and their ASCII values
        for (char ch = 'A'; ch <= 'Z'; ch++) {
            System.out.println(ch + ": " + (int) ch);
        }

        // Print a-z and their ASCII values
        for (char ch = 'a'; ch <= 'z'; ch++) {
            System.out.println(ch + ": " + (int) ch);
        }

        // Print 0-9 and their ASCII values
        for (char ch = '0'; ch <= '9'; ch++) {
            System.out.println(ch + ": " + (int) ch);
        }
    }
}
```

while loop:

```
public class CharacterAndASCII {
    public static void main(String[] args) {
        // Print A-Z and their ASCII values using while loop
        char ch = 'A';
        while (ch <= 'Z') {
            System.out.println(ch + ": " + (int) ch);
            ch++;
        }

        // Print a-z and their ASCII values using while loop
        ch = 'a';
        while (ch <= 'z') {
            System.out.println(ch + ": " + (int) ch);
            ch++;
        }

        // Print 0-9 and their ASCII values using while loop
        ch = '0';
        while (ch <= '9') {
            System.out.println(ch + ": " + (int) ch);
            ch++;
        }
    }
}
```

Do-while loop:

```
public class CharacterAndASCII {
    public static void main(String[] args) {
        // Print A-Z and their ASCII values using do-while loop
        char ch = 'A';
        do {
            System.out.println(ch + ": " + (int) ch);
            ch++;
        } while (ch <= 'Z');

        // Print a-z and their ASCII values using do-while loop
        ch = 'a';
        do {
            System.out.println(ch + ": " + (int) ch);
            ch++;
        } while (ch <= 'z');

        // Print 0-9 and their ASCII values using do-while loop
        ch = '0';
        do {
            System.out.println(ch + ": " + (int) ch);
            ch++;
        } while (ch <= '9');
    }
}
```

Using for-each loop:

you'll need to convert the characters A-Z, a-z, and 0-9 into arrays before using the for-each loop.

```
public class CharacterAndASCII {
    public static void main(String[] args) {
        // Create arrays for A-Z, a-z, and 0-9 characters
        char[] upperCaseChars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
        char[] lowerCaseChars =
"abcdefghijklmnopqrstuvwxyz".toCharArray();
        char[] digitChars = "0123456789".toCharArray();

        // Print A-Z and their ASCII values using for-each
loop
        for (char ch : upperCaseChars) {
            System.out.println(ch + ": " + (int) ch);
        }

        // Print a-z and their ASCII values using for-each
loop
        for (char ch : lowerCaseChars) {
            System.out.println(ch + ": " + (int) ch);
        }

        // Print 0-9 and their ASCII values using for-each
loop
        for (char ch : digitChars) {
            System.out.println(ch + ": " + (int) ch);
        }
    }
}
```

Assignments on If-Else and Switch Case:

Conditional Operators:

Expected Output :
The greatest: 87

Find out the greatest number out of four different given numbers:
Input the 1st number: 25
Input the 2nd number: 78
Input the 3rd number: 87
Input the 4th number: 97

Expected Output :
The greatest: 97

2. Write a Java program to test a number is positive or negative.
Test Data
Input number: 35 -- positive number

3. WAP to check number is odd or even using If - Else
4. WAP to check given alphabet character is Vowel or Consonant using Switch - Case
5. WAP to run your test cases in a specific environment like: QA, Stage, Dev, UAT, Prod using using Switch - Case
6. WAP to book the specific type of car from the Uber app using Switch - Case.
    a. Car Type: Mini, Sedan, SUV, Premium
    b. If user passes wrong car type, print please select the right car type
7. WAP to launch browsers using If-ElseIf and Switch Case.
    a. Browser: Chrome/Firefox/IE/Safari
    b. If user passes wrong browser, print please pass the right browser name
8. WAP to define the interest rate on the basis of Loan type using Switch Case
    a. Loan Type: Car Loan, Housing Loan, Personal Loan, Education Loan
        i. For Housing Loan, if user's salary is less than 35000 USD - print : NOT APPLICABLE FOR Housing Loan

Ans:

Loops Assignments:

1. WAP to print following output:

I am Batman
I am Batman
I am Batman
I am Batman
I am Batman

2. WAP to print following output:
I am Batman 1
I am Batman 2
I am Batman 3
I am Batman 4
I am Batman 5
I am Batman 6
I am Batman 7
I am Batman 8
I am Batman 9

3. WAP to print 10 to 1 using for, while and do-while loop

4. Write a program in Java to print "Hello World" ten times using while loop

5. Write a program in Java to print all the multiplication of 5 from 1 to 100 using while /for/do-while loop

6. Print all odd and even numbers between 1 to 100

```
int i = 1;
while(i<=1){
System.out.println("Hi Java");
}
```

8. Print A-Z , a-z, 0-9 with the respective ASCII numbers the console one using while and for loop.

9. Print the following series:
1.0 2.0 3.0 ..... 10.0
0 9 18 27 36 ...99

10. Print only vowels (aeiou) using for and while loop. Start the loop from ‘a‘ to ‘z‘.

11. Print 1 to 10 and break the loop once you find the multiplication of 7 with a message "bye, see you tomorrow".

Time Complexity:

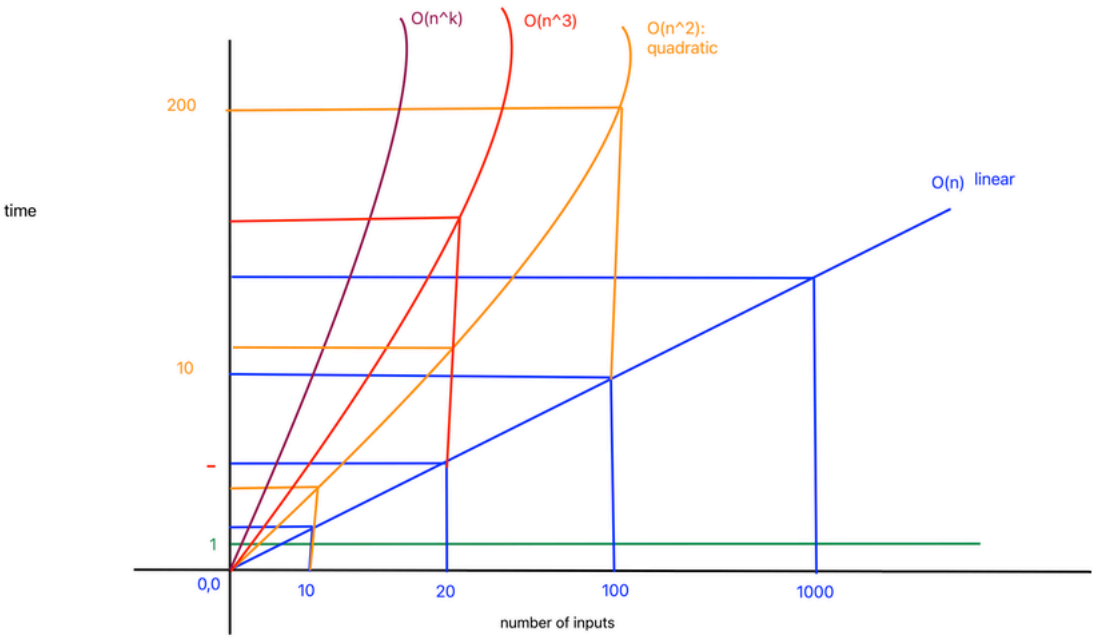
Time Complexity is a way to measure how an algorithm's performance (specifically its running time) grows as the size of its input grows. It helps us understand how "efficient" an algorithm is, regardless of the specific hardware it runs on.
Think of it like this: if you have a room of 10 people and need to find someone wearing a red shirt, you might need to look at each person once. If the room had 100 people, you'd need to look at 10 times as many people. This is an example of linear time complexity - as the input (number of people) grows, the time needed grows proportionally.

Common Time Complexities (from fastest to slowest):

1. O(1) - Constant Time This means the algorithm takes the same amount of time regardless of input size. Example: accessing an array element by its index.
2. O(log n) - Logarithmic Time The algorithm's time increases logarithmically with input size. Example: binary search in a sorted array.
3. O(n) - Linear Time Time grows linearly with input size. Example: finding the maximum value in an unsorted array.
4. O(n log n) - Linearithmic Time Common in efficient sorting algorithms like mergesort and quicksort.
5. $O(n^2)$ - Quadratic Time Time grows with the square of input size. Example: nested loops comparing each element with every other element.
6. $O(2^n)$ - Exponential Time Time doubles with each additional input element. Example: recursive calculation of Fibonacci numbers without memoization.

When analyzing time complexity, we focus on the worst-case scenario and ignore constants and less significant terms. For example, if an algorithm takes $3n^2 + 2n + 1$ operations, we simplify it to $O(n^2)$.
Understanding time complexity helps developers choose the right algorithm for their specific needs, balancing between performance requirements and implementation complexity.

Example	Description	Time Complexity
1	Printing the value of **i**	O(1)
2	Printing numbers from 1 to 10 using a **for** loop	O(n)
3	Printing numbers from 1 to 10 using a **while** loop	O(n)
4	Printing numbers from 1 to 10 and breaking at 5	O(n)
5	Printing pairs of numbers from 1 to 5 using nested loops	O(n^2)
6	Printing triplets of numbers from 1 to 5 using nested loops	O(n^3)
7	Printing numbers from 1 to 10 within a nested loop structure	O(n^2)



Static Array (Fixed length array/Static Array):

Array declaration

```
// Integer array:

 1. Array Literals:
int p[] = { 10, 2, 3, 5 }; // using this syntax we can assign values to an array

 2. using new keyword with default values:
int a[] = new int[4];
            a[0] = 1;
            a[1] = 2;
            a[2] = 3;
            a[3] = 4;
// **** iterate array through typical for loop
            for (int i = 0; i < p.length; i++) {
                System.out.println("The value at index:  i  + "=" + p[i] );
            }
Using typical for loop, we are iterating using index and hence we need to use integer while iterating.
/**** iterate an array using enhanced for loop. Here while using enhanced
for loop we use similar data type variable with respect to the array we are iterating */
int count = 0;
int p[] = { 10, 2, 3, 5 }; // using this syntax we can assign values to an array
for( int e : p) {
      System.out.println("The value at index : " + count + " = " + e );
      count++;
}
/************ Float array **********************
While assigning values to the float array, there is no need to specify decimal values to the array elements. By default all floating literals are considered as Double literals. If we mention decimal values in float array then we have to append each value with f or F otherwise we get error - Type mismatch: cannot convert from double to float
float q[] = new float[] { 1, 2, 3, 4 };
float f[] = new float[] { 1.0f, 2.0f, 5.0f, 6.0f };
float k[] = new float[4];
            k[0] = 1.0f;
            k[1] = 2;
            k[2] = 3.0f;
            k[3] = 4.0f;
// The concept of object array comes into picture wherein if we want to save heterogeneous data.
            Object obj[]=new Object[4];
            Object obj1[]=new Object[] {"John","Male",5.7,57.2};
            for(Object e:obj1) {
                System.out.println(e);
            }

//object static array:
Object emp[] = new Object[5];//0-4
emp[0] = "Tom";
emp[1] = 25;
emp[2] = 12.33;
emp[3] = 'm';
emp[4] = true;

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

System.out.println("-----");

for(Object e : emp) {
System.out.println(e);//Tom
```

	System.***out***.println("hi...."); **break**; } } ========== In Java, the term "array literals" refers to a shorthand notation for creating and initializing an array. An array literal allows you to define the elements of an array directly within the code without explicitly specifying the size of the array or using a loop to populate it. Here's an example of an array literal for an array of integers: int[] myArray = {1, 2, 3, 4, 5}; In this example, the array **myArray** is created and initialized with the values 1, 2, 3, 4, and 5. The size of the array is inferred from the number of elements provided in the curly braces. You can also use array literals for other data types: String[] myStringArray = {"apple", "banana", "cherry"}; **double** d[] = {12.33, 44.55, 8.99}; **char** c[] = {'a', 'b', 'r'}; String emp[] = {"Shhubham", "Pooja", "Naresh", "Adil"}; Object studentInfo[] = {"Vijay", 25, 34.44, 'm', "Pune", "India", **false**}; System.***out***.println(Arrays.*toString*(d)); System.***out***.println(Arrays.*toString*(c)); System.***out***.println(Arrays.*toString*(emp)); System.***out***.println(Arrays.*toString*(studentInfo));
What is array? ***Types of array..***	If we want to store similar type of data, then we should not create different variables. We should go with Arrays. We can combine all data together, using a single variable name and allocate common memory. Example: store all student names, store all product names. **<u>There are 2 types of array:</u>** 1- static array 2- dynamic array

About static array..

The size of the array will be fixed in this case.

How to declare an 'int' type array:

int p[] = new int[4];

* Here, length of the array is 4.
* How much memory is allocated? -> int occupies 4 bytes each. so, 4 * 4=16 bytes
* There is Li(Lowest Index) and Hi (Highest Index), where Li is always 0 and Hi is Length-1.
- LI = 0
- Length = p.length
- HI = Length-1
- Length = HI + 1

How to declare double type array:

double d[]= new double[2];

* Since each double value occupies 8 bytes of memory, here the above declared array will occupy 2*8=16 bytes

Problems with static array:
1-either the memory is over-allocated. i.e., initially declared with size 499,
and if currently only 100 size is required to store products, it is wastage of 399 memory slots.
2-or there will be scarcity of memory. i.e, if initially declared with size 499,
and if currently 600 size is required to store similar data,
then we would have to shut our app and then increase the array size and then start the server.
If this was taken care in 30 mins, then there will be huge business loss in 30 mins.

Where to use static array:
*where count of data is static. example: number of days in a calendar, number of months, in an application
where menu items will always be same etc.

Note: If we try to store a value in index greater than the size of the array, then we see **arrayIndexOutOfBoundException**.
Also, if we try to assign a value to index '-1', then also we see **arrayIndexOutOfBoundException**.

Example:
int a[]=new int[4]; //array declared with size 4
a[4]=98; //this will throw arrayIndexOutOfBoundException as the 'Hi' is 3.
a[-1]=98; //this will throw arrayIndexOutOfBoundException as the 'Li' is 0.

- **-ve indexing is not allowed in Java but it is allowed in Python.**

How to print all the values of an array -> by using 'for' loop or 'for each' loop

The array we can define with below types:
-> int
-> char
-> String
-> double
-> Object // If we are storing different types of data in array, then use Object type of array. Example below:

Object employee[]=new employee[5];
employee[0]="Lisa"; // denoting name-string type
employee[1]='F'; //denoting gender-char type
employee[2]=25; //denoting age-int type
employee[3]=24.55; //denoting salary-double type
employee[4]=true; //denoting isActive - boolean type

Important notes about static arrays in Java:

Note	Description
Declaration	An array must be declared with a specified size and type. Example: **int[] myArray = new int[10];**
Initialization	Arrays can be initialized at the time of declaration. Example: **int[] myArray = {1, 2, 3, 4, 5};**
Fixed Size	Once created, the size of a static array cannot change. Attempting to access an index out of the array's bounds will result in an **ArrayIndexOutOfBoundsException**.
Default Values	Each element in an array is automatically initialized with a default value (0 for numeric types, **false** for **boolean**, **null** for objects).
Indexed Access	Array elements are accessed via their index, with the first index being 0. Example: **myArray[0]** refers to the first element.
Iteration	Arrays can be iterated using a **for** loop or an enhanced **for-each** loop.
Type-Specific	An array can hold primitives or objects, but all elements must be of the declared array type.
Memory Allocation	Memory for an array is allocated on the heap, and the array holds references to the actual objects.
Length Property	The length of an array can be accessed with the **length** property. Example: **myArray.length**.

Example when to use new int[] and Array Literals	**Scenario**	Example

Example where you might use array initialization with **new int[4]**.	Suppose you are building a program to store and process the temperatures recorded each day of the week. You want to store the temperatures for Monday to Friday.
	Using new int[4]:
	
	In this approach, you might use **new int[5]** to allocate memory for an array of five integers to store the temperatures.
	Since you don't know the temperatures at the time of declaration, you initialize the array with default values (0 for temperatures).
		int[] temperatures = new int[5]; //initial all are zero (default temp = 0)
		
		//Assigning the temp later:
		temperatures[0] = 25; // Monday
		temperatures[1] = 26; // Tuesday
		temperatures[2] = 24; // Wednesday
		temperatures[3] = 23; // Thursday
		temperatures[4] = 27; // Friday

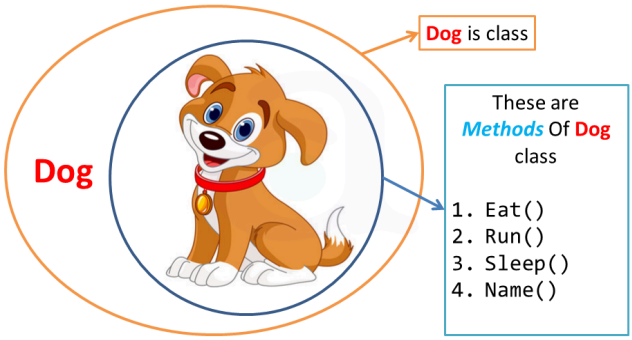
Example where you might use array initialization **Array Literals.**	**Using array literals:** In this approach, you know the temperatures at the time of declaration, so you can use array literals for initialization.	int[] temperatures = {25, 26, 24, 23, 27};

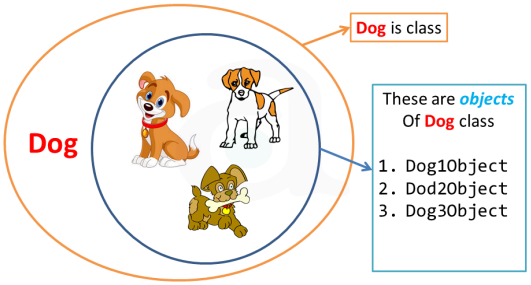
Java Class_And_Objects_References_NullReference:

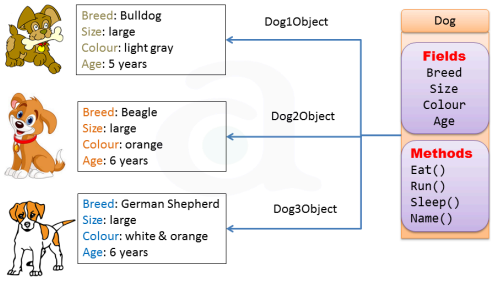
Class	*Object*

• It is a ***blueprint*** or ***template*** or ***category***. • It is used to declare or create object. • You can create multiple objects by using a single class.	• Object is an ***instance of class.*** • Multiple references are allowed for a single object.
• ***No memory is allocated*** when a class is declared.	• Memory is created as soon as an object is created (Run time). • The **new** keyword is used to allocate memory ***at runtime.*** • All objects get memory in ***Heap memory*** area.
• Class is a ***logical entity*** (blue print).	• Object is a ***physical entity.***
• Class can only be declared once.	
• Multiple objects can be created as per requirement.	
• For example Car is a class	• Object of class car can be BMW, Jaguar, Audi
• Class c=new Class();	• c is stored in stack, where as the object resides in heap. The memory related to heap is taken care by Garbage collector of JVM.

Class and Object Illustrations







What is Object in Java?

- An object is an instance of the class which has state and behavior.
  - **State:** represents the data (value/variables) of an object. (what it has?)
  - **Behavior:** represents the behavior (functionality) of an object. (what it can do?)
-

How to create an Object in Java?

- There are 3 ways to initialize object in Java. Initializing an object means storing data into the object.
  i. By reference variable
  ii. By method
  iii. By constructor

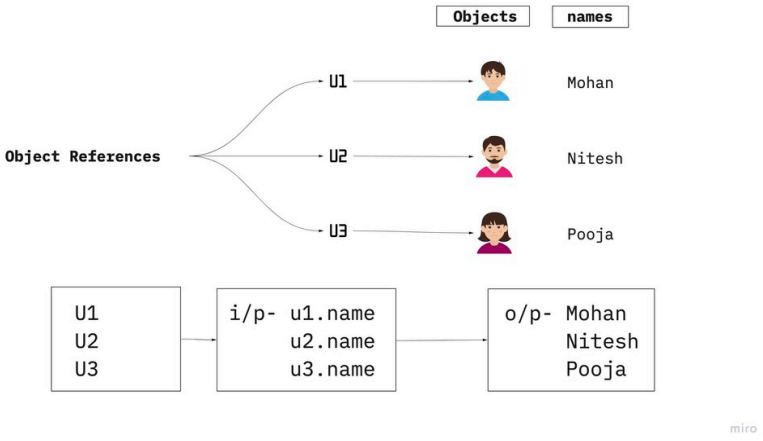
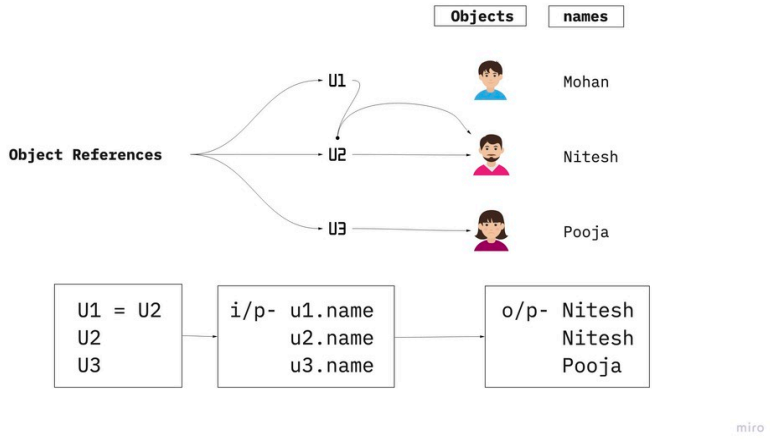
No Reference Object

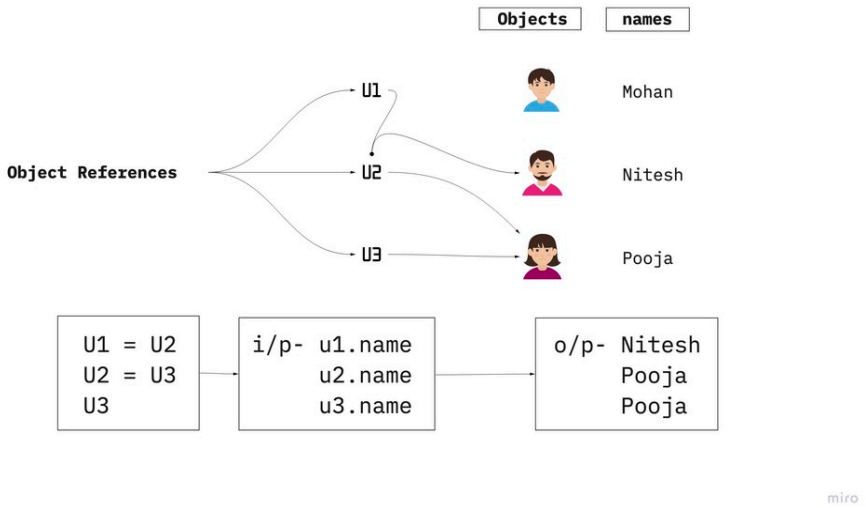
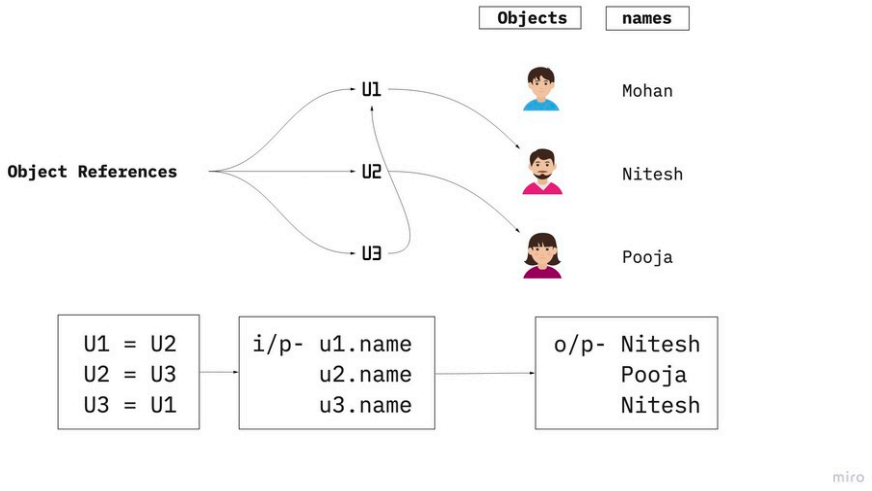
- **Employee e1= new Employee();**
  - **Employee :** Class name
  - **e1 :** Object_Reference_Name/ reference variable
  - **new Employee():** is the real object(RHS)
  - Here **e1** is not an object the real object **new Employee()** is being referred by the reference variable e1.

- **new Employee();**
  - We can create an object without a reference variable it's called as ***No Reference Object***.
  - But it's not a good practice to create unnecessary many objects without a reference.

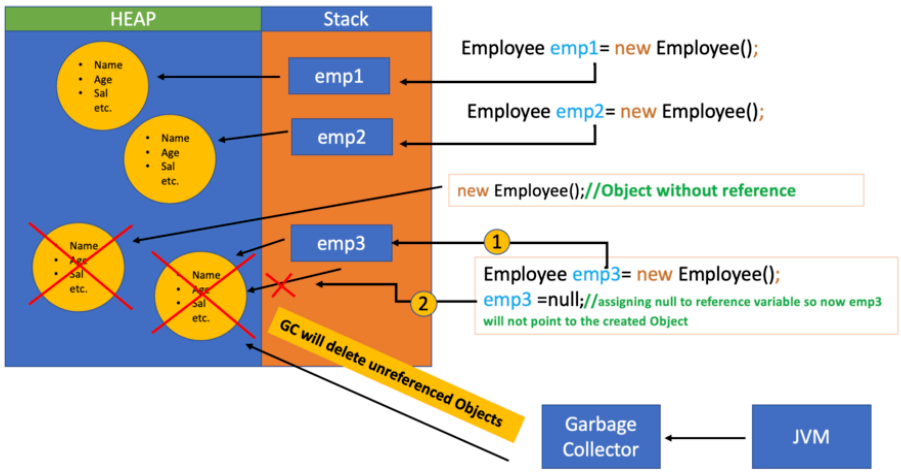
Null Reference Object	○ How many times this statement is executed that many number of objects are created and corresponding memory is allocated in heap.
	
	--- 	
	• Here ObjectReference e3 pointing to **new Employee()** object
	`Employee e3= new Employee();`
	`e3.name="Peter";`
	`//System.out.println(e3.name); ------o/p= 'Peter'`
	`e3=null;`
	• Now ObjectReference e3 pointing to **null**
	`System.out.println(e3.name); ---> (null.name)`
	☐ **o/p:** NullPointerException: Cannot read field "name" because "e3" is null
	
	• Now **new** Employee() is being **NullReferenceObject** ready for garbage collection.

Object References:

Object References U1,U2,U3 pointing to their respective objects.	
ObjectReference U1 pointing to U2.	

Before U2 pointing towards U3 ,U2 was pointing towards itself that's why U1 gives Nitesh as o/p	
Only think of current situations . *Where the given reference is pointing at this moment of time???*	

NOTE	☐ One object can have ***multiple references.***
Default Values for Different Data Types	• Default value of **String** is **null** • Default value of **Integer** is **0** • Default value of **double** is **0.0** • Default value of **char** is **blank space** • Default value of **boolean** is **false**
Garbage collector eligible for:	• NonReferenceObject. • NullReferenceObject i.e reference pointing to null. • Java GC operates only in the heap memory section not in the stack memory section.
System.gc()	• It might call the jvm or not.Garbage collector will run to reclaim the unused memory space upon instructions from JVM.

Memory management Java	 • Object is always created in Heap memory. • Reference variable is created in Stack memory.
Role of Garbage Collector and JVM in memory management	◦ GC will destroy those objects references which are referring to the Null from heap memory. • For ex: **emp3**=Null; ▪ It will break the connection and If we try to print then we will get Null pointer exception • ◦ We can also request GC to collect non-referenced objects by using ***System.gc();*** ◦ After GC receives the users request, but waits for instructions from JVM for cleaning the heap memory. • ☐ **NOTE**: ***Garbage collector is defined only for heap memory GC can't access stack memory.***

NullPointerException	• ObjectReference e3 pointing to **new Employee()** object
	`Employee e3= new Employee();`
	`e3.name="peter";`
	`e3.age=24;`
	`//System.out.println(e3.name); ------o/p= Peter`
	• Now ObjectReference e3 pointing to **null**
	`e3=null;`
	`System.out.println(e3.name);`
	`//System.out.println(e3.name); ------o/p (null.name)`
	☐ NullPointerException: Cannot read field "name" because "e3" is null
	• Now **new** Employee() is being **NullReferenceObject** ready for garbage collection.
	• Here In NullReferenceObject reference is still there but pointing to null.

Functions/Methods_InJava_with_DifferentExamples

What are the functions/methods in java?	• A **method/function** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. • It is used to achieve the **re-usability** of code. We write a method once and use it many times. We do not require to write code again and again. • It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. ☐ ***NOTE: The method is executed only when we call or invoke it.***

Features of Methods/functions	• A function is also called as ***method***.
	• A function ***cannot be created inside a function.*** - no nested functions.
	• Functions are ***parallel*** to each other.
	• Functions are always ***independent of each other***.
	
	• To call a function we have to create the object of the class inside main() method. if it is non static.
	• We cannot create function inside main method.
	
	☐ ***We cannot use return and break together inside any method.***
	
	• If using **return**, it should be the last statement of your function.
	• **Return** keyword is not used in the main method.
	
	☐ ***void and return keyword should not be used together.***
	
	• Function name can start with small letter if its single word else should start with small letter and have capital letter for second word.
	• For Function naming use camel casing--
	  ◦ example: Public void launchBrowser()
	  ◦ Eg: public void sum();
	  public void getSum();
	
	☐ ***Only Non-static methods and variables are called as a data-members of the class.***

What is method signature?	• Every method has a method signature. It is a part of the method declaration. • It includes the **method name** and **parameter list**. • The return type and exceptions are not considered as part of it. Above ex. : ***max(int x, int y).***

How to call a non-static (Instance) method inside the main() method?	• The method of the class is known as a non-static/instance method. It is a non-static method defined in the class
	• without the static keyword. Before calling or invoking the instance method, it is necessary to create an object
	• of its class.
	
	• ***Create an object of a class Math*** (where the above max() method is defined) inside the main() method.
	  ◦ Here the max() method should be non-static.
	  Math m1= new Math();
	
	• ***Call the method using ObjectRefrence variable inside main () method***
	  m1.max(2,4);
	  //here 2,4 are called arguments

How to call a static method inside the main() method?	• A method that has **static** keyword is known as static method. In other words, a method that belongs to a class rather • than an instance of a class is known as a static method. • We can also create a static method by using the keyword static before the method name. • The main advantage of a static method is that we can call it without creating an object. • ***Inside the same class*** where max() method is defined as static Call the method directly as below: ***MethodName(arguments);*** ***max(2,4);*** *(in case max() method is static)* ☐ ***Outside the class*** Call the method as : ***ClassName.MethodName(arg);*** ***Math.max(2,4);*** • The best example of a static method is the main() method.
void	• void means it cannot return any value.

Default Function/Zero parameterised/ No input no return

```
public void getMarks()
    {
        int a=10;
        int b=20;
        int c=30;
        int total=a+b+c;
        System.out.println("total marks"+total);
        }
```

where

- getMarks()- Function name/ method name
- void- Return type i.e cannot return any value

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

        Math m1= new Math();
        int t=m1.getMarks();
        System.out.println(t+20);
    }
```

- A method which is preceding by void keyword wont return any value.

Types of Functions:-

- ***No input no return***
- ***No input and some return***
- ***Some input and some return***
- ***Some input and no return***

No input and some return

```
public int getMarks()
    {
        int a=10;
        int b=20;
        int c=30;
        int total=a+b+c;
        System.out.println("total marks"+total);
        return total;
        }
```

NOTE:

- ☐ ***return statement should be the last statement of any function with return type.***

Some input and some return

```
public int add(int a, int b ) {.   // int a and int b are parameters

        System.out.println("add-method");
        int sum=a+b;
        return sum;

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

        Math m1= new Math();
        int s1=m1.add(23,35);   // 23 and 35 are arguments
        System.out.println(s1);
    }
```

Difference between Parameters and Arguments	☐ **Parameters-** At the time of creating function what we give. ☐ **Arguments-** Actual values that we are passing while calling required method in main method.

Return Keyword:

return Keyword:	• ***we cannot write 2 or more return keywords together in a function, return should be the last statement of the function.*** • Return can be any of these datatypes (string, int, boolean, arraylist,array,double,float..) • if we add a return statement to a function , but use void while writing a function , you will see a error in the code • return statement datatype should match the declaration of the type on the function • eg - public void sum(){ ○ int a = 7 ; return a ; } - will throw a error while coding as we have declared return type for function as void, but returning a int. • holding variable is good practice to store values returned by the function in a class object and manipulate it further • A function can not be created inside the main method. • Call a function have to create the object of the class. **NOTE**: ☐ ***return statement should be the last statement of any function with return type.*** ☐ ***There should be only one return statement per function.*** ☐ ***Key point: Any statement after return statement will result in compile-time error stating "Unreachable code".***
Can we change return type of main() method in java?	• The *public static void main()* method is the entry point of the Java program. • Whenever you execute a program in Java, the JVM searches for the main method and starts executing from it. • You can write the main method in your program with return type other than *void*, the program gets compiled without compilation errors. • But, at the time of execution JVM does not consider this new method (with return type other than *void*) as the entry point of the program. • It searches for the main method which is public, static, with return type void, and a String array as an argument. ○ `public static int main(String[] args) { int a=34; return a; }` • If such a method is not found, a run time error is generated. ☐ Error: Main method must return a value of type void in class demo.Demoyt, please define the main method as: public static void main(String[] args)
Why main() method in Java is Void and Static?	• JVM call main() method to execute the program.Once main() method has finished executing, java program also terminates. • If we provide return statement in main(), JVM cant do anything with that return value. So main() method is always void in nature • main() method is static because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

Methods Assignments:

1.Write a program to print the **addition/subtraction/division/multiplication** of two numbers entered by user by defining your own method

2. Define a method that returns the product of two double numbers entered by user.

3.Write a program to print the circumference and area of a circle of radius entered by user by defining your own method.

---

4. Define two methods to print the maximum and the minimum number respectively among three numbers entered by user.

---

5. Define a program to find out whether a given number is even or odd - return true/false.

---

6. A person is eligible to vote if his/her age is greater than or equal to 18.
Define a method to find out if he/she is eligible to vote. - return true/false

---

7. Write a program which will ask the user to enter his/her marks (out of 100). Define a method that will display grades according to the marks entered as below:

Marks	Grade
91-100	AA
81-90	AB
71-80	BB
61-70	BC
51-60	CD
41-50	DD
<=40	Fail

---

8. Write a program to print the factorial of a number by defining a method named 'Factorial'.
Factorial of any number n is represented by n! and is equal to 1*2*3*....*(n-1)*n. E.g.-
4! = 1*2*3*4 = 24
3! = 3*2*1 = 6
2! = 2*1 = 2
Also,
1! = 1
0! = 1

Method Overloading:

- **Definition of Method Overloading**:
  - Method overloading occurs when you have multiple methods within the same class with the same name but different parameters.
  - The parameters can differ in number, type, or sequence.
- **Rules for Method Overloading**:

1. Methods must have the same name.
2. Methods can have a different number of parameters.
3. Methods can have parameters of different types.
4. Methods can have parameters in a different sequence.
5. The return type of the method does not matter for overloading.

Polymorphism	Notes

Static or Compile-Time Polymorphism:	Polymorphism and why method overloading is referred to as static or compile-time polymorphism: **Static or Compile-Time Polymorphism**: ○ Method overloading is an example of static or compile-time polymorphism in Java. ○ In method overloading, the compiler determines which overloaded method to call based on the method signature at compile time. ○ The decision of which method to call is made during the compilation phase, hence the term "compile-time polymorphism." ○ This is in contrast to dynamic or runtime polymorphism, which is achieved through method overriding in inheritance hierarchies. ○ Static polymorphism offers better performance compared to dynamic polymorphism because the method binding occurs at compile time, eliminating the need for runtime method resolution. ○ Method overloading allows for cleaner and more concise code by providing multiple methods with the same name for different use cases, improving code readability and maintainability.

Examples:

Can we overload main() method in Java?

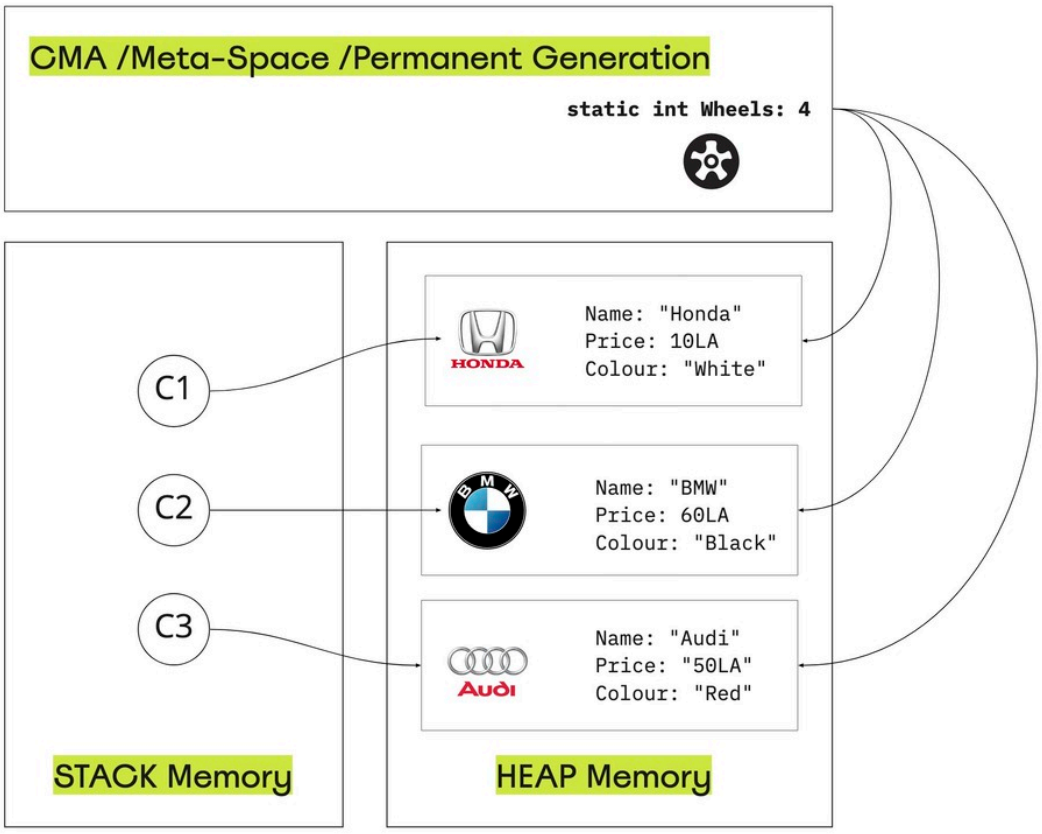
The question is that **"can we overload main() method in Java?"**

- Yes, We can **overload the main() method in Java**.
- JVM calls any method by its signature or in other words JVM looks signature and then call the method.
- If we **overload a main() method** in a program then there will be multiple **main() methods** in a program. So JVM calls which method? we don't need to confuse if we have multiple **main() methods** then JVM calls only one **main() method** with (string[] argument) by default.

Example: Overloading main() method

Java Memory : Heap & Stack

Java Memory	is divided into following Sections. 1. Heap 2. Stack 3. CMA/Meta-Space/ Permanent Generation • ***The Stack section of memory contains regular methods, local variables, and reference variables.*** • ***The Heap section of memory contains Objects.***



- ***PermGen Memory:*** This is a special space in java heap which is separated from the main memory where ***all the static content*** is stored in this section. Apart from that, this memory also stores the application metadata required by the JVM.
- The biggest disadvantage of PermGen is that it contains a limited size which leads to an Out of Memory Error.
- Also the PermGen space cannot be made to auto increase. So, it is difficult to tune it. And also, the garbage collector is not efficient enough to clean the memory.
- Due to the above problems, **PermGen has been completely removed in Java 8.** In the place of PermGen, a new feature called **Meta Space(Common Memory Allocation)**has been introduced. MetaSpace grows automatically by default. Here, the garbage collection is automatically triggered when the class metadata usage reaches its maximum metaspace size.
- If CMA is fully *dynamic memory allocation* happens i,e if beyond 200mb, space will be shared from systems RAM.

final keyword for a variable.	• If we initialize a variable with the final keyword, then we cannot modify its value.
	  ◦ **ex:- static final int wheels = 4;**
	  ◦ Now wheels is constant ,you can't change the value of wheels to any other value.
	  ◦ Naming convention for final variables - Use capital letters. e.g. PAGE_REFRESH_TIMEOUT, URL, COUNTER, etc.

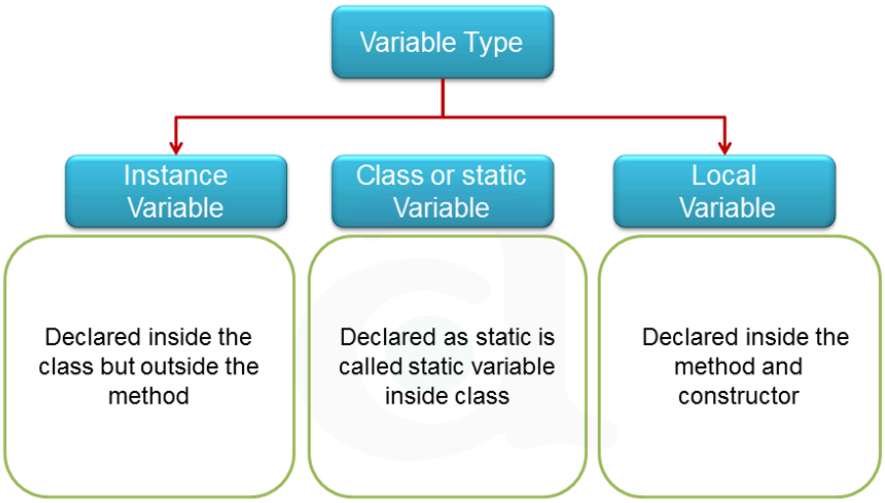
How to call static variables and methods	☐ ***Inside the same class*** where max() method is defined as static Call the method directly as below: ***MethodName(arguments);*** ***max(2,4);*** *(in case max() method is static)* ☐ ***Outside the class*** Call the method as : ***ClassName.MethodName(arg);*** ***Math.max(2,4);***

Can we declare local variables as static?	• In Java, a static variable is a ***class variable (for whole class).***
	• So if we have static local variable (a variable with scope limited to function), it violates the purpose of static. ***Hence compiler does not allow static local variable.***
	
	`class Test {`
	`  public static void main(String args[]) {`
	`   System.out.println(fun());`
	`  }`
	
	`  static int fun()`
	`  {`
	`   static int x= 10;  //Error: Static local variables are not allowed`
	
	`   return x--;`
	`  }`
	`}`

Can we declare local variables as final ?	***YES***
	`public static void main(String[] args) {`
	
	`System.out.println(fun());`
	`}`
	
	`static int fun()`
	`{`
	`final int x= 10;`
	
	`//allowed But you can't change the value of x once you declare it`
	`return x;`
	`}`

Types of variables:

Types of Variables



- Instance variables : One copy per object. Every object has its own instance variable.
  - E.g. x, y, r (centre and radius in the circle)
- Static variables : One copy per class.
  - E.g. numCircles (total number of circle objects created)

Local vs Instance vs Static/Class variables

	Local variable	Instance variable	Static/Class variable
Declaration	Inside methods/ constructors/ blocks	Inside the class but outside the method/constructor/block	Declared as static inside the class but outside the method/ block/ constructor
Scope	Only inside the methods/ constructors/blocks	Inside all methods/ blocks/ constructors within a class.(not inside a static method directly ,need to create object.)	Everywhere inside the class including static methods
When variable get allocated in memory?	When method/constructor/ block get executed it allocates memory and get destroyed after exiting from block/method.	Instance variables are created when an object/instance of the class is created and destroyed when the object is destroyed.	Static variables are created at the start of program execution , when *.class* file executed and destroyed automatically when execution ends.
Stored in ..?	Stack memory	Heap memory	Non-Heap /Static memory
Default value	Don't have default values. Initialization of the local variable is mandatory before using it in the defined scope.	Int 0 Boolean false String null	Int 0 Boolean false String null
Access specifier /Modifiers	We can't use access specifiers with local variables.	Can be used.	Can be used
How to access?	Directly inside the block/ method/ constructor.	For static method call it directly. For simple method create object to access it.	Inside the class directly. Outside the class by using ClassName.b By using object reference name.

Stack Over Flow Error:

We have a class **Demo** with three methods **t1()**, **t2()**, and **t3()**. Each method calls the next one in a circular manner. This leads to an infinite loop of method calls,

1. The **main** method creates an instance of **Demo**.
2. **d.t1()** is called from **main**, which prints "Method t1" and then calls **t2()**.
3. **t2()** prints "Method t2" and calls **t3()**.
4. **t3()** prints "Method t3" and calls **t1()**.
5. **t1()** prints "Method t1" and calls **t2()**.
6. The cycle repeats indefinitely: **t2()** calls **t3()**, **t3()** calls **t1()**, and so on.

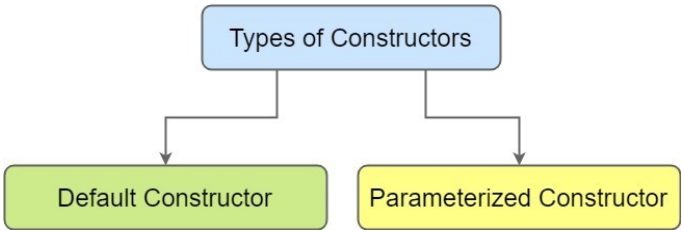
Since there's no base case or termination condition in the method calls, the recursion continues infinitely. Eventually, the call stack overflows, and a **StackOverflowError** is thrown by the JVM.

To fix this issue, you need to ensure that the recursive method calls have proper termination conditions to prevent infinite recursion. In this example, adding a termination condition to any of the methods would break the infinite loop and prevent the stack overflow error.

Topic: Constructors_ThisKeyword

What is constructor?	• A constructor is a block of codes which is used to initialize the object. • 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. • 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. • It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.
Rules for creating Java constructor	• Constructor name must be the same as its class name. • A Constructor must have no return type. • We can overload the constructor. They are differentiated by the compiler by the number of parameters in the list and their types. ***NOTE*** ☐ We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java. ☐ ***Rule***: *If there is no constructor in a class, compiler automatically creates a default constructor.*
Constructor vs Method	**Java Constructor** / **Java Method** A constructor is used to initialize the state of an object. / A method is used to expose the behavior of an object. A constructor must not have a return type. / A method must have a return type. The constructor is invoked implicitly. / The method is invoked explicitly. The Java compiler provides a default constructor if you don't have any constructor in a class. / The method is not provided by the compiler in any case. The constructor name must be same as the class name. / The method name may or may not be same as the class name. ☐ The constructor will be called the moment you create an object of the class. ☐ The method will be called when you create the object of the class and use objectReference to call the method. ☐ We never write a business logic inside the constructor ,it's only used to initialise the object that's it.

Constructor Types



1. ***Default Constructor:***
   - A constructor is called "Default Constructor" when it doesn't have any parameter.
   - If there is no constructor in a class, compiler automatically creates a default constructor.

1. ***Parameterised Constructor:***
   - A constructor which has a specific number of parameters is called a parameterized constructor.
   - The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Constructor program

```
public class Employee {

    String name;
    int age;
    String city;            //Class variables
    double salary;
    boolean isPerm;

    public Employee(String name,int age) {  //Local variables

        System.out.println("Employee constructor");
     // this keyword used to access current class variables
        this.name=name;
        this.age=age;
    }
    public static void main(String[] args) {

 // you have to pass only name and age here ,here its compulsory while creating object
        Employee e1= new Employee("Amol",24);
        System.out.println(e1.name);
        System.out.println(e1.age);
    }
}
```

NOTE:

- [ ] If you don't assign values to the class variables using 'this' keyword it simply will print default values of class variables e.age=0; e1.name=null.

Constructor Chaining this() Method

This() is used to achieve constructor chaining. In other words, It is the way in which constructors call each other.

To add multiple Constructors use to short cut - Right Click → Source → Generate Constructors using Fields option.

this() Example

```
public class ChainWithinClass
{
   ChainWithinClass(){
       System.out.println("\nThis is the no-arg constructor.");
   }

   ChainWithinClass(int y){
       this();
       int var1 = y;
       System.out.println("You passed one argument: " + var1);
   }

   ChainWithinClass(int a, int b){
       this(3);
       int var2 = a;
       int var3 = b;
       System.out.println("You passed two arguments: " + var2 + " and " + var3);
   }

   public static void main(String[] args){
       ChainWithinClass chainObj = new ChainWithinClass(2,4);
   }
}
```

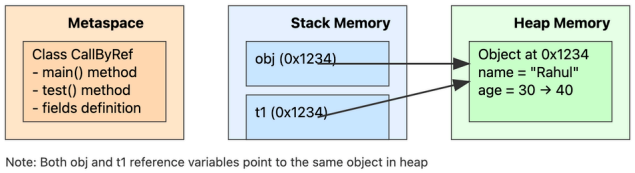
Running this code will produce the following output:

```
This is the no-arg constructor.
You passed one argument: 3
You passed two arguments: 2 and 4
```

this() Vs this.	this. keyword is used to refer to the current object whereas this() for calls among constructors.
Private Constructor	Used when want to restrict creation of objects of that class. e.g. System (Inbuilt class) Though all of methods in system class are static.

Advantage of Constructors • Constructor needs to follow the rules of overloading • Duplicate constructors are not allowed • No business logic in constructors • Constructor should have the same name as that of the class	• Constructor restricts creating the unnecessary objects in heap memory . • Constructor is used to create physical entity in the form of object inside the memory which will help me to initialize the class variables. • different constructors can be called based on the arguments passed while creating instances of the class • Constructor is called when the object is created. • Constructor never returns any value.

Call By Reference:



Note: Both obj and t1 reference variables point to the same object in heap

Call By Reference Notes:

1. Object Reference Passing

- When we pass an object to a method, Java passes the reference of that object
- The parameter t1 in the test method receives a copy of the reference, not a copy of the object
- Both obj and t1 point to the same object in memory

2. Memory Behavior

- In this code, when obj is created with name="Rahul" and age=30
- When test(obj) is called, t1 points to the same object
- Any changes made to object properties through t1 affect the original object
- This is demonstrated when t1.age = 40 changes the age value that can be seen through obj
- Each new object gets its own memory space
- Default values are used (null for String, 0 for int) when no values are assigned
- The same test method can work with different object references

Key Points

- Java is always pass-by-value, but for objects, the value being passed is a reference
- Changes to object properties persist outside the method
- Changes to the reference itself (pointing t1 to a new object) would not affect the original reference
- Instance variables (name, age) are shared between all references to the same object

Common Use Cases

- When you need to modify object state within methods
- Working with complex data structures
- Implementing builder patterns or chainable methods
- Reducing memory overhead by avoiding object copying
-

This example demonstrates how Java handles object references and how modifications to object properties through method parameters affect the original object.

```
package javasessions;

public class CallByRef {

String name;
int age;

public static void test(CallByRef t1) {
System.out.println("test method");
System.out.println(t1.name);
System.out.println(t1.age);
t1.age = 40;
}

public static void main(String[] args) {

CallByRef obj = new CallByRef();
obj.name = "Rahul";
obj.age = 30;

System.out.println(obj.name);
System.out.println(obj.age);

CallByRef.test(obj);// call by reference

System.out.println(obj.age);

CallByRef obj1 = new CallByRef();
CallByRef.test(obj1);// call by reference

}

}
```

output:

```
Rahul // Initial obj.name
30 // Initial obj.age
test method // From test method
Rahul // t1.name (same as obj.name)
30 // t1.age before change
40 // obj.age after test method (changed via t1)
```

Constructor Assignments

Create a Java class named "Person" with the following instance variables:

- name (String)
- age (int)
- gender (char)
- height (double)

Create a constructor for the Person class that takes in the name, age, gender, and height as parameters and initializes the instance variables.

Create a main method that creates two instances of the Person class using the constructor and prints out their information.

Questions:

1. What is the purpose of a constructor in Java?
2. How does a constructor differ from a regular method in Java?
3. Can a Java class have multiple constructors? If so, how are they differentiated?
4. What happens if a constructor is not defined in a Java class?

- If you don't implement any constructor in your class, the Java compiler inserts default constructor into your code on your behalf. You will not see the default constructor in your source code (the .java file) as it is inserted during compilation and present in the bytecode (.class file).

1. Can a constructor call other methods or constructors within the same class?

Assignment 2:

Create a Java class named "Rectangle" with the following instance variables:

- length (double)
- width (double)

Create a default constructor for the Rectangle class that sets both the length and width to 0.0.

Create a constructor for the Rectangle class that takes in the length and width as parameters and initializes the instance variables.

Create a method in the Rectangle class named "calculateArea" that returns the area of the rectangle (length * width).

Create a main method that creates two instances of the Rectangle class using both constructors, calculates and prints out their respective areas.

Questions:

1. What is the purpose of a default constructor in Java?
2. How can you differentiate between a default constructor and a constructor that takes in parameters?
3. What is the access level of a constructor in Java?
4. Can a constructor be private? If so, why would you want to make a constructor private?
5. Can a constructor call a method from another class?

Assignment 3:

Create a Java class named "Employee" with the following instance variables:

- id (int)
- name (String)
- salary (double)

Create a constructor for the Employee class that takes in the id, name, and salary as parameters and initializes the instance variables.

Create getter methods for each of the instance variables.

Create a main method that creates an instance of the Employee class using the constructor, prints out the employee's information using the getter methods, and gives the employee a 10% raise using the setter method for the salary instance variable.

Questions:

1. What is the purpose of a getter method in Java?
2. Can a getter method return void?
3. What is the access level of a getter method in Java?
4. What is the purpose of a setter method in Java?
5. Can a setter method return a value other than void?

Assignment 4:

Create a Java class named "Car" with the following instance variables:

- make (String)
- model (String)
- year (int)

Create a constructor for the Car class that takes in the make, model, and year as parameters and initializes the instance variables.

Create a default constructor for the Car class that sets the make, model, and year to "Unknown".

Create a main method that creates three instances of the Car class using both constructors and prints out their information.

Questions:

1. What is the purpose of a default constructor in Java?
2. Can a default constructor take in parameters?
3. Can a constructor be overloaded in Java? If so, what does it mean to overload a constructor?

Assignment 5:

Create a Java class named "BankAccount" with the following instance variables:

- accountNumber (String)
- balance (double)

1. Create a constructor for the BankAccount class that takes in the accountNumber and balance as parameters and initializes the instance variables.

Create a method in the BankAccount class named "deposit" that takes in a double value as a parameter and adds it to the balance instance variable.

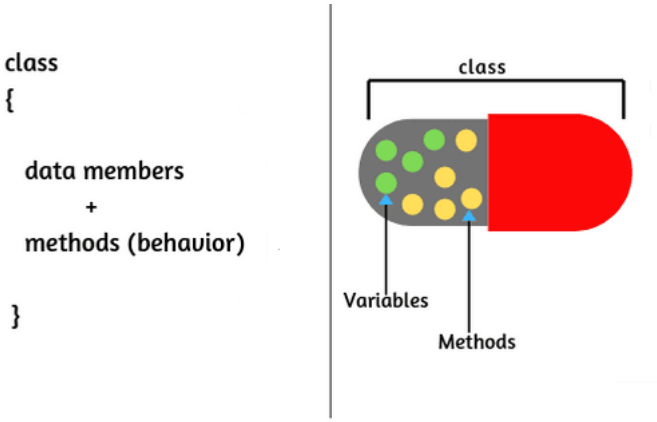
Create a method in the BankAccount class named "withdraw" that takes in a double value as a parameter and subtracts it from the balance instance variable.

Create a main method that creates an instance of the BankAccount class using the constructor and performs multiple deposits and withdrawals using the deposit and withdraw methods. Print out the updated balance after each transaction.

Questions:

1. What is the purpose of an instance variable in Java?
2. How are instance variables different from local variables in Java?
3. What is the access level of an instance variable in Java?
4. What is the purpose of a method in Java?
5. Can a method call other methods within the same class?

Encapsulation_ PrivateMethods

encapsulation	• Hiding data members of the class is called encapsulation. • We have hidden the data member variables inside a class and have also specified the access modifiers so that they are not accessible to the other classes.Thus encapsulation is also a kind of "***data hiding***". • Data and methods are enclosed in a single unit in encapsulation. • Hiding implementation of the logic / functionalities is called encapsulation. • Encapsulation acts as a protective shield around the data and prevents the data from unauthorized access by the outside world. In other words, it protects the sensitive data of our application.  Ex: ATM Machine ,Laptop(internal parts),capsule For access the private data we can use **setter** and **getter** method.
How to implement Encapsulation?	In Java, there are two steps to implement encapsulation. Following are the steps: 1. *Use the access modifier 'private' to declare the class member variables.* 2. *To access these private member variables and change their values, we have to provide the public getter and setter methods respectively.* 3. Now we can read the values of the private variables and set new values for these variables using getter and setter methods.

Use Private Methods via Public Methods using Encapsulation

```
package OOP_Encapsulation;

public class Browser {

    public void launchBrowser() {
        System.out.println("launching Browser");
        checkRAM();
        checkCPUUtilization();
        checkBrowserVersion();
        checkBrowserUpgrade();
        System.out.println("browser is
launched.....");
    }

    private void checkRAM() {
        System.out.println("checkRAM");
    }

    private void checkCPUUtilization() {
        System.out.println("checkCPUUtilization");
    }

    private void checkBrowserVersion() {
        System.out.println("checkBrowserVersion");

    }

    private void checkBrowserUpgrade() {
        System.out.println("checkBrowserUpgrade");
    }

}
```

Why to use Encapsulation?

- It provides you ***more control over the data.***
- It is a way ***to achieve data hiding*** in Java because other class will not be able to access the data through the private data members.
- By providing only a setter or getter method, ***you can make the class read-only or write-only.***
- The encapsulated class is ***easy to test***. So, it is better for unit testing.

private instance variable Protection	Setter(Mutator) Method	Getter(Accessor) Method
No Access	NO Setter	NO Getter method
Read-Only Access	NO Setter	Implement Getter method
Write-Only Access	Implement Setter method	NO Getter method
Read/Write Access	Implement Setter method	Implement Getter method

Private keyword

1. If you declare variable/method/class as a private then one can't access properties at the outside the class.
2. The scope of private is within the class only.
3. In order to access those properties indirectly in another class then we are using setters & getters methods.

Ex:

```
class emp
{
String ename;
int age;
private double sal;
}
psvm(string[] args)
{
emp e1=new emp();
e1.ename="peter";
e1.age=25;
e1.sal=30;//Scope :within the class
}
}
class test
{
psvm(String[] args)
{
emp e2= new emp();
e2.name="sam";
e2.age=30;
e2.sal=40.4;//compile time error because this is private variable the scope is within class only.To overcome this we are using getters & setters method
}
}
```

Solution-1

Using setters & Getters method

```
public Class Employee
{
public String ename;
public int age;
private double esal;
public void setsal(double sal)
{
this.sal=sal;
}
public void getsal()
{
}
}
Class Test
{
psvm(String[] args)
{
Employee e=new Employee();
e.setsal(20.45);
double salary=e.getsal();
syso(salary+300);
}
}
```

Note:

1. private keyword can be applied to variables, methods and inner class in Java.

Encapsulation Assignments:

Assignment 1:
Objective: The objective of this assignment is to create a class that uses encapsulation to protect its data and provide getter and setter methods for accessing the data.
Instructions:

1. Create a class called "Person" with the following private attributes: name (String), age (int), and gender (String).
2. Create getter and setter methods for each attribute.
3. Write a method called "printInfo" that prints out the name, age, and gender of the person.
4. Create an instance of the "Person" class and set its attributes using the setter methods.
5. Call the "printInfo" method to verify that the data was set correctly.

Questions:

- What is encapsulation and how does it relate to object-oriented programming?
- Why is it important to use getter and setter methods instead of accessing attributes directly?
- How can encapsulation improve the security and reliability of a program?
- What is the difference between a private attribute and a public attribute?
- How does encapsulation help with code maintainability and scalability?

Assignment 2:
Objective: The objective of this assignment is to create a class that uses encapsulation to hide its implementation details and provide a simple interface for clients.
Instructions:

1. Create a class called "BankAccount" with the following private attributes: accountNumber (String), balance (double), and owner (String).
2. Create getter and setter methods for each attribute.
3. Write a method called "deposit" that takes a double parameter and adds it to the balance.
4. Write a method called "withdraw" that takes a double parameter and subtracts it from the balance.
5. Write a method called "printStatement" that prints out the account number, owner name, and balance.
6. Create an instance of the "BankAccount" class and set its attributes using the setter methods.
7. Call the "deposit" and "withdraw" methods to modify the balance of the account.
8. Call the "printStatement" method to verify that the data was set correctly.

Questions:

1. How can encapsulation be used to hide implementation details from user of a class?
2. What are the benefits of using private attributes in a class?
3. What is the difference between a getter method and a setter method?
4. How can encapsulation improve the readability of code?

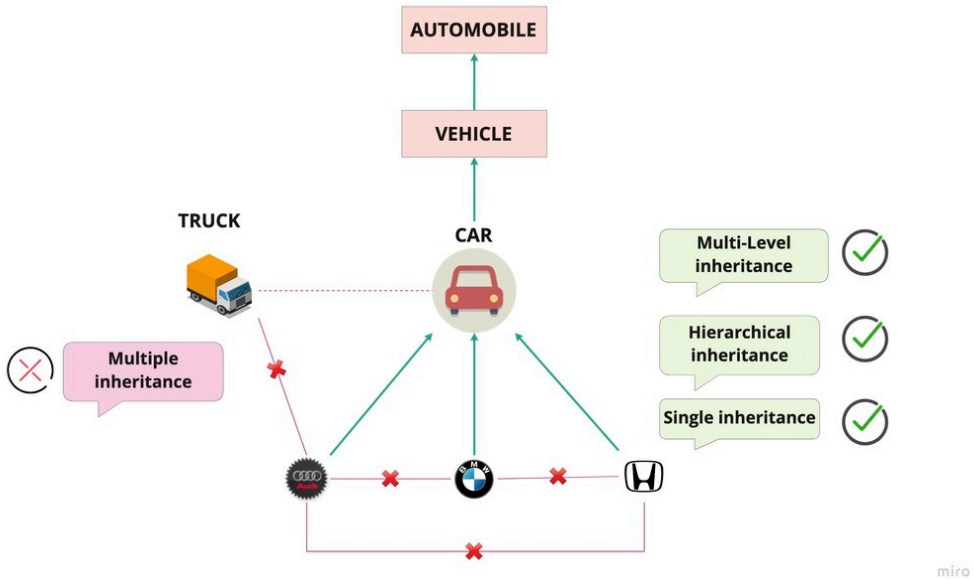
Inheritance_TopDown_Casting_MultiLevelInheritance MethodOverriding

- Parent(Super) class and Child(Sub) class
- 'extends' keyword usage
- Method Overriding in Inheritance
  - Static and Private method can't be overridden, only public method can be overridden.
  - Static and Private methods can be overloaded
- @Override annotation- Optional usage but Good practice
- Compile time Polymorphism- Static
- Run time Polymorphism - Dynamic
- *Top casting vs Down casting*
- *reference type check*

\ClassCastException

-

Diagrammatic view of Inheritance



NOTE:

- [ ] Inheritance represents the **IS-A relationship** which is also known as a ***parent-child*** **relationship.**
- [ ] Java supports ***Single/MultiLevel/ Hierarchical inheritances*** only.
- [ ] Java doesn't support Multiple and hybrid inheritance, They are achieved through interface only.

What is Inheritance?

- Its the mechanism in Java through which one class acquires all the properties and behaviors OR inherits the methods and fields of another class.
- you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Why we are using Inheritance?

- **For Method Overriding** (so runtime polymorphism can be achieved).
- **Code Reusability** : is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

"extends" keyword:

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

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

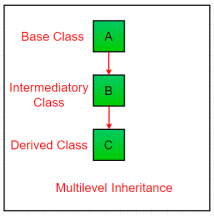
- ***Super Class/Base Class/Parent Class***: is the class from where a subclass inherits the features.
- ***Sub/Child/Derived/Extended Class:*** is a class which inherits the other class.

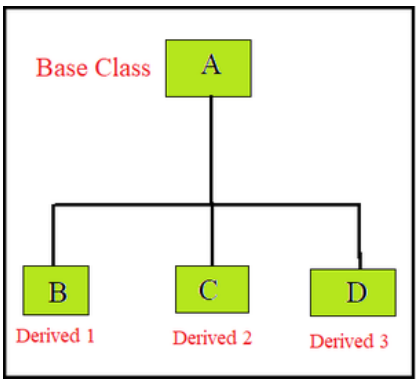
Inheritance Rules Parent-Child Relationships

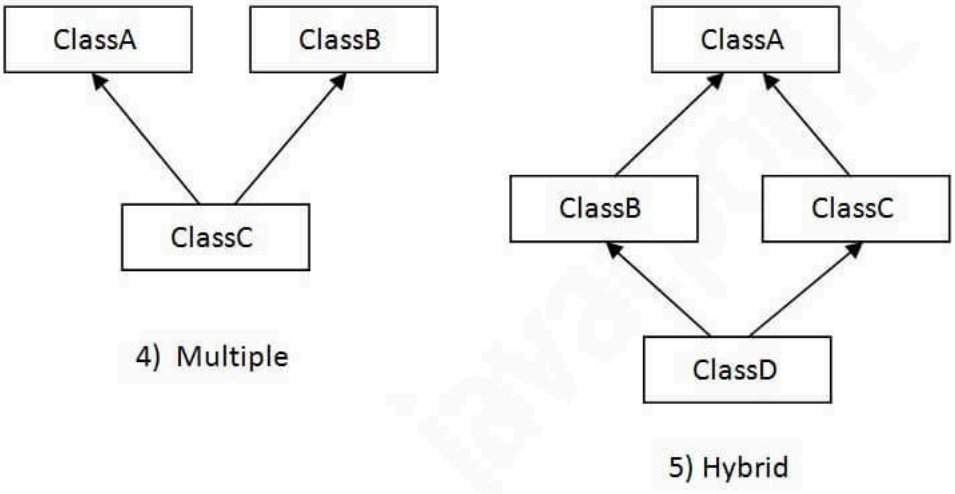
NOTE:

- [ ] Child classes/siblings cannot inherit properties of each other.
- [ ] Child can inherit properties from its parent but parent cant inherit childs properties.
- [ ] Parent class can inherit from its parent class and so a child can inherit from its grandparent class.
- [ ]

Single Level Inheritance	• When a single class inherits another class, it is known as a *single inheritance*.
	• **Single Level inheritance template**
	Class Car{
	}
	Class BMW extends Car{
	}

Multi Level Inheritance
 	
If method test() in class a is overridden in class b , but not overridden in class c , Class C referencing the method test - will be actually from the overidden test method in Class B
	• When there is a chain of inheritance, it is known as *multilevel inheritance*.
	• class c inherits method from both the base class as well as the intermediary class.
	 
	
	• **Multi level inheritance template**
	Class GrandParent{
	}
	Class Parent extends GrandParent{
	}
	Class Child extends Parent{
	}

Hierarchical Inheritance Or Linear Inheritance	• When two or more classes inherits a single class, it is known as *hierarchical inheritance*. 

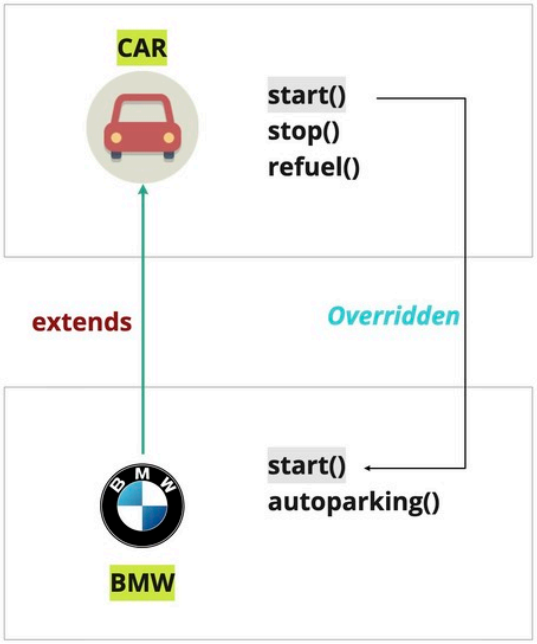
Multiple inheritance Hybrid inheritance	• Multiple inheritance and hybrid inehritance is not allowed in java through Class (Diamond Problem)
	 
	
	.
	Template
	
	Class A{
	}
	Class B{
	}
	Class C extends A, B{
	} // Not Allowed

Q) Why multiple inheritance is not supported in java?	• To reduce the complexity and simplify the language, multiple inheritance is not supported in java. • In above example where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class. • Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.
Method overriding Concept	• 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.***

Method overriding Exaample

- When you have a method in a parent class as well as in child class with the same name and same number of arguments is called **Method Overriding**.

Rules for method overriding




```
public class TestCar {

 public static void main(String[] args) {

  BMW b = new BMW();

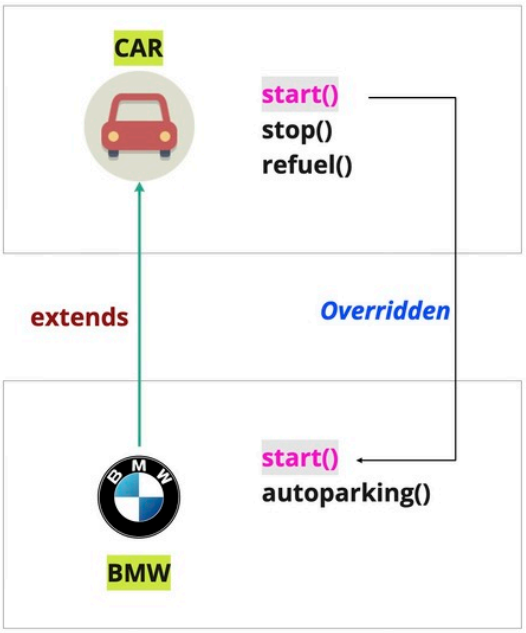
   b.start(); // overridden

   b.stop(); // inherited

   b.refuel(); // inherited

   b.autoParking(); // individual
```

miro



Rules for Method Overriding

- There should be inheritance between parent and child class
- Same method name
- Same return type
- Same number of parameters

miro

Can parent class access methods from child class?

- ☐ Parent class /grand parent cannot access any property or method from child class.

IMP: Private & Static methods can not be overridden in java

- ☐ **We can not override private & static methods in java**
- ☐ **We can not override methods with *final keyword* in java**
- ☐ **We can overload private, static methods in java**

Why can we not override static method?

- It is because the static method is bound with class whereas instance method is bound with an object.
- Static belongs to the class area, and an instance belongs to the heap area.
- ***So,Can we override java main method?***
  - No, because the main is a static method.

Why can we not override private method?

- **No, we cannot override private methods in Java**.
- Because Private methods in Java are not visible to any other class which limits their scope to the class in which they are declared.

Method Overloading vs Method Overriding

No.	Method Overloading	Method Overriding
1)	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.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	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.

Method Hiding concept

- Method hiding can be defined as, ***"if a subclass defines a static method with the same signature as a static method in the super class, in such a case,***

the method in the subclass hides the one in the superclass." In that case the method of superclass is hidden by the subclass.

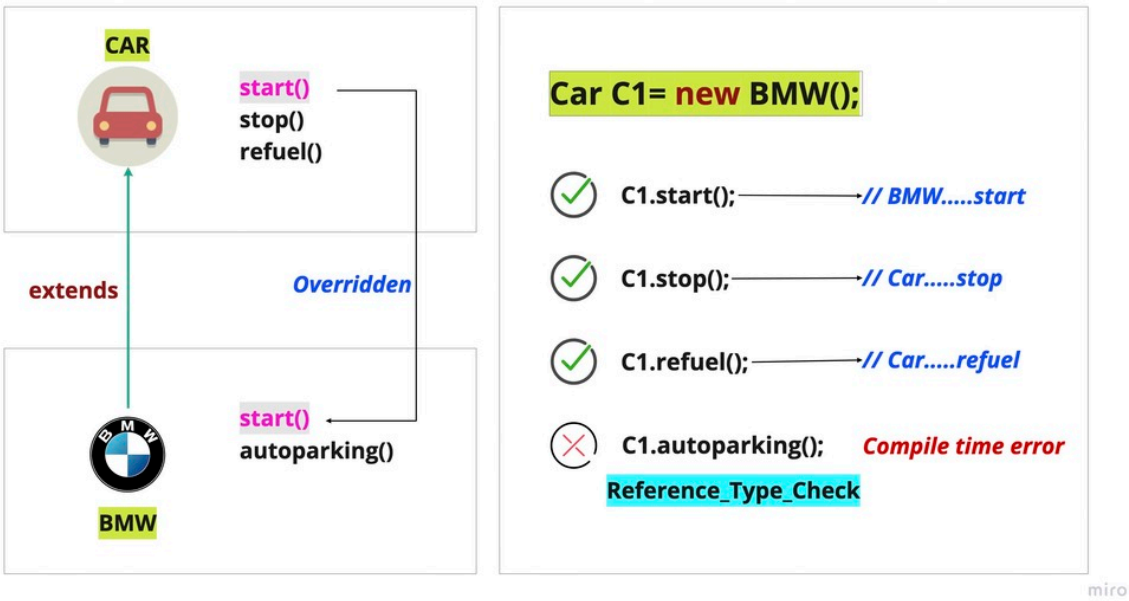
- It signifies that : The version of a method that is executed will NOT be determined by the object that is used to invoke it, since its static and is decided by compiler at compile time itself

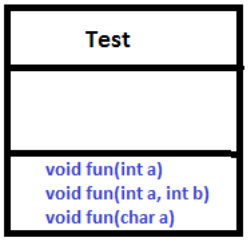
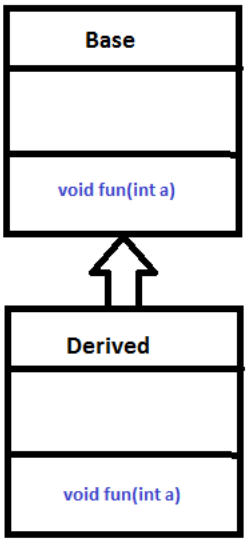
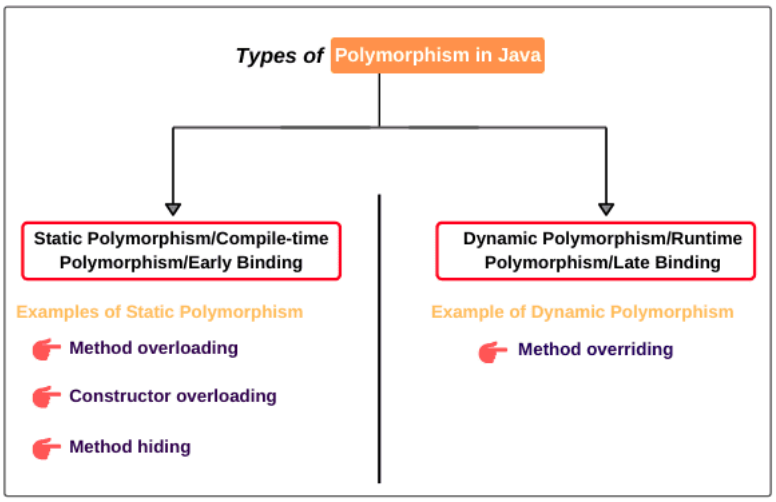
NOTE:

- ☐ Static methods are hidden, non-static methods are overriden.

Method Overriding Vs. Method Hiding

Method Hiding	Method Overriding
Both methods must be static.	Both methods must be non-static.
Method resolution takes care by the compiler based on the reference type.	Method resolution takes care by JVM based on runtime object.
It is considered as compile-time polymorphism or static polymorphism or early binding.	It is considered as runtime polymorphism or dynamic polymorphism or late binding.

TopCasting	• A child class object referred by **parent/super parent class** reference variable this concept is called top casting. • Casting does not change the actual object type. Only the reference type gets changed. • **Car c1= new BMW();**  **Note:** ☐ By using top-casting, ***we can inherit ONLY Overridden/Inherited methods but not individual methods.***
Why do we need Upcasting in Java?	• We need up casting when we want to write code that deals with only the parent class.
DownCasting	• Parent class object referred by any child class reference this concept is called as downcasting. • **BMW b1= new Car();** • **Honda h1=(Honda)new Car();** • Here compiler doesn't throw any error compile time but run time **we can get ClassCastException**
Class Cast Exception	• When we hold the object of parent class into reference variable of child class and then try to access parent class properties • then it always throw **"Class cast Exception" at runtime.** • (down casting; putting big box into small box, it can be done by tearing big box into small pieces and inserting, • so there will be no compile time error, but @ runtime we will not be able to access any thing properly)
Reference type check concept	• whenever we try to access individual method of child class with the reference variable of parent class, java will always try to match the ref type, this concept is called **"ref type check."**

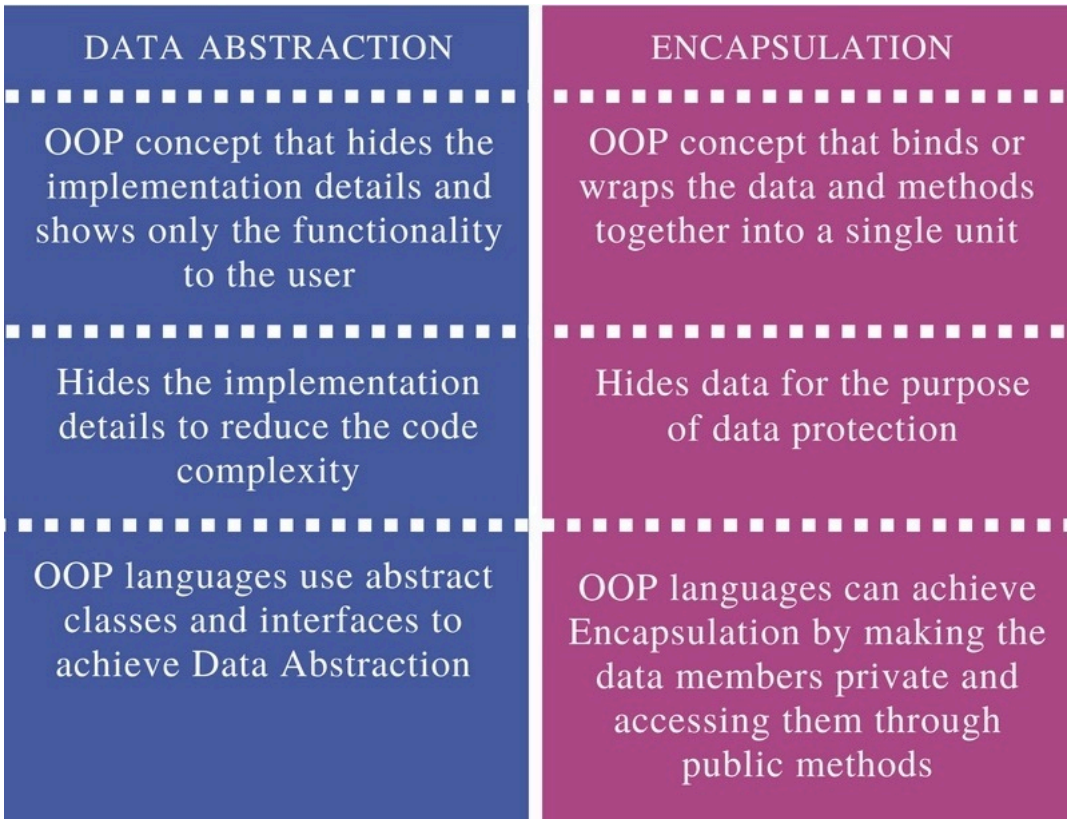
Polymorphism	• Polymorphism is a concept by which we can perform a single task in different ways. • The word “poly” means many and “morphs” means forms, So it means many forms. • Method Overloading & Method Overriding both will comes under polymorphism. • **Method Overloading is static/compile time polymorphism. Here compiler takes decision at compile time.** • **Method overriding always comes under runtime/dynamic polymorphism. Here decision has taken at run time only.**  Overloading  Overriding
Types of Polymorphism	

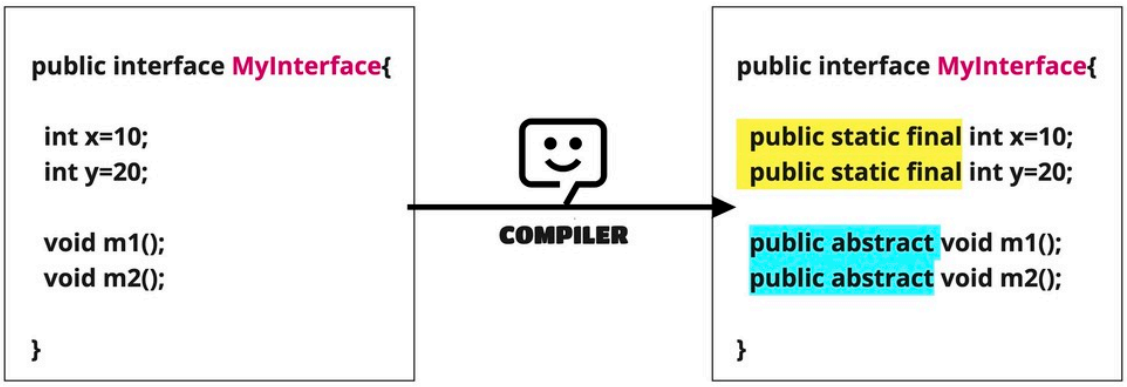
Abstraction_InterfaceConcept_MultipleInheritance_DiamondProblem

- Multiple Inheritance through Interface.
- Interface
- implements,extend keywords
- Abstract Method concept
  - No Method body
  - Only Method Declaration

- Class can implements multiple interfaces together and can be done by "Comma" separated.
  - Interface A
  - Interface B
  - Interface C
  - Class D implements A, B, C
- Can't create object of Interface
- **Down Casting is not allowed in interfaces, because we can't create object for Interface.**
- Abstract method can't be static & final in nature.
- Abstract method can't be declare as private.
- Interface variable by default Static and final in nature.
- After JDK 1.8 release,interface can have
  - Static method with method body
  - Default method with method body
- Default methods can be overridden, but make it public in child class.
- Interface can only have parent interface, it doesn't have parent class.
- Final method can't be overridden.
- Final keyword is used to provide constant values, to prevent Inheritance & as well as method overriding
- final variables are allowed in interface but final methods are not allowed in interface

Abstraction in Java	• Abstraction in Java is another OOPs principle that manages complexity. • It is a process of hiding complex internal implementation details from the user and providing only necessary functionality to the users. • In other words, abstraction in Java is a technique by which we can hide the data that is not required to a user. • It hides all unwanted data so that users can work only with the required data. It removes all non-essential things and shows only important things to users.
Why do we need Abstraction?	• It reduces the complexity of viewing the things. • Avoids code duplication and increases reusability. • Helps to increase the security of an application or program as only important details are provided to the user.
How to achieve Abstraction in Java?	• There are two ways to achieve or implement abstraction in java program. • They are as follows: 1. **Abstract class (0 to 100%)** 2. **Interface (100%)**

Difference between Abstraction and Encapsulation	
Interface ***What is an interface?***	• An interface is a collection of **abstract methods and constants (i.e. static and final fields).** It is used to achieve complete abstraction. • **Every interface in java is abstract by default.** So, it is not compulsory to write abstract keyword with an interface. • Once an interface is defined, we can create any number of separate classes and can provide their own implementation for all the abstract methods defined by an interface. • A class that implements an interface is called **implementation class**. A class can implement any number of interfaces in Java • interfaces cannot have business logic. • A class can implement more than one interface. • Interface cannot have parent class but it can have parent interface. • A class can extend and implement as well, but extend keyword should be followed by implements.
Why do we use interface?	1. It is used to achieve ***100% full abstraction IS-A relationship.*** 2. By interface, we can ***support the functionality of multiple inheritance.***

Interface Syntax	• accessModifier **interface** interfaceName {
	// declare constant fields.
	// declare methods that abstract by default.
	}
	
	NOTE:
	☐ **Interface fields are public, static and final by default.**
	☐ **Interface methods are public and abstract by default.**
	
	• The Java compiler adds **public** and **abstract** keywords before the interface method. Moreover, it adds **public, static** and **final** keywords before data members.
	
	 
	
	miro

Interface Variables	• **All the variables declared in an interface are considered as *public, static, and final* by default and acts like constant. We cannot change their value once they initialized.** • The variables will be available to any classes that implement interface because it is by default public, static, and final. • The values can also be used in any method as part of the variable declaration or anywhere in the class. • How to call directly : **InterfaceName.variableName**
Java JDK-8 onwards...	• We can also declare default methods and static methods with method bodies inside interfaces. • An interface can also declare its private methods.
Interface KeyPoints/ Features	1. Interface provides pure abstraction in java. It also represents the Is-A relationship. 2. It can contain three types of methods: abstract, default, and static methods. 3. All the (non-default) methods declared in the interface are by default abstract and public. So, there is no need to write abstract or public modifiers before them. 4. The fields (data members) declared in an interface are by default public, static, and final. Therefore, they are just public constants. So, we cannot change their value by implementing class once they are initialized. 5. **Interface cannot have constructors.** 6. The interface is the only mechanism that allows achieving multiple inheritance in java. 7. A Java class can implement any number of interfaces by using keyword implements. 8. Interface can extend an interface and can also extend multiple interfaces.

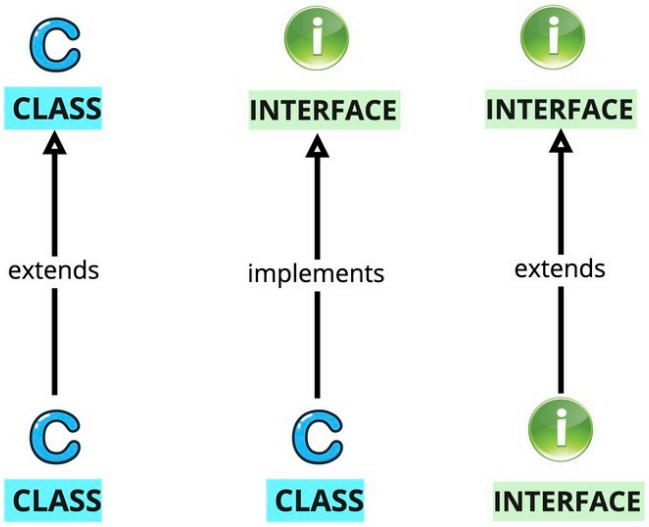
Interface Rules--

- We can't create an object of an interface But You can provide reference to the child class object with parent interface reference variable.
  - **USMedical us= new FortisHospital();**

    //TOP-CASTING IS ALLOWED
  - *By using **'us'** object reference you can access the methods from USMedical interface only.*

- Down Casting is not allowed in interfaces (not even @compile time) because you can't create an object of an interface.
  - **FortisHospital fh = new USMedical();**

    //DOWN-CASTING IS NOT ALLOWED



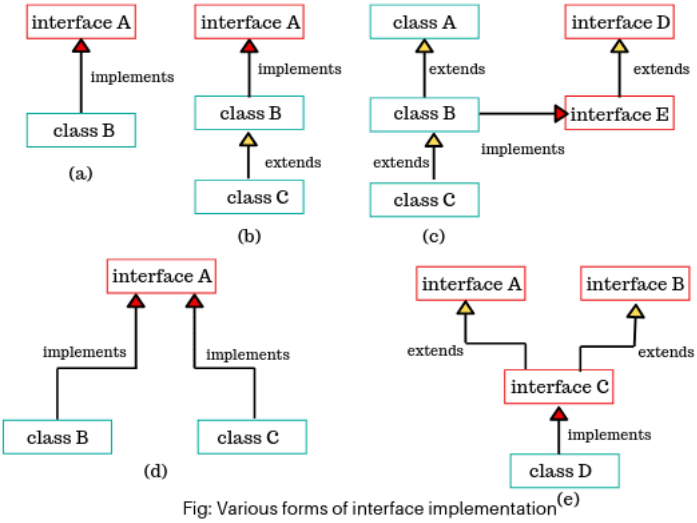
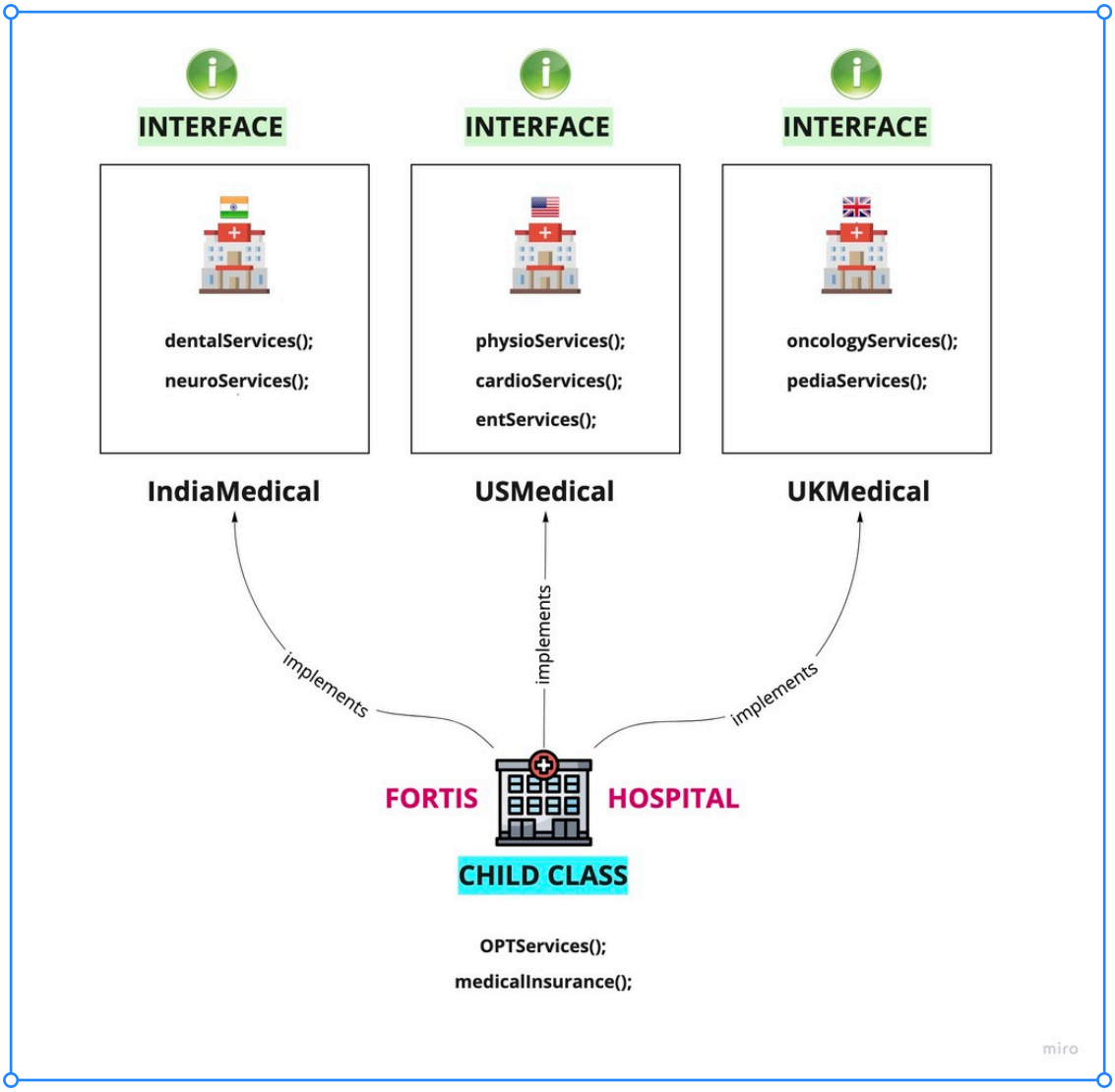
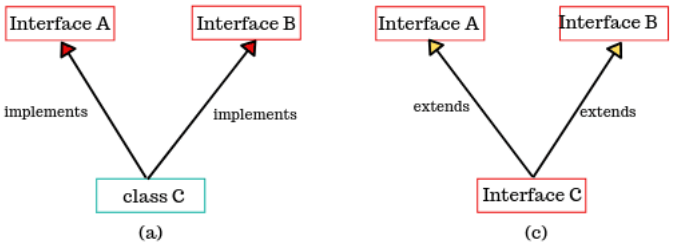


Fig: Various forms of interface implementation

- If we declare a variable in an interface, it must be initialized
- at the time of declaration. It cannot have instance variables.
- A class that implements an interface, must provide its own ***implementations of all the methods defined in the interface.***

- You should not declare interfaces with private/protected/final keywords or it will generate compile time error.

	• If you add any new method in interface, all the classes which implement that interface must provide implementations for newly added method because all methods in interface are by default abstract.
Interface illustration	
Can we have one common method for above 3 interfaces ?	• YES we can have it ,But it will be implemented only once in the child class which is implementation class of all 3 interfaces.
Can we declare interface's abstract methods as static?	• NO we can't declare abstract methods of interfaces as static because we can't override the static methods. • If we declare abstract methods as static we won't be able to provide its implementation in the child class so abstract methods can never be static.
Can we declare interface's abstract methods as private?	• NO we can't. • If we declare abstract methods as private then we won't be able to access them in implementation class as private methods can't be accessed outside of the class

Multiple Inheritance in Java by Interface	• Multiple inheritance in java is achieved through Interfaces using **implements** key word. • Interfaces specify what a class must do and not how. It is the blueprint of the class. • When a class implements more than one interface, or an interface extends more than one interface, it is called multiple inheritance.  Fig: Various forms of Multiple inheritance by Interface in Java

Java Class vs Interface	1. **Keyword**: a. A class is declared by using a keyword called class. b. An interface is declared by using a keyword interface. 2. **Instantiate**: a. A class can be instantiated. The keyword new can only create an instance of a class. b. An interface can never be instantiated. That is an object of interface can never be created. 3. **Variables**: a. Variables in a class can be declared using any access modifiers such as public, protected, default, and private. They can also be static, final, or neither. b. All variables in an interface are always public, static, and final. 4. **Methods**: a. Methods declared in a class are implemented. i.e, methods have a body. Methods that have a body is called concrete method. b. Methods declared in an interface cannot be implemented. i.e, In an interface, methods have no body. It contains only abstract method. c. ***Note***: Java 8 introduces default method, which allows us to give the body of a method in an interface. 5. **Access modifiers**: a. The members of a class can be declared with any access modifiers such as private, default, protected, and public. b. The members of an interface are always public by default. 6. **Constructors**: a. A class can have constructors to initialize instance variables. b. An interface cannot have any constructors. 7. **Inheritance**: a. Class allows only multilevel and hierarchical inheritances but do not support multiple inheritance. b. Interface supports all types of inheritance such as multilevel, hierarchical, and multiple inheritance. 8. **Implement**: a. A class can implement any number of the interface. b. Interface cannot implement any class. 9. **Extends**: a. A class can extend only one class at a time. b. An interface can extend multiple interfaces at a time. 10. **Use**: a. Interface is used for defining behavior that can be implemented by any class anywhere in the class hierarchy. b. A class is used to define the attributes and behaviors of an object.11. 11. **Final:** a. The method in an interface cannot be final because if you declare a method as final in interface, the method cannot be modified by any of its subclasses. b. A method in a class can be declared as final. 12. **Main method:** a. A class may contain main() method. b. Interface cannot have main() method.

In Java, Multiple Inheritance is not supported through Class but it is possible by Interface, why?	• In multiple inheritance, subclasses are derived from multiple superclasses.If two superclasses have the same method name then which method is inherited into subclass is the main confusion in multiple inheritance.That's why Java does not support multiple inheritance in case of class. • But, it is supported through an interface because there is no confusion. This is because its implementation is provided by the implementation class.

AbstractClass_WebDriver_With_Interface_RealTimeExample

- **abstract class explanation with real time example for webpage**
- **class to abstract class we can use extends keyword**
- **class to class ->extends**
- **class to interface->implements**
- **When to use abstract class & interface**
- **Difference between abstract class and interface**
- **abstract class allows abstract + non abstract methods**
- **we can't create objects for abstract class but we can create constructor for abstract class.**
- **constructor will call when you create object for child class**
- **default constructor created by JVM if you don't create a constructor in the child class**
- **example for interface concept is WebDriver implementation with different browsers**
- **static is not part of OOPS because we nevkker create objects for static properties**
- **abstract class contains once constructor & child class contains no constructor then sequence always first execute parent class constructor an9d then child class constructor**

Abstract Class

- An abstract class must be declared with an **abstract** keyword.
- It can have **abstract and non-abstract** methods.It can be abstract even without any abstract method.
- Abstract class allows to define private, final, static and concrete methods. Everything is possible to define in an abstract class as per application requirements.
- It cannot be instantiated.You Cannot create object of abstract class.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.
- Can create constructor of abstract class, This constructor will be called when object will be created of child class.
- It can implement one or more interfaces in java.

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.

NOTE:

- ☐ *You can't instantiate/create the object of both abstract class and interface.*
- ☐ *But you can have the constructor inside the abstract class while interface doesn't have any.*
- ☐ *The constructor of an abstract class will be called when you create an object of its child class.*

Abstract class vs Interface 12 points comparision

1. **Keyword(s) used:**
   a. Two keywords abstract and class are used to define an abstract class.
   b. Only one keyword interface is used to define an interface.

2. **Keyword used by implementing class:**
   a. To inherit the abstract class, we use the extends keyword.
   b. To implement an interface, we can use the implements keyword.

3. **Variables:**
   a. Abstract class can have final, non-final, static, and non-static variables.
   b. Interface cannot have any instance variables. It can have only static variables.

4. **Initialisation:**
   a. The abstract class variable does not require performing initialization at the time of declaration.
   b. Interface variable must be initialized at the time of declaration otherwise we will get compile-time error.

5. **Method:**
   a. Every method present inside an interface is always public and abstract whether we are declaring or not. That's why interface is also known as pure (100%) abstract class.
   b. An abstract class can have both abstract and non-abstract (concrete) methods.

6. **Constructors:**
   a. Inside an interface we cannot declare/define a constructor because the purpose of constructor is to perform initialization of instance variable but inside interface every variable is always static.
   b. Therefore, inside the interface, the constructor concept is not applicable and does not require.
   c. Since an abstract class can have instance variables. Therefore, we can define constructors within the abstract class to initialize instance variables.

7. **Static and Instance blocks:**
   a. We cannot declare instance and static blocks inside an interface. If you declare them, you will get compile time error.
   b. We can declare instance and static blocks inside abstract class.

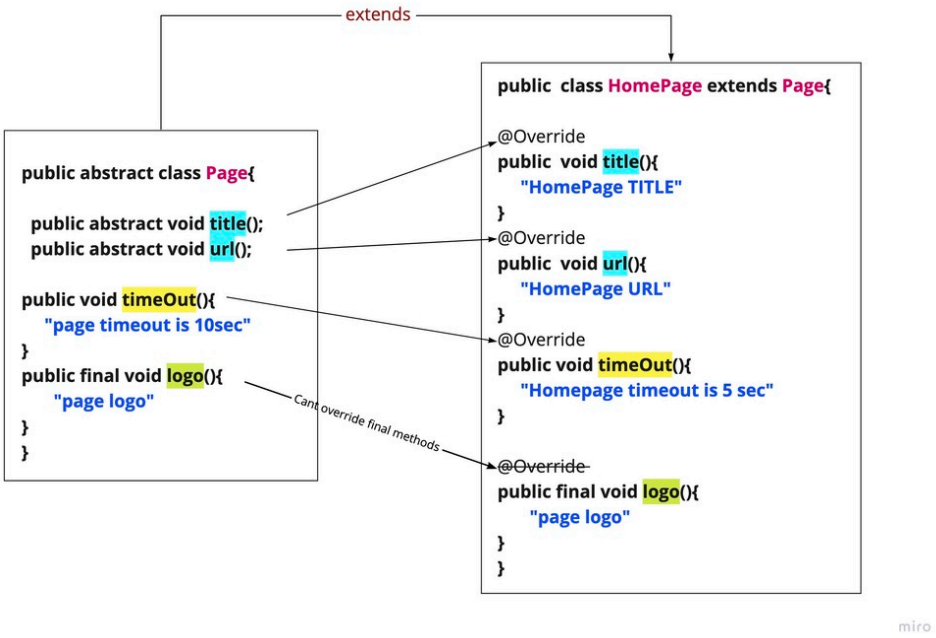
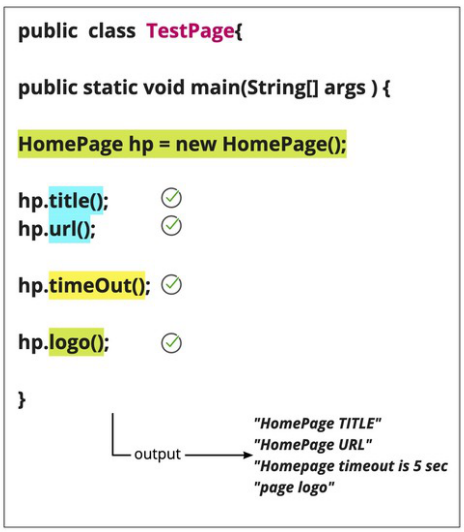
8. **Access modifiers:**
   a. We cannot define any private or protected members in an interface. All members are public by default.
      i. There is no restriction in declaring private or protected members inside an abstract class.

9. **Single vs Multiple inheritance:**
   a. A class can extend only one class (which can be either abstract or concrete class).
   b. A class can implement any number of interfaces.

10. **Default Implementation:**
   a. An abstract class can provide a default implementation of a method. So, subclasses of an abstract class can just use that definition but subclasses cannot define that method.
   b. An interface can only declare a method. All classes implementing interface must define that method.

11. **Difficulty in making changes:**
   a. It is not difficult to make changes to the implementation of the abstract class. For example, we can add a method with default implementation and the existing subclass cannot define it.
   b. It is not easy to make changes in an interface if many classes already implementing that interface. For example, suppose you declare a new method in interface, all classes implementing that interface will stop compiling because they do not define that method.

12. **Uses:**
   a. If you do not know anything about the implementation. You have just requirement specification then you should go for using interface.
   b. If you know about implementation but not completely (i.e, partial implementation) then you should go for using abstract class.

What is abstract method in java?	• A method which is declared as abstract and does not have implementation is known as an abstract method. • If there is an abstract method in a class, that class must be abstract. • If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract. ***NOTE:*** ☐ *It is mandatory to write 'abstract' keyword before abstract methods in abstract class ,but in case of interfaces it's not compulsory to write 'abstract' keyword before abstract methods because all the methods of an interface are abstract by default.*
Rules of Abstract method in Java	• Abstract method can only be declared in an abstract class. • A non-abstract class cannot have an abstract method whether it is inherited or declared in Java. • It must not provide a method body/implementation in the abstract class for which it is defined. • Method name and signature must be the same as in the abstract class. • Abstract method cannot be static or final. • It cannot be private because the abstract method must be implemented in the subclass. If we declare it private, we cannot implement it from outside the class.
NOTE:	☐ If a new abstract method is added in the abstract class, all non-abstract subclass which extends that abstract class, must implement the newly added abstract method. If it does not implement all the abstract method, the class must be declared as abstract. ☐ If a new instance method is added in the abstract class, all non-abstract subclass which extends that abstract class, is not necessary to implement newly added instance method.
Abstract class demonstration	 

Why abstract class has constructor even though we cannot create object of it?

- We cannot create an object of abstract class but we can create an object of subclass of abstract class. When we create an object of subclass of an abstract class, it calls the constructor of subclass.
- This subclass constructor has super in the first line that calls constructor of an abstract class. Thus, the constructors of an abstract class are used from constructor of its subclass.
- If the abstract class doesn't have a constructor, a class that extends that abstract class will not get compiled.

How Constructor is called in abstract class?

CASE 1:
When child class doesn't have user-defined constructor and abstract class has its own constructor

```
public abstract class Page {

//you can't create the object of abstract //class but
you can create a constructor.

public Page() {

System.out.println("Page class constructor");

}

}
```

```
public class HomePage extends page {

// doesnt have any user defined Constructor

}
```

```
public class TestPage{

public static void main(String[] args ) {

HomePage hp = new HomePage();

}
```

output

Page class constructor

miro

CASE 2:
When child class has one user-defined constructor + abstract class has one its own constructor

```
public abstract class Page {

//you can't create the object of abstract //class but
you can create a constructor.

public Page() {

System.out.println("Page class constructor");

}

}
```

```
public class HomePage extends page {

// doesnt have any user defined Constructor

public HomePage() {

System.out.println("HomePage class constructor");

}

}
```

```
public class TestPage{

public static void main(String[] args ) {

HomePage hp = new HomePage();

}
```

Page class constructor
HomePage class constructor

miro

- [ ] In the first line of constructor, internally super will call the constructor of an abstract class. The control of execution will be immediately transferred to the constructor of abstract class.Therefore, the first output is "Page Class constructor".
- [ ] After executing abstract class constructor, control of execution again comes back to execute subclass constructor. The second output is "HomePage class constructor".

CASE 3:
Overloading the abstract class constructor with same scenarios as in CASE 2

```
public abstract class Page {

//you can't create the object of abstract //class but you can create a constructor.

 public Page() {
 System.out.println("Page class constructor");
}

public Page(int a){
System.out.println("Page class a constructor"+a);
 }}
```

```
public  class HomePage extends page {

// doesnt have any user defined Constructor

 public HomePage() {

System.out.println("HomePage class constructor");

 }

}
```

```
public  class  TestPage{

public static void main(String[] args ) {

HomePage hp = new HomePage();

}
```

Page class constructor
HomePage class constructor

miro

CASE 4:
Overloading both abstract class constructor and child class constructor

```
public abstract class Page {

//you can't create the object of abstract //class but you can create a constructor.

 public Page() {
 System.out.println("Page class constructor");
}

public Page(int a){
System.out.println("Page class a constructor"+a);
 }}
```

```
public  class HomePage extends page {

 public HomePage() {

 System.out.println("HomePage class constructor");

 }
public Page(int a){
System.out.println("HomePage constructor"+a);
 }}
```

```
public  class  TestPage{

public static void main(String[] args ) {

HomePage hp = new HomePage(10);

}
```

Page class constructor
HomePage constructor 10

miro

NOTE:

- ☐ *It's mandatory to have a parent abstract class default constructor if you want to call child class constructor .*
- ☐ *If user doesn't define the constructor inside abstract class jvm will create a default constructor.*

- Abstract class provides Abstraction
- When there is constructor in child and parent class preference will be given to parent abstract class constructor
- It is compulsory to create default constructor in parent class.

- Zero Abstract /No abstract methods-0% Abstraction
- only abstract methods-100% Abstraction
- abstract methods+ non abstract methods-Partial abstraction

When to use abstract class ?

- When we talk about implementation but not completely then we should go for abstract class.

When to use Interface ?

- If we don't know anything about implementation just we have requirements/specification then we should go for interfaces

This vs Super Keywords:

Comparison of `this` and `super` Keywords in Java

Aspect	`this` keyword	`super` keyword
Purpose	References current class instance	References parent class instance
Constructor Call	`this()` calls current class constructor	`super()` calls parent class constructo
Field Access	`this.variable` accesses current class field	`super.variable` accesses parent cl field
Method Call	`this.method()` calls current class method	`super.method()` calls parent class method
Position in Constructor	Must be first statement if used	Must be first statement if used
Multiple Usage	Can be used anywhere in class	Can be used anywhere in inherited cla
Chaining	Can chain constructors in same class	Can call parent constructor only
Variable Shadowing	Resolves naming conflicts between local and instance variables	Resolves naming conflicts between pa and child class members

Example Code

```
class Parent {
    int value = 10;
    void display() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    int value = 20;  // shadows parent's value

    void display() {
        // using both this and super
        System.out.println("Child value: " + this.value);    // prints 20
        System.out.println("Parent value: " + super.value);  // prints 10
        super.display();  // calls parent's display method
    }
}
```

ExceptionHandling:

- Exception handling - Try Catch Mechanism
- Throws keyword
- Throw
- Difference between throw and throws keyword
- When to use throw and throws keyword.
- Difference between exception and error
- Error concept
- How to handle errors in the code?
- Good practice is to handle exception in the same method when you use throws keyword

Exception	• An unwanted or unexpected event that occurs during the execution of the program and interrupts the normal flow of program instructions. • These are the errors that occur at compile time and run time. • It occurs in the code written by the developers. It can be recovered by using the try-catch block and throws keyword.
Errors	• Errors are problems that mainly occur due to the lack of system resources. • It cannot be caught or handled. It indicates a serious problem. • It occurs at run time. These are always unchecked

Exception vs Error

Basis of Comparison	Exception	Error
Recoverable/ Irrecoverable	Exception can be recovered by using the try-catch block. An error cannot be recovered.	
Type	It can be classified into two categories i.e. checked and unchecked.	All errors in Java are unchecked.
Occurrence	It occurs at compile time or run time.	It occurs at run time.
Package	It belongs to java.lang.Exception package.	It belongs to java.lang.Error package.
Known or unknown	Only checked exceptions are known to the compiler.	Errors will not be known to the compiler.
Causes	It is mainly caused by the application itself.	It is mostly caused by the environment in which the application is running.
Example	**Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundException, NullPointerException, ArithmaticException	Java.lang.StackOverFlow, java.lang.OutOfMemoryError

What is Throwable is java?

- Throwable is a class which is derived from Object class, is a top of exception hierarchy from which all exception classes are derived directly or indirectly. It is the root of all exception classes.
- Throwable is parent class of Exception class.
  Error ,Exception both are children of Object class.

Checked Exceptions

- The exceptions that are ***checked by Java compiler at compilation time*** is called checked exception in Java.
- If a method throws a checked exception in a program, the method must either handle the exception or pass it to a caller method.
- Checked exceptions must be handled either by using try and catch block or by using throws clause in the method declaration. If not handles properly, it will give a compile-time error.
- List of Checked Exceptions in JavaA list of some important checked exceptions are given below:
  - ***ClassNotFoundException***
  - ***InterruptedException***
  - ***InstantiationException***
  - ***IOException***
  - ***SQLException***
  - ***IllegalAccessException***
  - ***FileNotFoundException, etc***

NOTE:

- ☐ *Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.*
- ☐ *All exceptions always occur at runtime only but some exceptions are detected at compile-time and some other at runtime.*
- ☐ *We should not handle the exception with the main method. The method which is responsible for the exception should handle the exception.*

Unchecked Exceptions

- **Unchecked Exceptions (Runtime Exceptions)** are checked by JVM, not by java compiler.
- They occur during the runtime of a program.
- All exceptions under runtime exception class are called unchecked exceptions or runtime exceptions in Java.
- We can write a Java program and compile it. But we cannot see the effect of unchecked exceptions and errors until we run the program.
- This is because Java compiler allows us to write a Java program without handling unchecked exceptions and errors.Java compiler does not check runtime exception at compile time whether programmer handles them or not.
- If a runtime exception occurs in a method and programmer does not handle it, JVM terminates the program without the execution of rest of the code.
- List of Unchecked Exceptions in JavaSome important examples of runtime exceptions are given below:
  - ***ArithmeticException***
  - ***ClassCastException***
  - ***NullPointerException***
  - ***ArrayIndexOutOfBoundsException***
  - ***NegativeArraySizeException***

Java Exception keywords

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

Exception Handling

- It's technique that allows us to handle runtime errors in a program so that the normal flow of the program can be maintained.

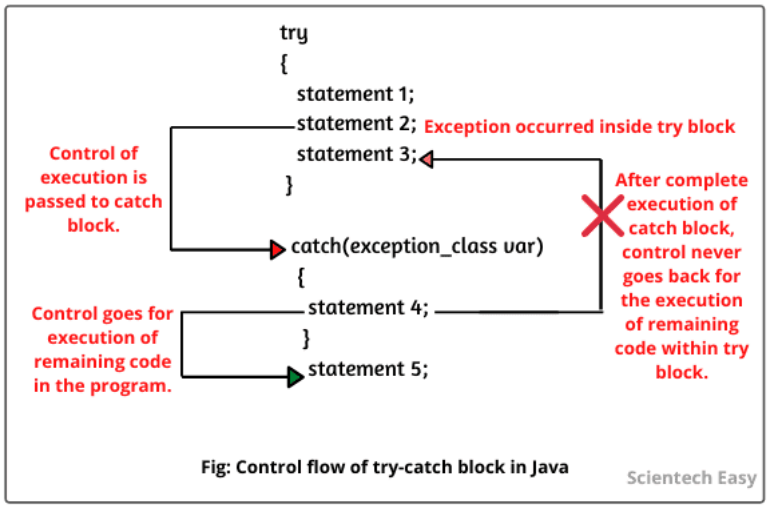
Try-Catch Block

- The Java code that may generate an exception during the execution of program, must be placed within a try block.
- We should place risky code that may generate exception inside try block. We should not keep normal code inside try block.
- When there is exception , the method in which exception occurs will create object and that object will store three things :
  a. **Exception Name** >> Name of the exception
  b. **Description** >> Cause of the exception
  c. **Stack Trace** >> Line no. where issue is coming

- Try >> We write Risky Code in Try
  Catch >> In Catch we writes the Handling code.

- Catch is block of code that handles the exception thrown by the try block. That's why it is also known as ***exception handler block***.
- A catch block that catches an exception, must be followed by try block that generates an exception.

Try Catch Mechanism




```
public class TryCatchEx1 {

    public static void main(String[] args) {

        System.out.println("11");
        System.out.println("Before divide");

        try {
            int x = 1 / 0;
            System.out.println("After divide");
        }

        catch (ArithmeticException e)
        // Here, e is a reference variable of exception object.
        {
            System.out.println("A number cannot be divided by zero");
        }

        System.out.println("22");
    }
}
```

Output:

11
Before divide
A number cannot be divided by zero
22

Multiple Catch Blocks

- A single try block can have multiple catch blocks. When statements in a single try block generate multiple exceptions, we require multiple catch blocks to handle different types of exceptions. This mechanism is called multi-catch block in java.
- Each catch block is capable of catching a different exception. That is ***each catch block must contain a different exception handler.***

Nested Try Catch J dava	• When a try block is defined within another try, it is called nested try block in java. • The try block which encloses another try block is called outer try block and the enclosed try block is called inner try block. ☐ **CASE1**: • If an exception occurs within outer try block, the control of execution is transferred from the outer try block to outer catch block that will handle the exception thrown by outer try block. • After handling the exception by catch block, the execution continues with the statement following the outer catch block. The inner try block and its corresponding catch blocks are skipped in this case. https://www.scientecheasy.com/wp-content/uploads/2020/05/nested-try-block.png ☐ **CASE 2**: • If an exception does not occur inside outer try block, the control of execution enters into the inner try block. If an exception occurs inside inner try block, the catch block associated with this inner try block is searched for a proper match. • If a match is found then the execution of outer catch block is skipped and the execution continues with statements following outer catch block. • If no match is found, the control of execution is transferred to the next outer try-catch block to handle the exception of inner try block. This process continues until and unless an appropriate match is found. • If no match is found then Java runtime system will handle the exception at runtime and the program terminates abnormally. https://www.scientecheasy.com/wp-content/uploads/2020/05/java-nested-try-catch-block.png
Key Points to Remember:	• An exception can be handled using try, catch, and finally blocks. • We can handle multiple exceptions using multiple catch blocks. • There can be a possibility for several exceptions inside the try block but at a time only one exception will be raised. • A single try block in Java can be followed by several catch blocks. • A catch block cannot be without try block but a try block can have without catch block. • **We cannot write any statement between try and catch blocks.**
Good Practice	• Multiple Catch Block instead of writing generic Exception way. • If there are multiple exceptions and multiple catch blocks, the first matching catch is executed and then finally block is executed. • we can have try without catch, but in this case there has to be finally block. • we can have nested try catch blocks also • We can have multiple try blocks • For each try block there can be zero or more catch blocks, but **only one** finally block. • It is not a **good programming practice to handle exceptions using Object/throwable/Exception** class • We can write specific exceptions followed by Exception class • We can use throwable class for catching Exceptions, but not Object.
Can we write try block without catch block?	• Yes we can use try block without catch block by using finally{} block • *try* *{* *statement1;* *statement2;* *}* *finally // finally block* *{* *statement3;* *}* ☐ **Rule**: For each try block there can be zero or more catch blocks, but only one finally block. ☐ **Rule:** Finally block must be defined at the end of last catch block. If finally block is defined before a catch block, the program will not compile successfully. ☐ **Rule:** Unlike catch, multiple finally blocks cannot be declared with a single try block. That is there can be only one finally block with a single try block.

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. https://static.javatpoint.com/core/images/java-finally-block.png ☐ If you don't handle the exception, before terminating the program, JVM executes finally block (if any). ☐ The finally block will not be executed if the program exits (either by calling ***System.exit()*** or by causing a fatal error that causes the process to abort).
Why use Java finally block?	• Generally, finally block or clause is used for freeing up resources, cleaning up code, db closing connection, io stream, etc. • The important statements to be printed can be placed in the finally block.
Finally block	• Ex: when System.exit(1) is encountered and executed , then finally code block is not executed • For each try block there can be zero or more catch blocks, but only one finally block for a try-catch block • The finally block is optional. It always gets executed whether an exception occurred in try block or not . • And if exception does not occur then it will be executed after the try block. • The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.
Conditions where finally block does not execute	• When System.exit() method is invoked before executing finally block. • When an exception happens in the finally block. • When the return statement is declared in the finally block, the control is transferred to the calling routine, and statements after return statement inside finally block will not be executed. ***NOTE:*** ☐ ***The return statement in finally block always overrides the return statements from try and catch blocks.***

Return statement in try block and finally block
 	
output
	`public class FinallyReturn1 {`
	
	`public int m1() {`
	
	`try {`
	`System.out.println("I am in try block");`
	`return 30;`
	`}`
	
	`finally {`
	`System.out.println("I am in finally block");`
	`return 50;`
	`}`
	`}`
	
	`public static void main(String[] args) {`
	`FinallyReturn1 obj = new FinallyReturn1();`
	`System.out.println(obj.m1());`
	`}`
	`}`
	
	I am in try block
	I am in finally block
	50

Return statement in catch block and finally block	**public class** FinallyReturn2 {
	  public int m1() {
	  try {
	  System.***out***.println("I am in try block");
	  int x = 10 / 0;
	  System.***out***.println("Result: " + x);
	  }
	  catch (ArithmeticException ae) {
	  System.***out***.println("I am in catch block");
	  return 40;
	  }
	  finally {
	  System.***out***.println("I am in finally block");
	  return 50;
	  }
	  }
	
	  public static void main(String[] args) {
	  FinallyReturn2 obj = **new** FinallyReturn2();
	  System.***out***.println(obj.m1());
	  }
	}

output	**I am in try block** **I am in catch block** **I am in finally block** **50**

Return statement in catch block and finally block but a statement after finally block

```
public class FinallyReturn3 {
    int m1() {
        int a = 20, b = 0;
        try {
            System.out.println("I am in try block");
            int c = a / b;
            System.out.println("Result: " + c);
        } catch (ArithmeticException ae) {
            System.out.println("I am in catch block");
            return 40;
        } finally {
            System.out.println("I am in finally block");
            return 50;
        }
        System.out.println("Statement after finally block");
        // This is unreachable code because its executing after return statement.you can't return anything after return statement

    }

    public static void main(String[] args) {
        FinallyReturn3 obj = new FinallyReturn3();
        System.out.println(obj.m1());
    }
}
```

output

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    Unreachable code

    at FinallyReturn3.m1(FinallyReturn3.java:21)
    at FinallyReturn3.main(FinallyReturn3.java:26)
```

Throw Keyword

- The Java throw keyword is used to throw an exception explicitly.If we want to throw an exception manually, for this, Java provides a keyword throw.
- Throw in Java is a keyword that is used to throw a built-in exception or a custom exception explicitly.
- Using throw keyword, we can throw either checked or unchecked exceptions in java programming.

☐ **Examples**:

- **throw new** *exception_name*("error message");
  - where *exception_name* is a reference to an object of Throwable class or its subclass.
  - Instances of classes other than Throwable class or its subclasses cannot be used as exception objects.
- **throw new** IOException("sorry device error");
- **throw new** NumberFormatException();

1. Only one object of exception type can be thrown by using throw keyword at a time. Throw keyword can be used inside a method or static block provided that exception handling is present.
2. Using throw keyword, we can throw either checked or unchecked exceptions in java programming.When an exception occurs in the try block, throw keyword transfers the control of execution to the caller by throwing an object of exception.

```
public class TestThrow1 {
    // function to check if person is eligible to vote or not
    public static void validate(int age) {
      if (age < 18) {
            // throw Arithmetic exception if not eligible to vote
       throw new ArithmeticException("Person is not eligible to vote");
      }
          else {
              System.out.println("Person is eligible to vote!!");
           }
    }

    // main method
    public static void main(String args[]) {
          // calling the function
          validate(13);
          System.out.println("rest of the code...");
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: **Person is not eligible to vote**

Throws keyword

- Throws keyword in Java is used in the method declaration. It provides information to the caller method about exceptions being thrown and the caller method has to take the responsibility of handling the exception.
- **Throws keyword is used in case of checked exception only.**

- **access_specifie** ***return_type*** method_name(parameter_list) **throws** exception1, exception2, ... exceptionN

```
{
  // body of the method.
 }
```

- Which exception should be declared?Ans: **Checked exception only**, because:
  - **unchecked exception:** under our control so we can correct our code.
  - **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Throw Vs Throws	• Throw is used for customised exceptions. • Throws keyword is used to delegate the responsibility of exception handling to the caller (It may be a method or JVM),in which case the caller method is responsible to handle that exception. • Exception should be handled immediately in the same method where its prone to failure, which makes easy for maintenance/debugging and for checking logs and fix bugs. • JVM will not handle exceptions if the exceptions are not addressed/handled. • We should not handle exceptions using throws inside TESTNG/main method.(difficult to trace incase of issues).

JavaSession Topic: **AccessModifiers_WrapperClass_Finally**

- **Data type conversion (by using wrapper classes)**
- **NumberFormatException example**
- **Find Max & minimum values for primitive data types by using wrapper classes**
- **Escape character example**
- **Access Modifiers (Default, Private, Protected, Public)**
- **finally** block example (Some interesting scenarios)
- **System.exit(1)** => JVM shut down.

Data type conversion example

Very Imp while performing Selenium Automation Execution as Most of data we get from UI is in the form of String.

- Integer.parseInt(String)
- Double.parseDouble(String)
- Boolean.parseBoolean(String)
- String.valueOf(Int/Double/Boolean)

Note: **String to Int/Double conversion is only applicable to Numeric type of Strings, it doesn't work with Alphanumeric content.**

Example: "AA100"

String to Int:-

```
String s = "100";
    System.out.println(s+20);//10020
int a = Integer.parseInt(s);
System.out.println(a+20);//120
```

String to Double

```
String s="100.00";
double d1=Double.parseDouble(s);
System.out.println(d1+20);//120.00
```

String to boolean

```
String s="true";
boolean b=Boolean.parseBoolean(s);
if(b)
{
System.out.println("Hi");
}
else
{
System.out.println("Hello");
}
```

Int to String:-

```
int i = 200;
       System.out.println(i+10);//210
       String t = String.valueOf(i);
    System.out.println(t+10);//20010
```

Boolean to String

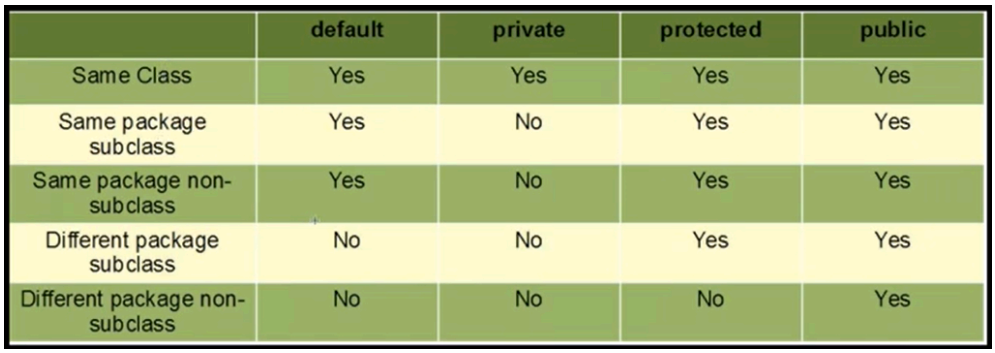
```
boolean b=true;
String b1=String.valueOf(b);
if(b1.equalsIgnoreCase("true"))
{
sopln("bye");
}
else
{
sopln("hi");
}
```

Note:

- parseInt,
- parsedouble,
- parseCharacter,
- parseByte,
- parseShort,
- parseBoolean,
- valueOf
  - all these methods are **static** methods. so we are accessing these methods with class name.

Integer,Double,Boolean,Character,Short,Long all these are predefined classes in java.

Max & Minimum Values for Primitive data types.	System.out.println(Byte.MAX_VALUE);//127
	System.out.println(Byte.MIN_VALUE);//-128
	
	System.out.println(Short.MAX_VALUE);//3277
	System.out.println(Short.MIN_VALUE);//-3278
	
	System.out.println(Integer.MAX_VALUE);//2147483647
	System.out.println(Integer.MIN_VALUE);//-2147483648
	
	System.out.println(Long.MAX_VALUE);//9223372036854775807
	System.out.println(Long.MIN_VALUE);//-9223372036854775808
	
	System.out.println(Float.MAX_VALUE);//3.4028235E38
	System.out.println(Float.MIN_VALUE);//1.4E-45
	
	System.out.println(Character.MAX_VALUE);//?
	System.out.println(Character.MIN_VALUE);//blank space
	
	System.out.println(Double.MIN_VALUE);//4.9E-324
	System.out.println(Double.MAX_VALUE);//1.7976931348623157E308

NumberFormatException Example:	String p = "AA100"; int r = Integer.parseInt(p);//**NumberFormatException** System.out.println(r); Here we have to extract the numeric string and then parse and use it
Escape character ("\")	String j = "Hi \"this\" is java"; System.out.println(j);//**Hi "this" is java** System.out.println("this is a \'test\&%'") throws "java.lang.Error: Unresolved compilation problem" Invalid escape sequence (valid ones are \b \t \n \f \r \" \' \\
Access Modifiers: **Used in Java to restrict the scope of variable, method or class.** **Note**: These behavior is applicable to Variables and Methods in similar manner.	 • Public --always YES • Private--always NO except from the same class • Protected--always YES except for different package and non subclass • Default --works only for same package but not from different package
Scope of different access modifiers.	• When no access modifier is defined for any class, method or variable, by default its "default" access modifier. • Keyword "private" and private means “only visible within the enclosing class”. • Keyword "public", and Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members. • Keyword "protected", Anything declared as protected can be accessed by classes in the same package and subclasses in the other packages.
Interview Questions on Access Modifiers	1. least restrictive access modifier in Java => **public** 2. most restrictive access modifier in Java? => **private** 3. access modifier is also known as Universal access modifier? **=> public**

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. - **Note 1**: We can write **finally** block without catch block also. finally won't catch exception. - **Note 2:** At any case program will return only ONE value. - **Note 3**: **finally** block without **try** block is not allowed. - **Note 4:** There are few cases where finally block will not execute, for example JVM is down. Application stopped abruptly...etc Ex: System.exit(1) //we won't use it - **Note 5**: If there is return statement condition satisfied and also there is finally block, then it will hold the return value and return value always executed lastly post finally block execution.

Real Time Example of Finally block.
 	
• Close DB Connections
• Close Excel files
• Driver.quit()
• Driver.close()
• close streaming objects
	`//db connection -- pass`
	`//pass sql string -- pass`
	`//try{`
	`//results from db -- exceptions // no excep`
	`//}`
	`catch() {`
	`some sql exception is coming`
	`}`
	`finally {`
	`//close db connection`
	`}`
	`//print the result from db`
	
	Reference: https://alvinalexander.com/blog/post/jdbc/-decent-example-of-using-try-catch-finally-with-jdbc/

Strings in Java:

- String Constant Pool mechanism
- Constructor- ArrayList and Array passing as Argument to Constructor
- split() method

What is String Constant Pool?	• String pool is nothing but a storage area in Java heap where string literals stores. • It is also known as ***String Intern Pool*** or ***String Constant Pool.*** It is just like object allocation. By default, it is empty and privately maintained by the Java String class. • Whenever we create a string the string object occupies some space in the heap memory. Creating a number of strings may increase the cost and memory too which may reduce the performance also. • The JVM performs some steps during the initialization of string literals that increase the performance and decrease the memory load. To decrease the number of String objects created in the JVM the String class keeps a pool of strings. • When we create a string literal, the JVM first check that literal in the String pool. If the literal is already present in the pool, it returns a reference to the pooled instance. If the literal is not present in the pool, a new String object takes place in the String pool.

Creating String in Java

There are two ways to create a string in Java:

1. ***Using String Literal***
2. ***Using new Keyword***

- ***Using String literals:***

```
String S1="Java";
String S2="Java";
String S3="Java";
```

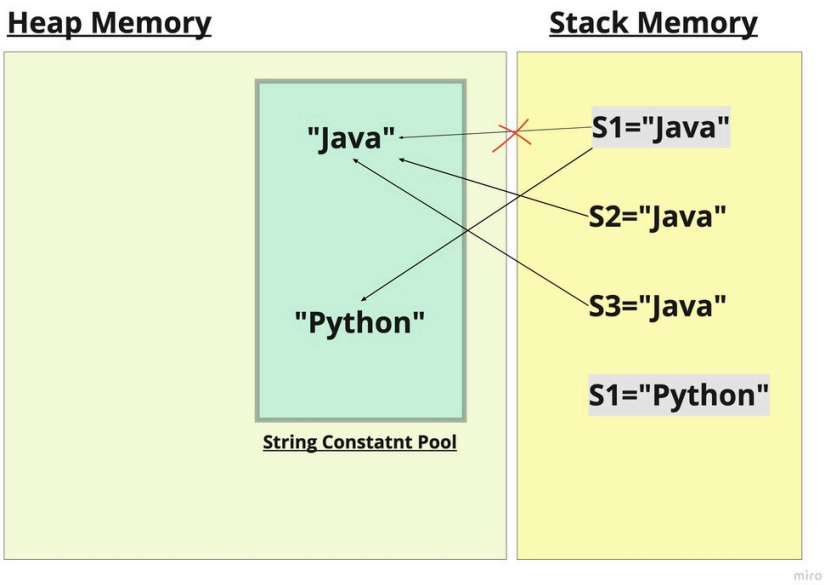
- How many objects will be created in memory? - 1 object.

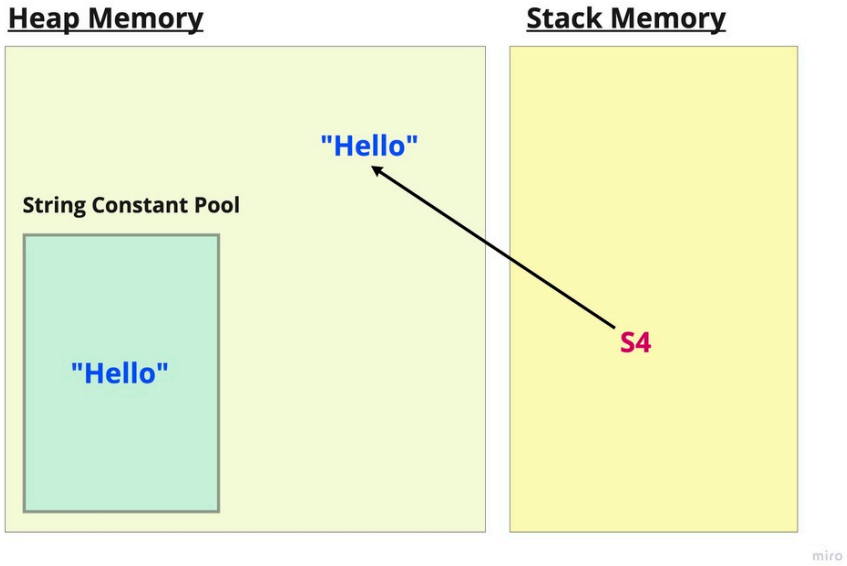
- ***Java Memory = Stack Memory + Heap Memory***
- ***String Constant Pool is a part of a Heap Memory.***

```
System.out.println(S1==S2); //true
S1="Python";
```

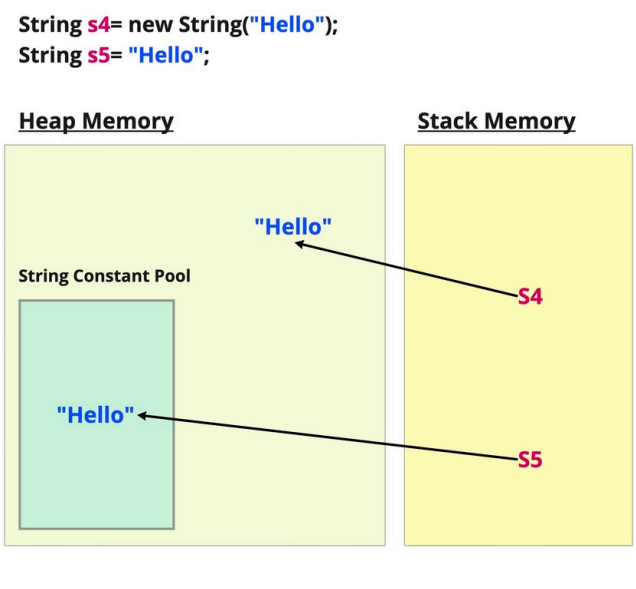
- Now How many objects will be created in memory? - 2 object.
- Now S1 will start pointing to "Python".

```
System.out.println(S1==S2);//false
```



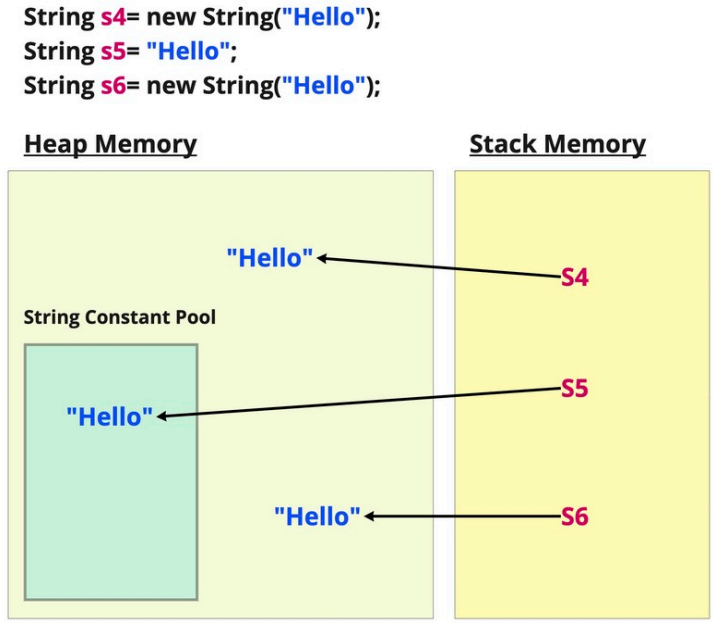
Creating String Using "new" keyword	• String s4= **new** String("Hello"); • How many objects will be created? => 2 objects • 1 object is in SCP, 1 object is in Heap memory. • S4 will be pointing to "Hello" inside the heap memory only and the "Hello" object inside SCP will not be referred by s4 

- String s5= "Hello";
- If "***Hello***" is already present then no new object reference is created inside the SCP.



miro

- String s6= new String("Hello");
- In above case only one object is created inside the heap as "Hello" reference is already present inside the SCP.



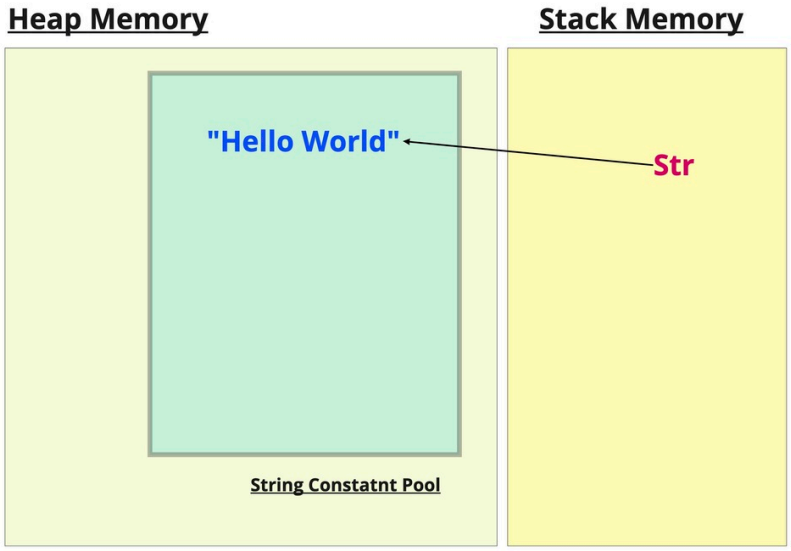
miro

- ```
  System.out.println(s4==s5); //false
  System.out.println(s5==s6); //false
  System.out.println(s4==s6); //false
  ```

- When we write like this:-
  String str = new String(); then it will always try to create 2 objects one in heap memory and one in SCP (for the first time).
  ```

Are Strings Mutable/Immutable?

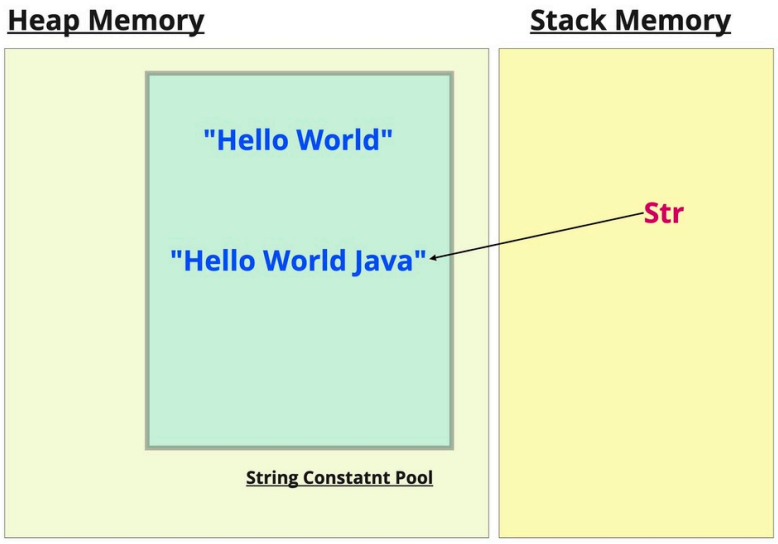
- String str="Hello World";



String str= "Hello World";

miro

```
str = str+ "Java";
System.out.println(str); //Hello WorldJava
```



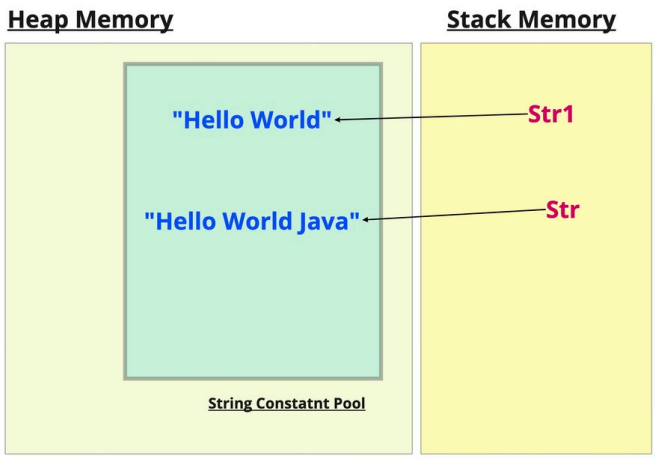
String str= "Hello World";

str= str+ " Java";

miro

- ***String is Immutable object, can't change the value, just pointing of reference is changed.***

- String str1="Hello World";
  System.out.println(str1); //Hello World




```
String str= "Hello World";
str= str+ " Java";
String str1= "Hello World";
```

- // No new object is going to be created, as its already present inside the SCP.

Why Strings are immutable in java	1. A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. 2. Why only Strings are immutable in java--Because strings are most commonly used data types. 3. ***Memory Optimisation***:The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently. 4. ***Security***:Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

GC mechanism present inside the SCP?	• Do we have GC mechanism with **String Constant pool**? => Yes, it also destroy the unused references from the SCP. • SCP is always having unique values, doesn't contain duplicate values.

split(): function Returns a String Array

The split() method of String class can be used to extract a substring from a sentence. ***It accepts arguments in the form of a regular expression.***

- String m="Java_Python_Ruby_C#";

```
String test[]= m.split("_");

System.out.println(test[0]);//Java
System.out.println(test[1]);//Python
System.out.println(test[2]);//Ruby
System.out.println(test[3]);//C#
System.out.println(test[4]);//ArrayIndexOutOfBoundsExcept
```

-
```
for(String e:test) {
 System.out.print(e+" ");
 }
```
  - Java Python Ruby C#

- String demo="xXJavaxXXPythonXxXRuby";

```
String[] top=demo.split("xX");

System.out.println(top[0]);//blank
System.out.println(top[1]);//Java
System.out.println(top[2]);//XPythonX
System.out.println(top[3]);//Ruby
```

Data driven Approach using String	• String empData = "tom; peter; 30; LA; USA; 909090";
	
	String emp[] = empData.split(";");
	
	for(String e:emp) {
	System.*out*.print(e);
	}
	• o/p: tom peter 30 LA USA 909090

String Assignments:

1. Write a program to check two different strings equality.
2. Remove all spaces in a String .

For example : “ Hello Everyone “ . Expected result: “HelloEveryone”.

3. Write a program that will print out the last character and first character of a word.

4. Write a program to verify a word or a character contained in the sentence.

5.Write a function/ method to reverse your own name.

6. Write a program that gives you the last half of the string.

7.Write a program to get the 3rd “ e “ of the string .

For example: “Welcome to Navеen Automation Labs ! “.

8. Write a method which gives an index of (-1) if string is not available. . it should return integer. if String is present, then it should return the specific index.

9. Write a program that breaks a whole string into small strings, and prints out its all values . (Hint: split, loop) .

10. Assume that a string consists of 3 words, print out the middle one.

11. get only numeric part from this String:
String s = "your transaction id is: 12345 and reference id is 34567";

ArrayList Concept:

Topic Name	Details	Written By

ArrayList	ArrayList- default class It is dynamic array • No defined using square bracket. (Square brackets are only for static arrays) • Size is not fixed • index based //Create the object of the ArrayList Class • ArrayList ar = new ArrayList (): • There is virtual capacity created for ArrayList defined in the memory • Physical capacity is the actual used memory • ar.size() will give physical capacity • ArrayList is not **synchronized**. That means, multiple threads can use same ArrayList simultaneously.	@Naveen AutomationLabs
Adding Virtual capacity	Initially java will allocate 10 Virtual capacity. so default virtual capacity is 10 When the virtual capacity is reached to max allocated, then additional (max physical capacity /2) virtual capacity is added.	@Naveen AutomationLabs

Generics in Java	ArrayList<Object> empData = new ArrayList<Object>();// for object
	ArrayList<Integer> empNum = new ArrayList<Integer>();// for Integer
	ArrayList<String> empName = new ArrayList<String>();// for String
	ArrayList<Double> empSalary = new ArrayList<Double>();// for Double
	ArrayList<Character> emp = new ArrayList<Character>();// for Character
	
	ArrayList<Boolean> empStatus = new ArrayList<Boolean>();// for Boolean
		@Naveen AutomationLabs

Dynamic Array → ArrayList

To overcome the static array problems related to memory, we use dynamic array for memory to be allocated during runtime. So we use **ArrayList**.

The size of the array will be automatically adjusted by Java when we add or remove data from arrayList.

ArrayList is a built-in class. We need to create its object to store data in ArrayList, as given below:

ArrayList ar = new ArrayList();
ar.add(10); //add() function is used to add data into arraylist object.
ar.add("Naveen");

Whenever a new arraylist object is created, java allocates 10 virtual memory segment capacity to it (index 0 to 9). Here physical memory is not allocated, the memory is virtual so no bytes allocated.
As and when we start adding the data into the segments, the physical memory starts getting filled and hence the bytes are allocated as per the type of the data.
When the entire initially allocated 10 virtual capacity segments are filled physically by data, the java again allocates it '**total physical memory consumed / 2**' i.e the second time, additional 5 virtual capacity will be allocated.
Once, additional 5 virtual segments are also filled, then additional 7 virtual segments capacity will be allocated (i.e., 15/2=7).

Note: Whatever arraylist functions are there, they will be applied on physical memory capacity and not virtual memory capacity. i.e., let's say initially VC is 10, as allocated by java. If we are adding only 2 data into arraylist, and if we write **ar.size()**, then this will give the result as 2 which is as per it's PC.

* If we want java to allocate virtual space as suggested by us, java will do that also. Example below:

ArrayList ar = new ArrayList(20);
Here, java will allocate 20 virtual memory segments initially and then as per the rule when the entire 20 segments are filled by physical data, then current 'capacity/2' virtual memory will be allocated the second time if more than 20 data have to be inserted.

Note: There is no method exposed to user to determine virtual memory capacity unlike physical memory capacity for which we use **ar.size()** method.

* **Use case for arraylist:**
In amazon or any e-commerce application, the number of products might vary based on the different occasions. Let's say during Christmas the product count is 500, then in normal days the product count is 100, then during good Friday, the product count again increases to 400. Here using arraylist would be a wise choice.

* **remove () function:**
There is remove() function, which helps in deleting a data at a specific index location. Example below:

ar.remove(3); //after this statement, the arraylist size will be shrinked. If the size of arraylist earlier to execution of this statement was 5, then now it will become 4 and the data earlier present at 4th index will be shifted to 3rd index.

get() function:
This function helps in getting the data at a specific index in arraylist.

ar.get(3);

```
ArrayList ar = new ArrayList();
ar.add(10);
ar.add("Diksha");

for(Object e: ar){ //Note: the ArrayType is raw (nt generics defined), so the data is Object type. Raw arraylist is never recommended. Always use Arraylist with generics.
      sysout(e);
}
```

* Generics:

ArrayList<Integer> ar = new ArrayList<Integer>(); //We can store only int type of data. Example: Employee ID of all employees.

ArrayList<String> ar=new ArrayList<String>(); //We can store only String type of data. Example: All employee names.

ArrayList<Object> ar=new ArrayList<Object>(); //Above we had seen raw arrayList wherein we could store any type of data. Since raw ArrayList is not recommended, we should use 'Object' generic type to store any type of data in arraylist.

Note: Doesn't matter what generics we are using, the default virtual capacity(VC) will always be 10.

Note: If we try to print the static array variable say, 'ar', directly by using sysout(ar), then a garbage value will be printed on the console. But if we try to print the dynamic array variable say, 'ar', directly by using sysout(ar), then all the values will be printed like -> [10, Diksha, ..].

We know that we can use 'indexed for loop' or 'for each loop' to print the values of arraylist, we use get() method to print the values in indexed for loop. When using for each loop, get() function is not required. Examples for both given below.

```
ArrayList<String> ar=new ArrayList<String>();
ar.add("Naveen");
ar.add("Robin");

for(int i=0;i<ar.size();i++){ // 'indexed for loop' to print all the arraylist values

      sysout(ar.get(i));

}


for(String name: ar){ // 'for each loop' to print all the arraylist values
      sysout(name);
}
```

*Why should we use generic types other than 'Object' generic types ?
-> we should use generic types other than 'Object' generic types to restrict the type of data to be stored in an arrayList. Example if an arrayList should have only names, then this arrayList should be created with generic type <String> so that nobody can enter the name like '200' which is integer value.

Note: We can add duplicate values in an arrayList.

* We can use add() method in 2 ways as below:
add(value to be added);

	* When we try to use add method like -> **add(index, value to be added);**, we should give the index as per the physical capacity and not virtual capacity. Let's see an example:
	ArrayList<Integer> ar = new ArrayList<Integer>();
	ar.add(200);
	ar.add(5, 300); // This will throw IndexOutOfBoundException because the PC of arrayList is 1 here, so we should add the data to next available position, i.e., index 1.
	
	If there is a need to update the list element based on the index then set method of ArrayList class can be used.
	set(index, value to be added);
	ex: ar.set(1, 100);	