Sept Batch 2024 Notes

Important Download Links

JDK Download URL:	https://www.oracle.com/in/java/technologies/javase/jdk11-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.

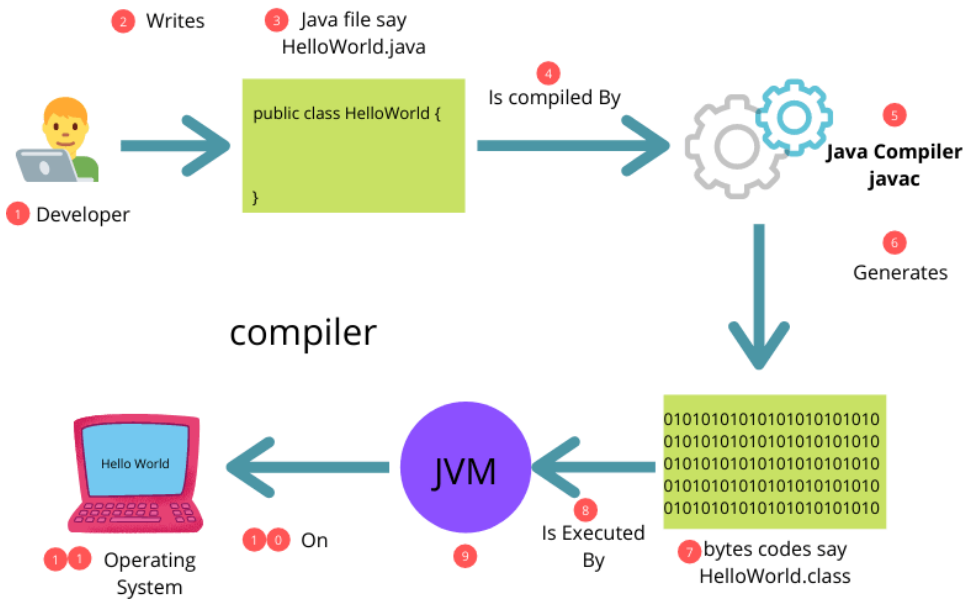
- [ ] The Java source code file (.java) is first compiled into Java bytecode (.class) using the Java compiler (javac). The bytecode is a platform-independent code that can be executed on any system with a Java Virtual Machine (JVM).

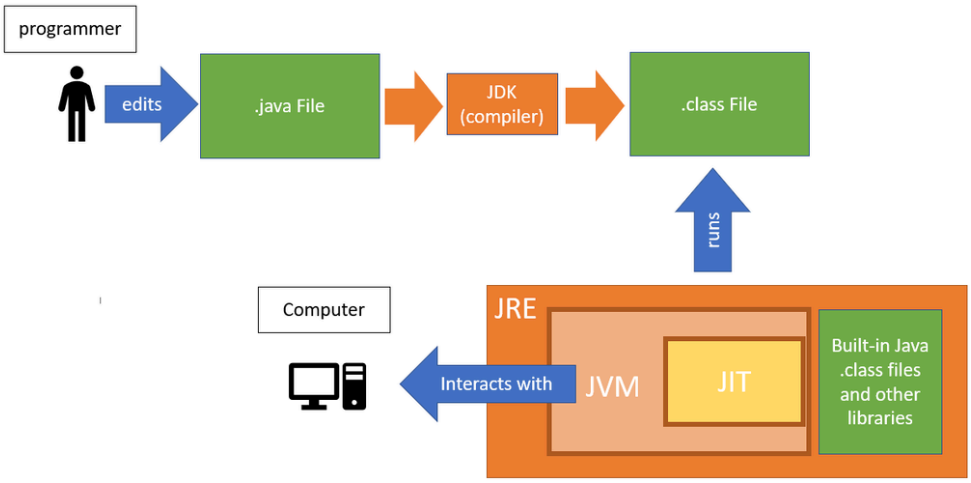
- When a Java program is executed, the JVM loads the bytecode into memory and runs it. The JVM interprets the bytecode and executes the corresponding machine code instructions on the underlying system. The JVM also provides a runtime environment for the program, which includes memory management, security,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



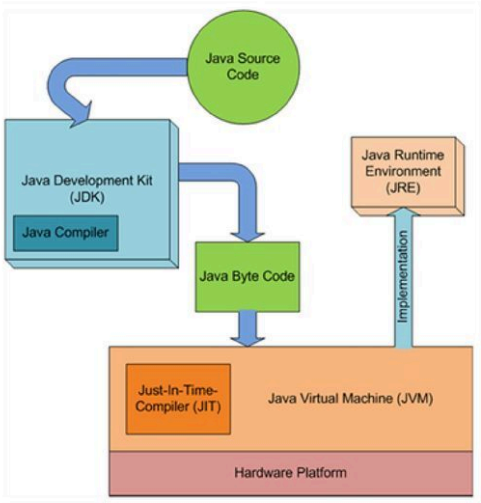


Difference between JDK/JRE/JVM/JIT

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.



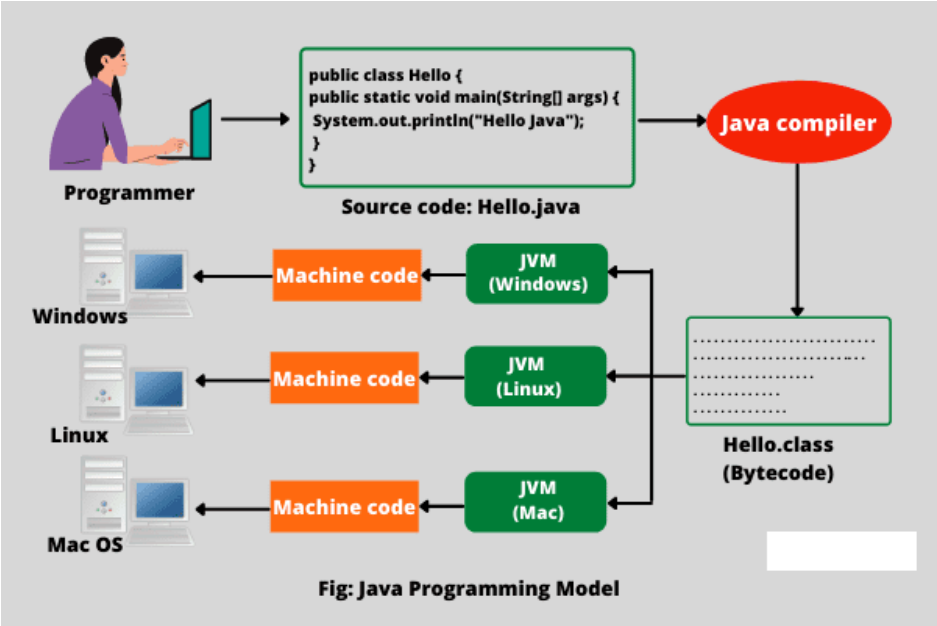
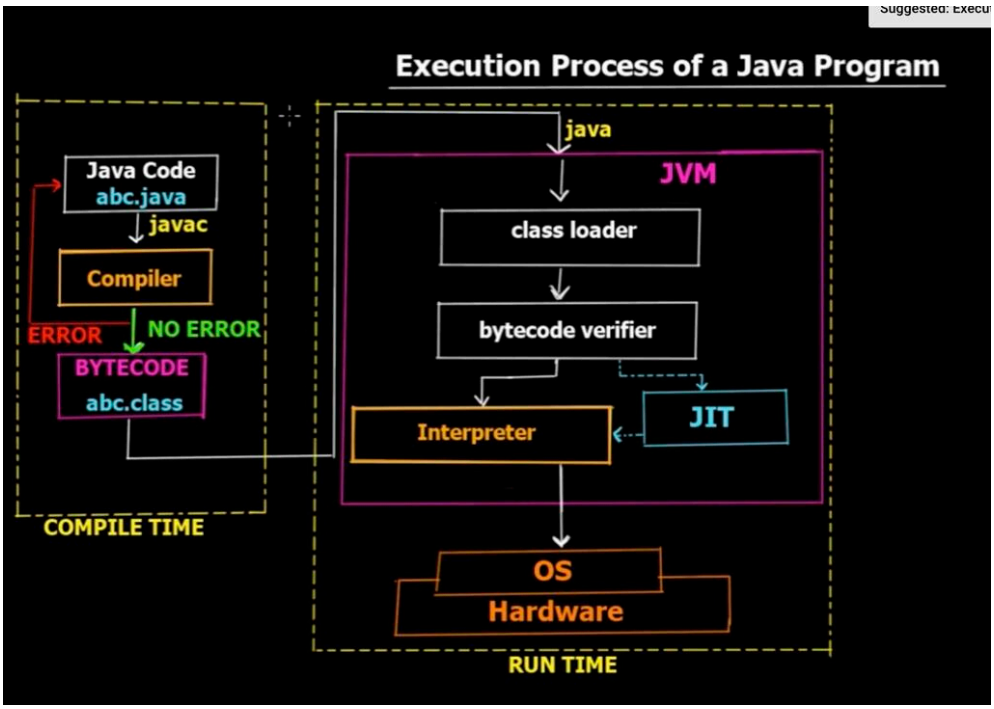
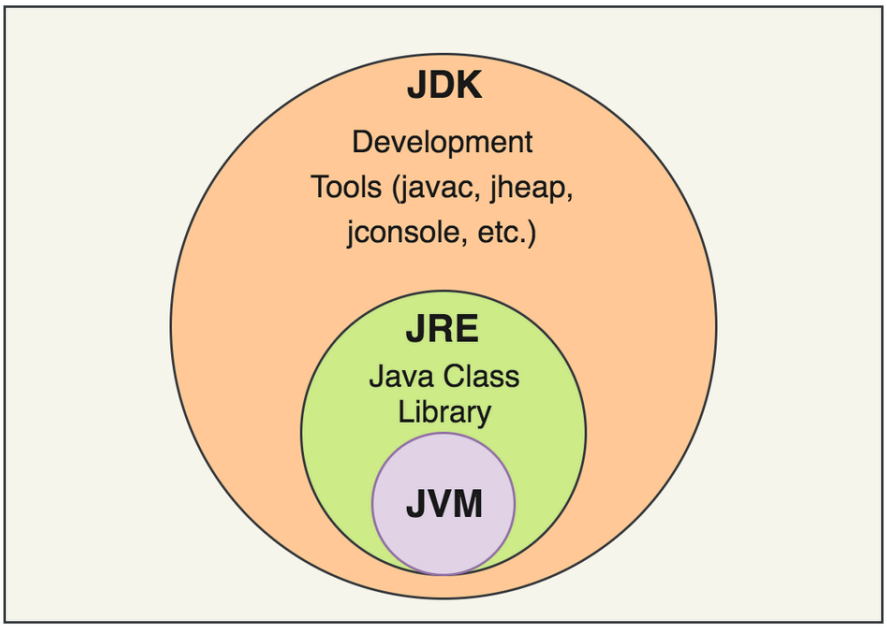


Fig: Java Programming Model



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

Execution Environment	Yes	Yes	Yes
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 InteiJ 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 Types : There are 8 primitive data types in Java.

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

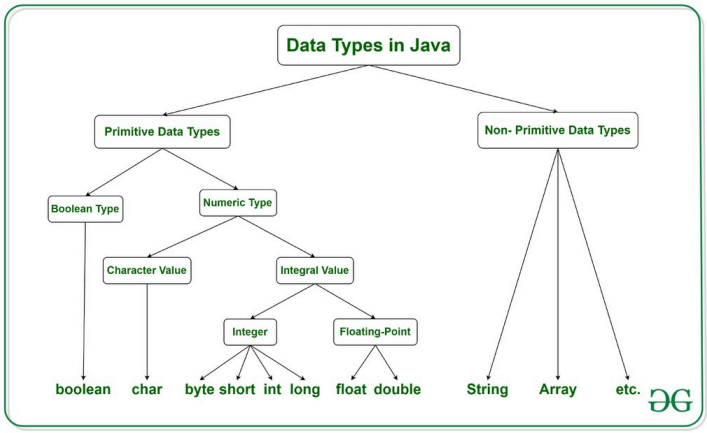
Topic Name	Details

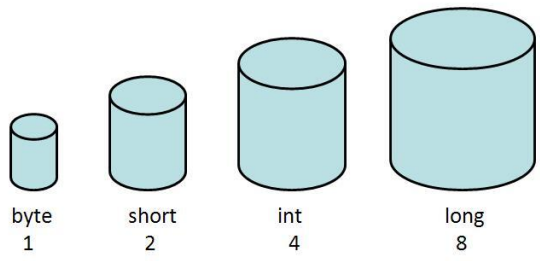
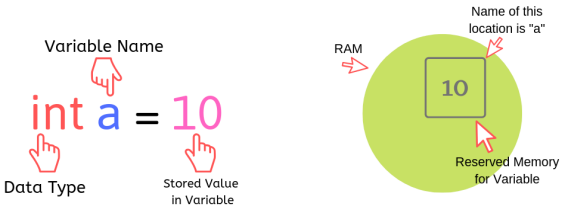
Difference between primitive data types and non primitive data types

- 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:	

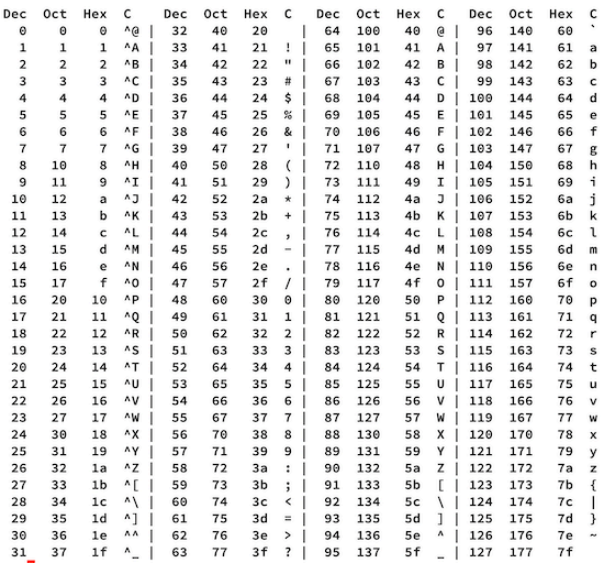
• 1 byte = 8 bits **How Memory allocated for a primitive data type:** **ex: int a = 10;**	 

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	
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	

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.

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=0;

        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 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++;

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

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

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

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

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);//1.5 + 3.5 = 5.0
    }
}
```

Type Casting:

Type Casting

Types of Data Types

Primitive or Fundamental Data Types
Referenced or Advanced Data Types

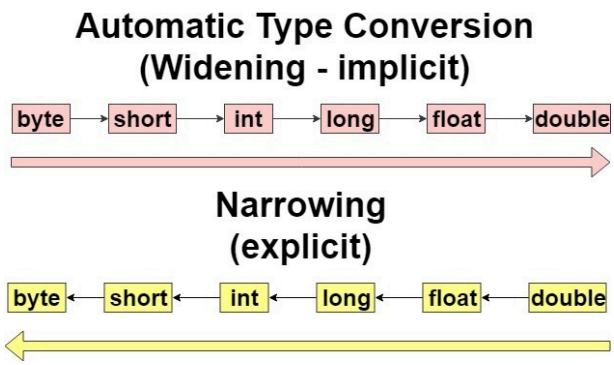
Casting Primitive Data Type

- **Widening:** Converting a lower data type into higher data type is called widening.
- **Narrowing:** Converting a higher data type into lower data type is called narrowing

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 IEEE 754 floating point.
6. **double** - This is a 64-bit IEEE 754 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 should be done carefully to avoid data loss.

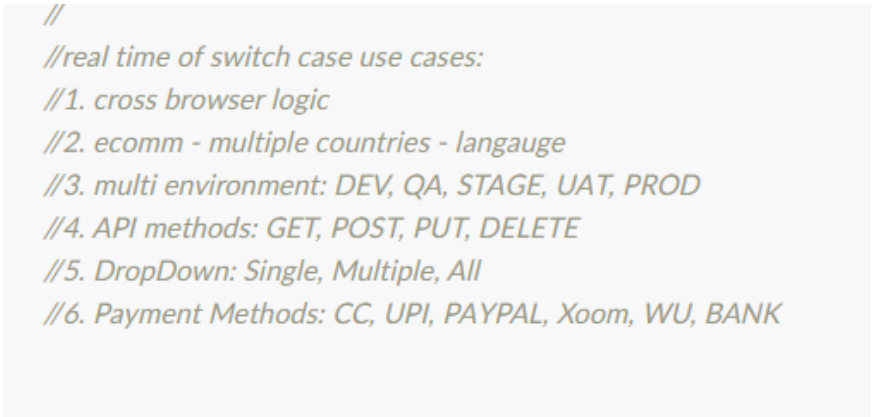
Widening Conversion Examples	// Example 1: From integer to floating-point types
	int myInt = 100;
	long myLong = myInt; // No casting required
	float myFloat = myLong; // No casting required
	double myDouble = myInt; // No casting required - int to double is also widening
	
	System.out.println("Int value: " + 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'	

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 - 
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

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

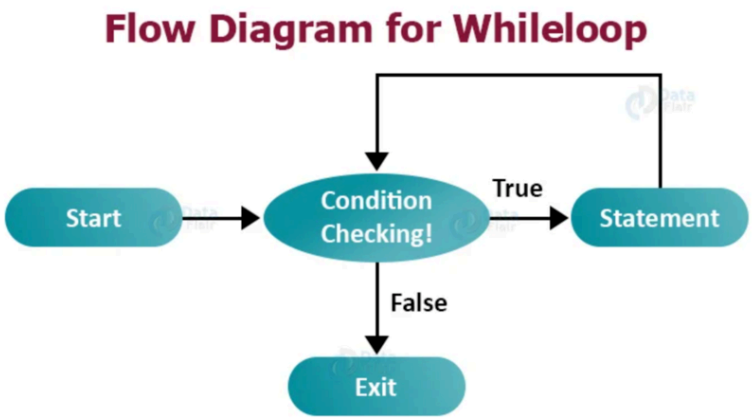
Remember these points when using conditional statements in Java:

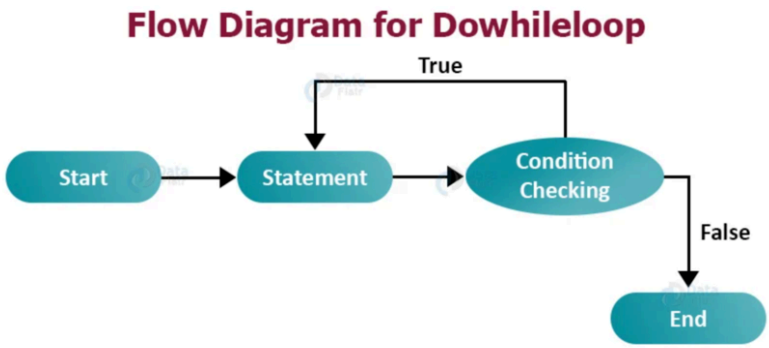
- Conditions should be boolean expressions (**true** or **false**).
- Use curly braces {} to define the scope of the code blocks.
- **else if** and **else** parts are optional.

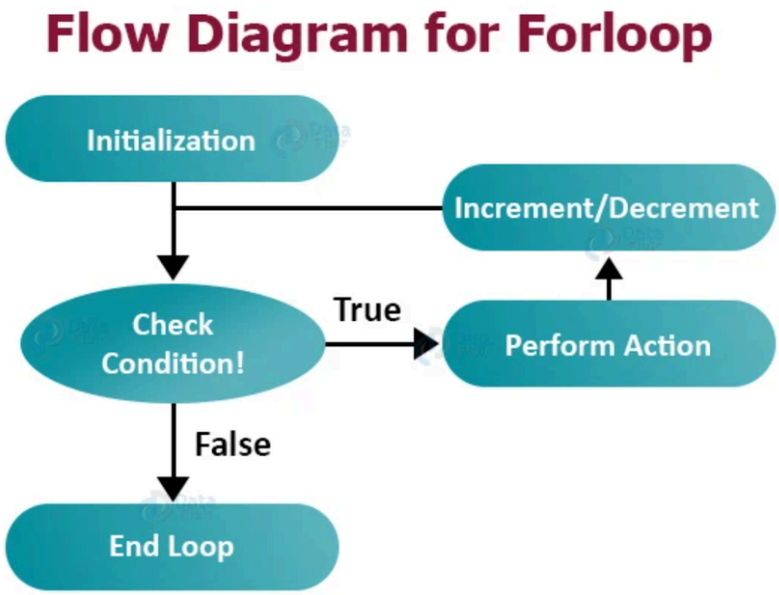
LoopsConcepts_For_While_DoWhile

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 might crash.**

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:- 2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100
	
	// a to z with ASCII
	for (**char** ch = 'a'; ch <= 'z'; ch++) {
	System.***out***.println(ch + " = " + (**byte**) 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
Input number: -11 -- negative 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

7. What will be the output of this program:

```
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".

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
if(e.equals("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. **There are 2 types of array:** 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};

Time Complexity:

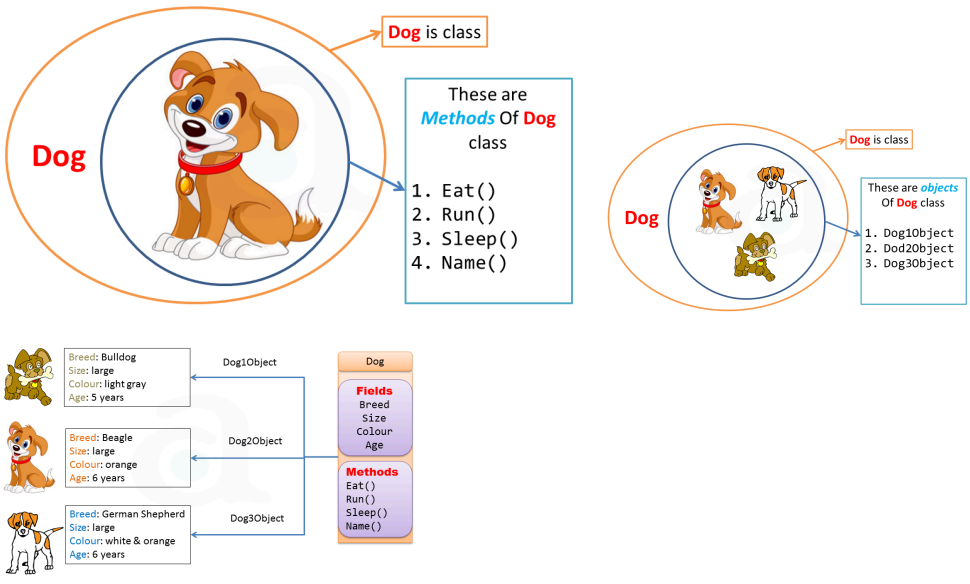
Example	Description	Time Complexity
1	Printing the value of **i**	O(1)
2	Printing numbers from 1 to 10 using a **for** loop	O(n)
3	Printing numbers from 1 to 10 using a **while** loop	O(n)
4	Printing numbers from 1 to 10 and breaking at 5	O(n)
5	Printing pairs of numbers from 1 to 5 using nested loops	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)
8	Halving the input size (**n**) in each iteration - Binary Search	O(log n)

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.

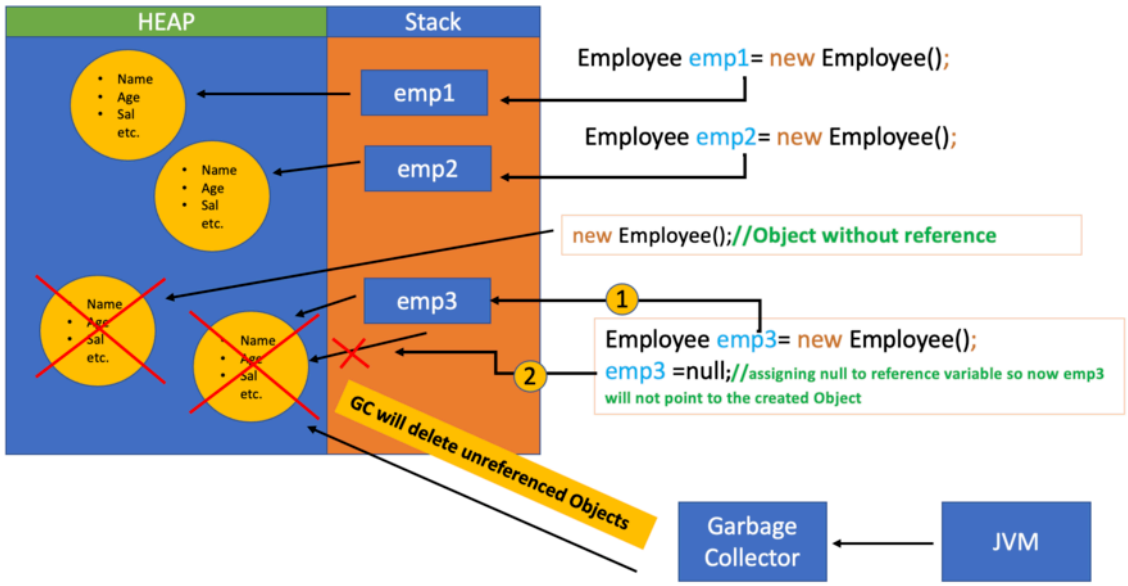
  i. By reference variable
  ii. By method
  iii. By constructor

--- *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.

--- 	
Null Reference Object
	--- 	
	• **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.
	◦ 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.

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.

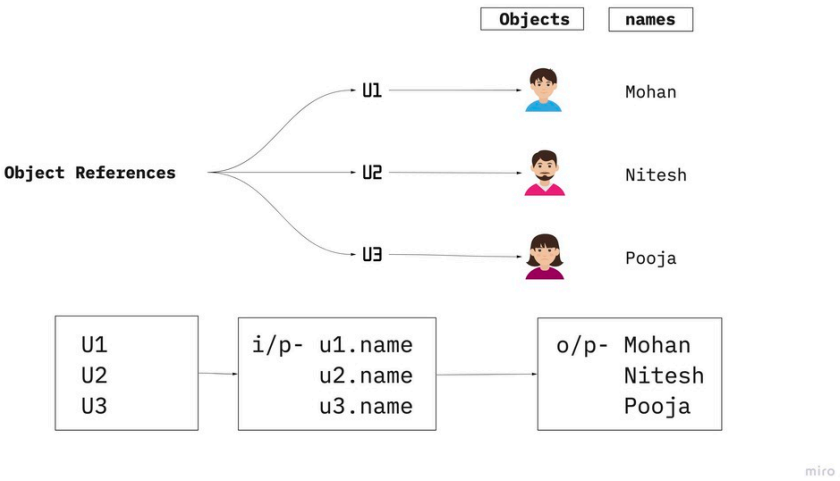
Memory management Java	 • Object is always created in Heap memory. • Reference variable is created in Stack memory.

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.***

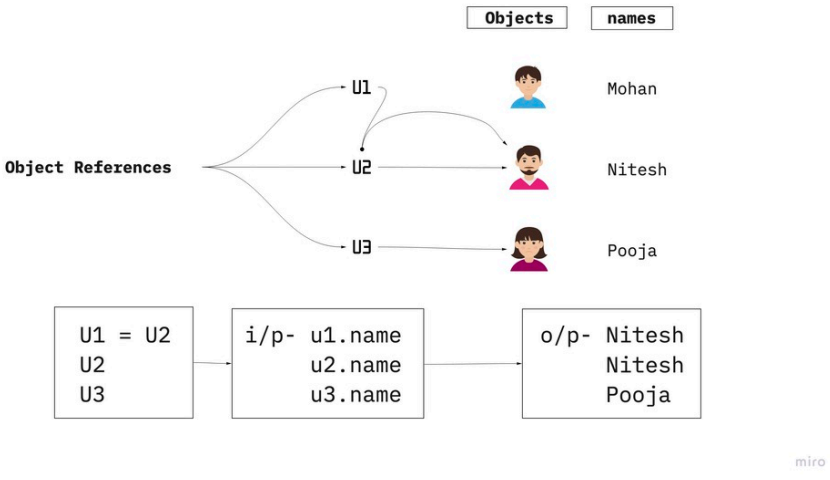
NullPointerException	• ObjectReference e3 pointing to **new Employee()** object
	`Employee e3= new Employee();`
	`e3.name="peter";`
	`e3.age=24;`
	`//System.out.println(e3.name); ------o/p= Peter`
	• Now ObjectReference e3 pointing to **null**
	`e3=null;`
	`System.out.println(e3.name);`
	`//System.out.println(e3.name); ------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.

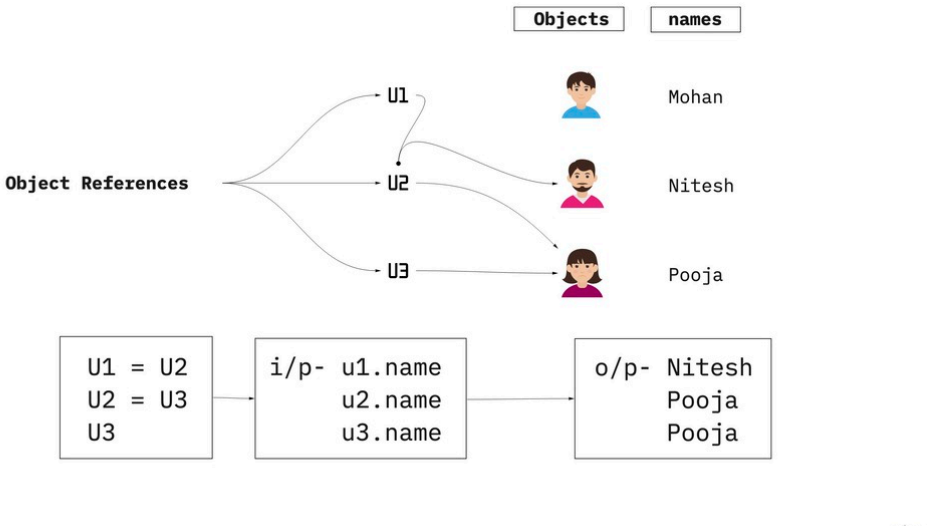
Object References:

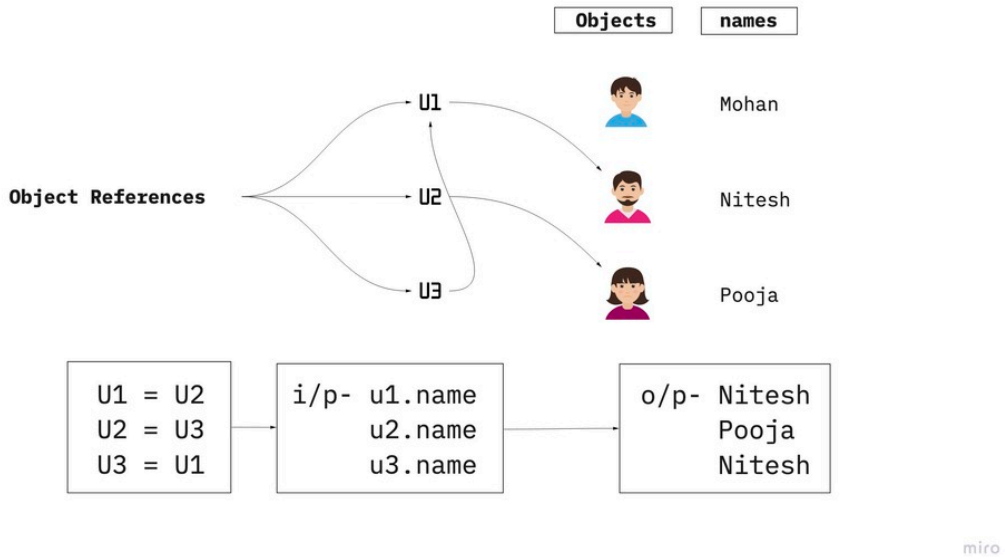
Object References U1,U2,U3 pointing to their respective objects.



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.

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.

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

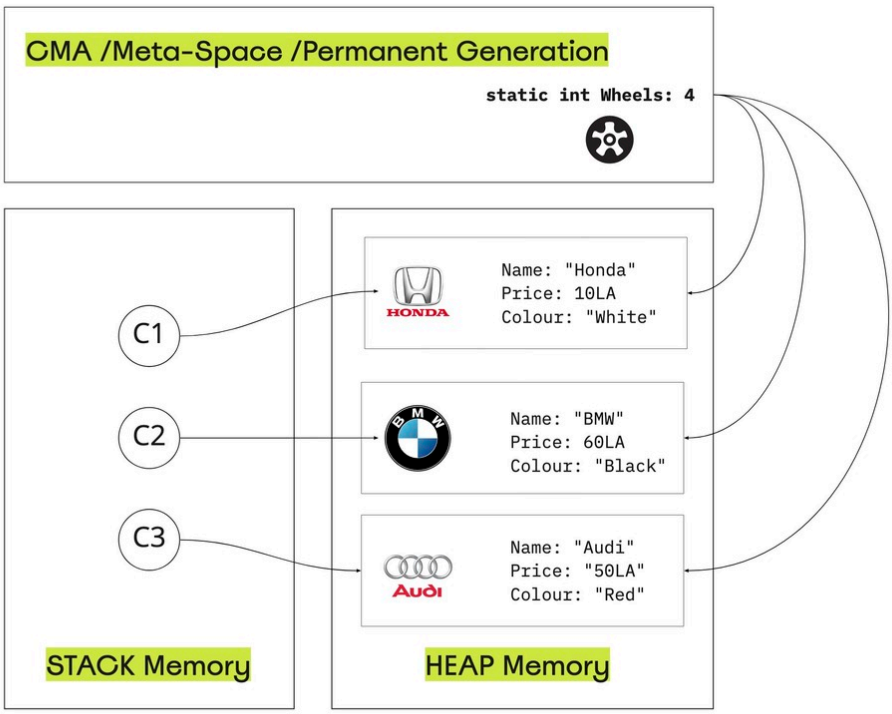
Java Memory : Heap & Stack

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.***



miro

- ***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.

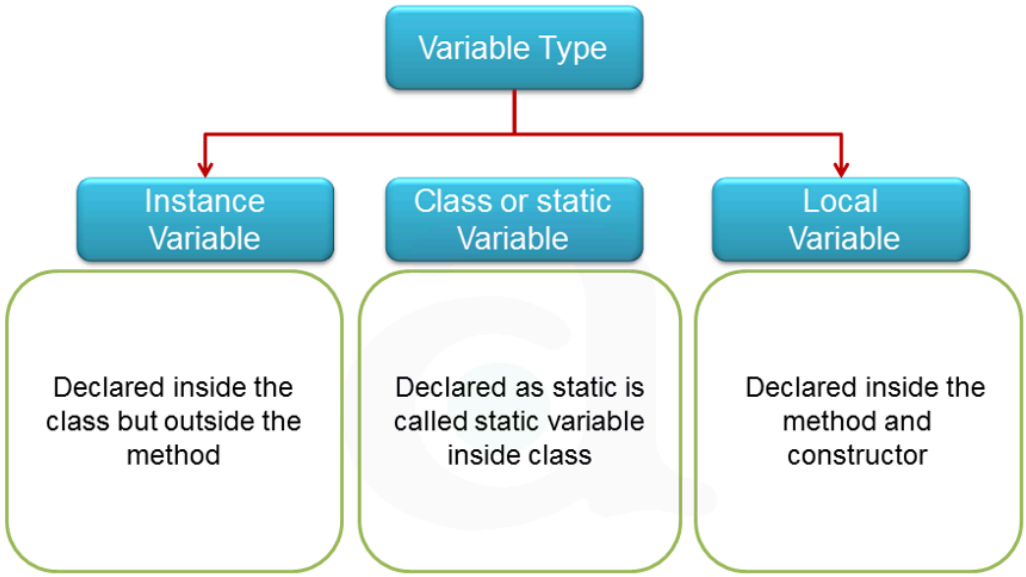
final keyword for a variable.	• If we initialize a variable with the final keyword, then we cannot modify its value. ◦ **ex:- static final int wheels = 4;** ◦ Now wheels is constant ,you can't change the value of wheels to any other value. ◦ Naming convention for final variables - Use capital letters. 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, eventually resulting in a **StackOverflowError**.
Execution flow:

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.

JavaSession

Topic: Constructors_ThisKeyword

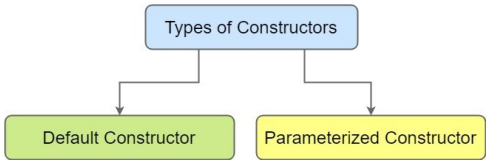
What is constructor?	• A constructor is a block of codes which is used to initialize the object. • It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. • Every time an object is created using the new() keyword, at 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.