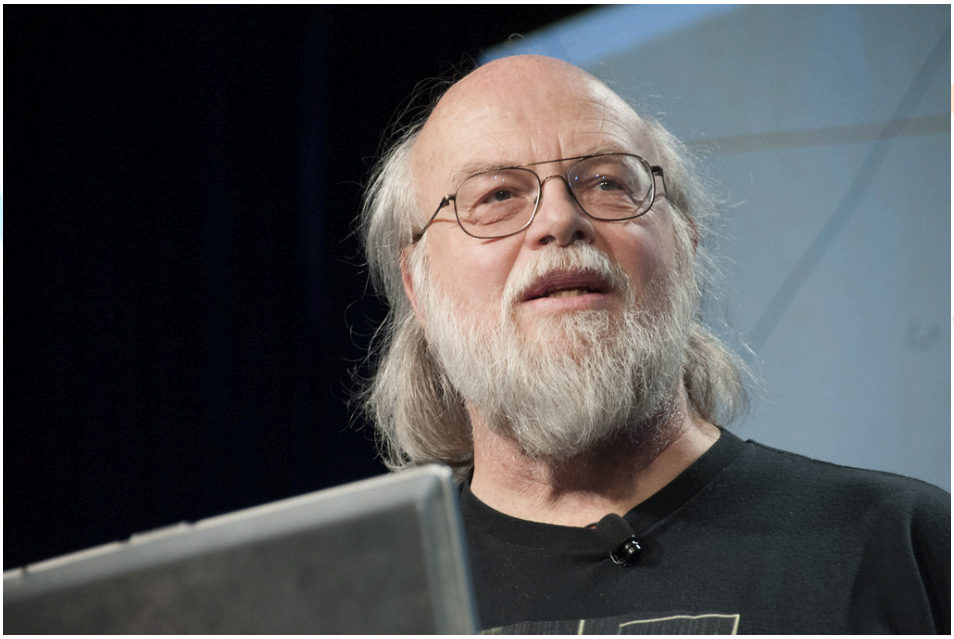
Java

"Empower Your Code: Mastering Java for Next-Level Development"

JAVA

Java is a high-level, object-oriented programming language that was developed by Sun Microsystems (acquired by Oracle Corporation in 2010).



James Gosling "Father of Java"

Here's a brief history of Java:

1. Origins (1991-1995):

- Java's story begins in 1991 when James Gosling, Mike Sheridan, and Patrick Naughton started a

project called ***"Green"*** at Sun Microsystems. The goal was to develop a programming language for consumer electronics.

- The team initially worked with C++, but they found it lacking for the kind of distributed and embedded systems they were targeting.

2. Java Development (1995):

- In 1995, Java was officially released to the public as Java 1.0.
- One of the key features of Java was its "Write Once, Run Anywhere" (WORA) principle, which means that once a program is written, it can run on any device that supports Java without modification.

3. Java 2 and Enterprise Edition (J2EE) (Late 1990s):

- Java 2 (J2SE) was released in 1998, introducing significant improvements and new features.
- The Enterprise Edition (J2EE) was introduced to address the needs of enterprise-level computing, providing a set of specifications for building large-scale, distributed, and multi-tiered enterprise applications.

4. Open Sourcing (2006):

- In 2006, Sun Microsystems released the Java Development Kit (JDK) under the GNU General Public License (GPL), making Java an open-source language.

5. Acquisition by Oracle (2010):

- Oracle Corporation acquired Sun Microsystems in 2010, including the rights to Java. This acquisition has since influenced the development and licensing of Java.

6. Java SE 7 and 8 (2011-2014):

- Java SE 7 was released in 2011, introducing new features such as the try-with-resources statement and the diamond operator for simplifying generic declarations.
- Java SE 8, released in 2014, was a major milestone with the introduction of lambda expressions, the Stream API for working with collections, and the java.time package for modern date and time handling.

7.Java SE 9 and Project Jigsaw (2017):

- Java SE 9, released in 2017, brought modularity to the platform with Project Jigsaw. It aimed to improve scalability, maintainability, and performance by introducing a module system.

8. Java SE 10 and Subsequent Versions

- Oracle transitioned to a time-driven release model, with a new version of Java released every six months. This change started with Java SE 10 in 2018.
- The rapid release cycle allows developers to access new features and improvements more frequently.Java remains one of the most widely used programming languages in the world, powering a vast array of applications, from mobile apps to large-scale enterprise systems. Its portability, robustness,

and versatility contribute to its enduring popularity in the software development community.

Java Flavors:

There are different "flavors" or editions of Java, each tailored for specific application domains:

Java Standard Edition (SE):

The standard edition is the core Java platform used for developing desktop applications and standalone command-line programs. It includes the essential libraries and components for generalpurpose programming.

Java Enterprise Edition (EE):

The enterprise edition is designed for developing large-scale, distributed enterprise applications. It extends the Java SE with additional features for building robust, scalable, and secure applications in a networked environment.

Java Micro Edition (ME):

The micro edition is a version of Java designed for resource-constrained devices such as mobile phones, embedded systems, and other devices with limited memory and processing power.

Characteristics of Java:

Write Once, Run Anywhere (WORA):

Java is known for its portability. Once a Java program is written, it can run on any device with a compatible Java Virtual Machine (JVM) without modification. This is achieved through the use of bytecode, an intermediate representation of the code that is executed by the JVM.

Object-Oriented:

Java is a purely object-oriented programming language. Everything in Java is treated as an object, and the language supports principles like encapsulation, inheritance, and polymorphism.

Platform Independence:

Java programs are compiled into bytecode, which can be executed on any device with a Java Virtual Machine (JVM). This enables Java applications to run on various platforms without modification.

Robust and Secure:

Java incorporates features such as automatic memory management (garbage collection) to enhance robustness and prevent memory-related errors. The language also includes built-in security features to protect against various threats, such as viruses and unauthorized access.

Multi-threading:

Java supports multithreading, allowing concurrent execution of multiple threads within a program. This enables developers to create applications that can efficiently handle multiple tasks simultaneously.

Distributed Computing:

Java has built-in support for distributed computing, making it suitable for developing networked and distributed applications. Java Remote Method

Invocation (RMI) is one of the mechanisms used for this purpose.

Dynamic:

Java is a dynamically extensible language, allowing developers to adapt and extend existing classes easily. This dynamic nature contributes to the flexibility and maintainability of Java programs.

Rich API:

Java comes with a vast set of libraries (APIs) that provide pre-built functionality for tasks such as networking, file I/O, database access, user interface development, and more. This extensive set of libraries simplifies development and saves time for programmers.

JVM Architecture :

Bytecode :

Bytecode is an intermediate representation of a program that is used by the Java Virtual Machine (JVM) during the execution of Java programs. Instead of directly executing machine code, which is specific to a particular hardware and operating system, Java source code is compiled into bytecode. This bytecode is platform-independent and can be executed on any device that has a compatible JVM.

Fundamentals of Java Programming :

Java is a versatile and widely used programming language that follows the principles of simplicity, portability, and

object-oriented programming. Here are some fundamental concepts of Java
programming:

1. Object-Oriented Programming (OOP):

In java we define everything in form of classes and objects. The four main principles of OOP are encapsulation, inheritance, polymorphism, and abstraction.

2. Variables and Data Types :

In java we need to store the data in a place. Hence we get the option called as variables, which will help us to provide the storage for the data. The data types will helps the system to under what type of data to be stored.

3. Control Flow Statements:

- Java supports various control flow statements, including if-else, switch, while, do-while, and for loops. These statements allow you to control the flow of program execution based on conditions and loops.

4. Methods :

Methods are the reusable block of code, which can be called anywhere in the program.

5. Classes and Objects :

Java is a class-based language, and most of the code is written within classes. A class is a blueprint for creating objects, and objects are instances of classes. Each class can have attributes (fields) and methods.

6. Inheritance:

Inheritance allows a class to inherit the properties and behaviors of another class. It promotes code reuse and the creation of a hierarchy of classes.

7. Polymorphism:

- Polymorphism allows objects to be treated as instances of their parent class, providing flexibility and extensibility in the code.

8. Encapsulation :

Encapsulation is the bundling of data (attributes) and methods that operate on the data within a single unit called a class. It protects the internal state of an object and restricts access to certain components.

9. Exception Handling:

- Java provides a robust exception-handling mechanism to deal with runtime errors. The `try`, `catch`, `finally`, and `throw` keywords are used to manage exceptions.

10. Arrays and Collections:

- Java supports arrays, which are fixed-size collections of elements, as well as more advanced collection classes like ArrayList and HashMap from the Java Collections Framework.

Variables and Data Types:

Variables :

Variables are used to store the values for the time being. To write a variable we have to follow some set of rules.

Variable Rules :

1. Variables shouldn’t have to start with numbers.

2. Variable Names shouldn't have to be keywords.
3. We can only initialize a variable single time.
4. Don't use special symbols to create a variables apart from underscore('_'), Dollar ('$').

Example :

DataTypes :

Datatypes are used to define that what type of data are storing inside a variable. In java we have two types of datatypes.

1. Primitive Datatypes

Type	Bit/Bytes	Range
boolean	1 bit	True or false
char	16 bit/ 2 bytes	0 to 65535
byte	8 bit/ 1 byte	-128 to 127
short	16 bit/ 2bytes	-32768 to 32767
int	32 bits/4 bytes	-2147483648 to 2147483647
long	64 bits/ 8 bytes	Huge To huge
float	32 bits/ 4 bytes	varies
double	64 bits /8 bytes	varies

2. Non Primitive Datatypes

1.Primitive Datatypes :

byte:

o 8-bit signed integer.
o Range: -128 to 127.

short:

o 16-bit signed integer.

o Range: -32,768 to 32,767.

int:

o 32-bit signed integer.

o Range: -2^31 to 2^31 - 1.

long:

o 64-bit signed integer.

o Range: -2^63 to 2^63 - 1.

float:

o 32-bit floating-point. IEEE 754 standard.

double:

o 64-bit floating-point.

o IEEE 754 standard.

char:

o 16-bit Unicode character.

o Represents a single character, e.g., 'A', '2', '$'.

boolean:

o Represents true or false values.

o Used for logical comparisons.

2.Non Primitive Datatypes/ Reference Data Types:

Arrays:

o Collections of elements of the same data type.

o Arrays can be one-dimensional, multi-dimensional, or jagged.

Class Types :

o Objects created from classes.

o Instances of user-defined types.

Interface Types:

o Similar to class types but defined by interfaces.
o Instances of interfaces.

Enumeration Types (Enums):

o A special data type that consists of a set of named constants.
o enum Days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY }

Primitive Wrapper Classes:

o Classes that encapsulate primitive data types.

String:

o Represents sequences of characters.
o Immutable in Java.

Operators :

In Java, operators are special symbols that perform operations on operands. Here is a list
of operators in Java:

1. Assignment Operators
2. Arithmetic Operators.
3. Relational Operators.
4. Logical Operators.
5. Increment/ Decrement Operators
6. Bitwise Operators
7. Conditional Operators

a. Assignment Operators :

Assignment operators are used to assign a value to a variable. The assignment operator is represented by equal to '=' .

Syntax :

Datatype variable assignment operator value;

Ex :

b. Arithmetic Operators :

Used to perform the arithmetic operations between two or more operands.

i. Java Follows the BODMAS rule to calculate.

ii. The below are the arithmetic operators.

1. *+ (Addition)*
2. *- (Subtraction)*
3. ** (Multiplication)*
4. */ (Division)*
5. *% (Modulus)*

Addition :

In addition we will add the values using the + sign.

Ex :

Subtraction :

In subtraction we will subtract the values using the – sign.

Ex :

Multiplication :

In multiplication we will multiply two values using the '*' sign.

Ex :

Division :

In Division we will divide the two values using the / sign.

Ex :

Modulo Division :

In modulo Division we will use to get the reminder by diving the two values using the % sign.

Ex :

c. Relation Operator :

Relational operators are used to check the relations between two or more values. It will only return either true or false. The relational operators are mentioned below :

1. *Equal to (==)*
2. *Greater than (>)*
3. *Less than (<)*
4. *Not equal to (!=)*
5. *Greater than equal to (>=)*
6. *Less than equal to (<=)*

a. Equal to :

Used to compare whether the two values are equal or not. Symbol for equal to (==).

b. Greater than :

Used to compare whether the two values are greater than or not. Symbol for greater than (>).

c. Less than :

Used to compare whether the two values are less than or not. Symbol for less than (<).

d. Not equal to :

Used to compare whether the two values are not equal or not. Symbol for not equal to (!=).

e. Greater than equal to :

Used to compare whether the two values are greater than equal or not. Symbol for greater than or equal to (>=).

f. Less than equal to :

Used to compare whether the two values are less than equal or not. Symbol for less than or equal to (<=).

Output :

d. Logical Operators :

logical operators are used to perform logical operations on boolean values. There are three main logical operators in Java:

1. && (logical AND),
2. || (logical OR),
3. ! (logical NOT).

1. AND (&&) :

a. It will generate true or false only when the value meets the condition. It is like multiplying the 0 with 1.

Input1	Input 2	Output
False	False	False
True	False	False

False	True	False
True	True	True

Example :

Output :

2. OR (||) :

a. It will generate true or false only when the values meets the condition. It is like adding the 0 with 1.

Input1	Input 2	Output
False	False	False
True	False	True
False	True	True
True	True	True

Example :

Output :

3. Not (!) :

a. It will just inverse the original value. If it is true changes to false = > if it is false changes to true.

Input	Ouput
True	False
False	True

Example :

Output :

e. Increment/ Decrement Operator :

Increment and decrement operators are used to increase or decrease the value of a variable by 1. These operators come in two forms:

1. increment
2. decrement.

Increment (++):

In increment we have two types which are pre and post.

1. Pre – increment : Will increase value before storing into the variable. We use the ++ symbol before the variable for the pre increment.

Syntax :

++ variable_Name;

2. Post – increment : Will increase value after storing into the variable. We use
the increment symbol behind the variable for post increment.

Syntax :

Variable _Name ++;

Decrement (--) :

In decrement we have two types which are pre and post and will do the same thing as in increment only difference is it will decrease the value by 1.

1. Pre – decrement : Will decrease value before storing into the variable. We use the -- symbol before the variable for the pre decrement.

Syntax :

--Variable_name;

2. Post- Decrement : Will decrease value after storing into the variable. We use the -- symbol behind the variable for the pre decrement.

Syntax :

Variable_name -- ;

Output :

f. Bitwise Operators :

In Java, bitwise operators are used to perform operations on individual bits of integers.

Here are the bitwise operators in Java:

1. AND Operator (&):

- The AND operator is represented by the symbol `&`.

- It performs a bitwise AND operation between corresponding bits of two integers.

- Example: `result = a & b;

Input1	Input 2	Output
0	0	0
1	0	0
0	1	0
1	1	1

Example :

Output :

2. OR Operator (|):

- The OR operator is represented by the symbol `|`.
- It performs a bitwise OR operation between corresponding bits of two integers.

- Example: `result = a | b;`

Input1	Input 2	Output
0	0	0

1	0	1
0	1	1
1	1	1

Example :

Output :

3. XOR Operator (^):

- The XOR operator is represented by the symbol `^`.
- It performs a bitwise XOR (exclusive OR) operation between corresponding bits of two integers.

- Example: `result = a ^ b;`

Input1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

Example :

Output :

4. NOT Operator (~):

- The NOT operator is represented by the symbol `~`.
- It performs a bitwise NOT operation, inverting each bit of the operand.

- Example: `result = ~a;

For each bit in the binary representation of the operand:

- If the bit is 0, it becomes 1 after the NOT operation.
- If the bit is 1, it becomes 0 after the NOT operation.
    - If the binary number starts with 0 then it is a +ve number.
    - If the binary number starts with 1 then it is a –ve number.

Example :

Output :

5. Left Shift Operator (<<):

- The left shift operator is represented by the symbol `<<`.
- It shifts the bits of the left operand to the left by a specified number of positions.

Example :

Output :

6. Right Shift Operator (>>):

- The right shift operator is represented by the symbol `>>`.
- It shifts the bits of the left operand to the right by a specified number of positions. The sign bit is used for signed integers.

Example :

Output :

7. Zero-fill Right Shift Operator (>>>):

- The zero-fill right shift operator is represented by the symbol `>>>`.
- It shifts the bits of the left operand to the right by a specified number of positions. It always fills the vacant positions with zero.
- Example: `result = a >>> 3;

g. Conditional Operator :

Conditional operators, also known as ternary operators, are used in programming languages to perform a conditional (if-else) operation in a concise and compact manner. The most common conditional operator is the ternary operator, which is denoted by the "?" symbol. It is often referred to as the "conditional operator" because it evaluates a condition and returns one of two values based on whether the condition is true or false.

Syntax :

condition ? expression_if_true : expression_if_false

Here's how it works:

- The condition is evaluated first.
- If the condition is true, the expression following the "?" is executed.
- If the condition is false, the expression following the ":" is executed.

Ex :

Selection/ Conditional Statement :

In Java, conditional statements are used to control the flow of a program based on certain conditions. The primary conditional statements in Java are the

1. if
2. else if , and
3. else statements.

1. if statement:

The `if` statement is used to execute a block of code only if a specified condition is true.

Syntax :

```
If(true){
        If – block
}
```

Example :

2. if-else statement:

The `if-else` statement is used to execute one block of code if the condition is true and another block if the condition is false.

Syntax :

```
if(true){
        if block statements
}
Else{
 Else block statement
}
```

Example :

3. if-else if-else statement :

The `if-else if-else` statement allows you to check multiple conditions in sequence.

Syntax :

```
if(true){
        if block statements
}
Else if (true){
        Else if block statement
}
Else{
       Else block statements
}
```

Example :

4. switch statement:

The `switch` statement is used to select one of many code blocks to be executed.

Syntax :

```
Switch(value) :
    Case value1 :
        Statement
        Break;
    Case value2 :
        Statement
        Break;
    Case valuen :
        Statement
        Break;
    Default :
        Statement
        Break;
```


Example :

Loops in Java :

In java loops are used to repeat a block of statements n-number of times. We are having
the 3 basic types of loops in java.

1. For
2. While
3. Do-while

1. For Loop :

a. It is used when we know the range of n, then it will make the work a lot easy.

i. Syntax :

```
for( initialization ; condition ; increment/decrement){
        // block of code
}
```

Ex :

2. While Loop :

a. It is used when we don't the range of n, can use similar as for loop.

i. Syntax :

```
Initialization;
 while( condition ){
        // block of while loop
        Increment/decrement;
 }
```

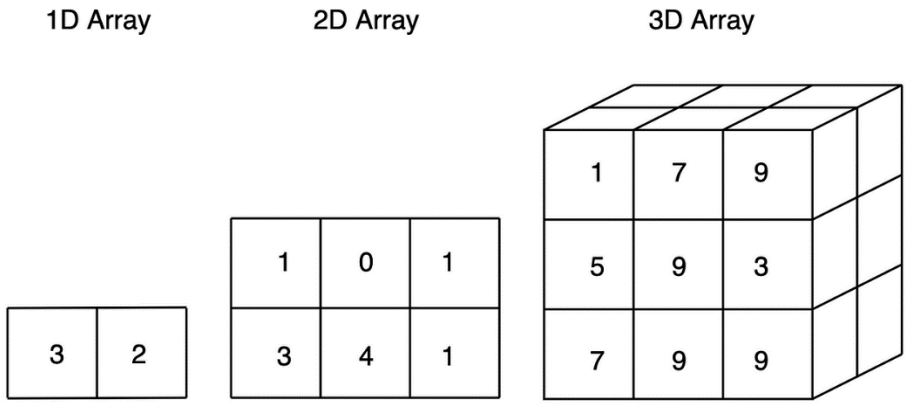
Ex :

3. Do –while Loop :

a. It is used to print the entire block of even when the condition is false, We can also use this similar to the other loops

i. Syntax :

```
Initialization;
do{
        // block of do while
        Increment/ decrement;
}while( condition );
```



Ex :

Note : there is also another loop which we didn't discussed which is foreach loop. We will cover it in arrays.

Arrays in Java

Arrays in Java are used to store multiple values of the same data type under a single variable name. They provide a convenient way to manage collections of elements. There are two types of arrays in java.

1. *Single Dimensional Arrays*
2. *Multi Dimensional Arrays*

1. Declaration and Initialization:

Syntax for declaration:

For 1D array :

Datatype variable[];

```
int a1[];
```

For 2D array :

DataType variable[][];

```
int a1[][];
```

Syntax for static initialization :

For 1D array :

Variable = new int[] {val,val2,…,valn};

```
a1 = new int[] {1,2,3,4};
```

For 2D array :

Variable[][] = { {value1,..valuen} …. {value1, valuen} };

```
a1 = new int[][] { {1,2,3,4} };
```

Syntax for Dynamic initialization :

For 1D array :

Variable[] = new int[size of array];

Variable[index] = value;

```
int a3[] = new int[4];

a3[0] = 1;
a3[1] = 2;
```

For 2D array :

Variable[][] = new int[row_size][col_size];

Variable[row_index][col_index] = value;

2. Accessing Array Elements:

Syntax :

For 1D array :

Variable[index];

For 2D array :

Variable[row_index][col_index];

3. Modifying Array Elements:

Syntax :

```
Variable[index] = value;
```

4. Array Length:

The length of an array is obtained using the `length` property:

Syntax :

```
Variable.length;
```

NOTE : it will return the row count for 2D array.

5. Iterating Through an Array:

We can use the loops to iterate the array or else we can also use the foreach loop. the enhanced for loop, also known as the "for-each" loop, provides a concise way to iterate through elements in an array or collections.

Syntax :

For 1D array :

```
for (element_type element : array_name) {
        // code to be executed for each element
}
```

For 2D array :

```
for (datatype[] outerArray : arrayName) {
        for (datatype element : outerArray) {
                // Code to be executed for each element in the 2D
                    array
        }
}
```

Regex in java

These is used to search the patterns in java. We are having a package to handle this regex that lib is java.util.regex which will have ‘matcher’ and ‘pattern’.

Basic Components :

1. **Literals** :
    a. Match the character that was provided.
2. **MetaCharacters** :

a. We use the special characters to match the pattern each character will have it's own meaning. They are mentioned in below

Metacharacter	Name	Matches
d\	Digit	Matches a digit
s\	Whitespace	Matches whitespace
[a-z, A-Z]	A range of letters	Matches any letter in the specified range.
.	Dot	Matches any one character
[…]	Character class	Matches any one character listed
[^…]	Negated character class	Matches any one character not listed
?	Question	One allowed, but it is optional
*	Star	Any number allowed, but all are optional
+	Plus	At least one required; additional are optional
|	Alternation	Matches either expression it separates
^	Caret	Matches the position at the start of the line
$	Dollar	Matches the position at the end of the line
{X,Y}	Specified range	X required, max allowed

Classes :

1. ***Pattern Class*** :
   a. It is used to represent the pattern that we need to match.
2. ***Matcher Class :***
   a. It matches the pattern that we created and using the pattern class.

Syntax :

```
Pattern patternObject = Pattern.compile( "RegularExpression" );
Matcher matchObject = Pattern.matcher("String");
```

Example for Password Matching :

```
public class Application {
public static void main(String[] args) {
        String passwordRegex =
"^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=]).{8,}$";
        String password = "ab@dAf123";
```

```
            Pattern pattern = Pattern.compile(passwordRegex);
            Matcher matcher = pattern.matcher(password);

            if(!matcher.matches()) {
                System.out.println("Invalid password");
            }else {
                System.out.println("Valid password");
            }
        }
}
```

Email Validation

Regular Expression: ***^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$***

Explanation :

Email Validation Regex:

- `^[a-zA-Z0-9._%+-]+`: Matches the local part of the email.
- `@`: Matches the '@' symbol.
- `[a-zA-Z0-9.-]+`: Matches the domain name.
- `\\.[a-zA-Z]{2,6}$`: Matches the top-level domain.

Password Validation

Requirements:

- At least 8 characters long
- Contains at least one lowercase letter
- Contains at least one uppercase letter
- Contains at least one digit
- Contains at least one special character from the set `@#$%^&+=`

Regular Expression: ***^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=]).{8,}$***

Explanation :

Password Validation Regex:

- `^(?=.*[0-9])`: Asserts that there is at least one digit.
- `(?=.*[a-z])`: Asserts that there is at least one lowercase letter.
- `(?=.*[A-Z])`: Asserts that there is at least one uppercase letter.
- `(?=.*[@#$%^&+=])`: Asserts that there is at least one special character.
- `.{8,}$`: Asserts that the string is at least 8 characters long.

Username Validation (Must Contain Digits)

Requirements:

- Alphanumeric characters
- At least one digit

Regular Expression: ***^(?=.*\d)[a-zA-Z0-9]+$***

Explanation :

Username Validation Regex:

- `^(?=.*\\d)`: Asserts that there is at least one digit.
- `[a-zA-Z0-9]+$`: Matches alphanumeric characters.

Classes and Objects :

Class :

A class is a blueprint or template that defines the characteristics and behaviors of objects. It serves as a way to create objects, which are instances of the class.

Syntax :

```
public class className{

    // Class body goes here
```

}

Objects :

Objects are instance of a class. Objects are created using the new keyword followed by the class constructor (a special method that initializes the object).

Syntax :

ClassName object = new className();

Strings :

Strings are objects that represent sequences of characters. The String class in Java is used to create and manipulate strings. Strings are immutable in java.

String Declaration:

1. ***Method one :***
   String variable = "String" ;

   String variable = new String("String");

2. ***Method two :***
   String variable = scannerObj.next( );

   String variable = scannerObj.nextLine( );

Note : The first method uses a string literal, and the second method uses the **String** class constructor.

String Methods :

We are having some string methods to manipulate the strings.

Certainly! Here is a list of commonly used methods provided by the `String` class in Java:

1. charAt(int index):

Returns the character at the specified index.

Syntax :

```
stringVar.charAt(index);
```

2. codePointAt(int index):

Returns the Unicode code point at the specified index.

Syntax :

```
StringVar.codePointAt(position);
```

3. **codePointBefore(int index):**

Returns the Unicode code point before the specified index.

Syntax :

```
StringVar.codePointBefore(position);
```

4. **codePointCount(int beginIndex, int endIndex):**

Returns the number of Unicode code points between the specified indices.

5. **compareTo(String anotherString):**

Compares two strings lexicographically.

6. **compareToIgnoreCase(String str):**

Compares two strings lexicographically, ignoring case differences.

7. **concat(String str):**

Concatenates the specified string to the end of the current string.

8. **contains(CharSequence sequence):**

Returns true if the string contains the specified sequence of characters.

9. **contentEquals(CharSequence cs):**

Compares the content of the string to the specified `CharSequence`.

10. **endsWith(String suffix):

Checks if the string ends with the specified suffix.

11. **equals(Object obj):

Compares the content of two strings for equality.

12. **equalsIgnoreCase(String anotherString):

Compares two strings for equality, ignoring case.

13. **format(String format, Object... args):

Returns a formatted string using the specified format string and arguments.

14. **getBytes():

Encodes the string into a sequence of bytes using the platform's default charset.

15. **indexOf(ch):

Returns the index of the first occurrence of the specified character.

16. **indexOf(String str):

Returns the index of the first occurrence of the specified substring.

17. **isEmpty():

Returns true if the string is empty.

18. **lastIndexOf(int ch):

Returns the index of the last occurrence of the specified character.

19. **lastIndexOf(String str):

Returns the index of the last occurrence of the specified substring.

20. **length():

Returns the length of the string.

21. **replace(char oldChar, char newChar):

Returns a new string resulting from replacing occurrences of `oldChar` with `newChar`.

22. **replaceAll(String regex, String replacement):

Replaces each substring of this string that matches the given regular expression with the given replacement.

23. **split(String regex):

Splits this string around matches of the given regular expression.

24. **startsWith(String prefix):

Checks if the string starts with the specified prefix.

25. **substring(int beginIndex):**

Returns a substring starting from the specified index.

26. **substring(int beginIndex, int endIndex):**

Returns a substring between the specified indices.

27. **toCharArray():**

Converts the string to a character array.

28. **toLowerCase():**

Converts all characters in the string to lowercase.

29. **toUpperCase():**

Converts all characters in the string to uppercase.

30. **trim():**

Removes leading and trailing whitespaces.

31. **valueOf(value)**

Returns the string representation of the datatype arguments. It simply converts from any datatype to String datatype.

String Builder and String Buffer :

Here string Buffer is a mutable, where we can change the content in the Strings. So that we don't need to create an object every time when we want to change the content in a string. It is having some built in methods. They are

1. append()
2. insert()
3. delete()
4. reverse()
5. capacity() - Used to check the size of the string buffer

The only difference between String buffer and String Builder is String builder is faster than String buffer.

NOTE : To check the time we can use the CurrentTimeMillis() to get the current time.

Example :

```
Long time = System.CurrentTimeMillis();
```

Class :

Class is a blue print of an object. In java we write the code inside the classes itself. As we know we use the class keyword inorder to create a class.

Syntax :

```
class className{
    //class code/ objects/ methods/ variables
}
```

Ex :

```
Class HouseBluePrint{
        String houseName;
        Int dateOfInaruguation;
}
```

Object :

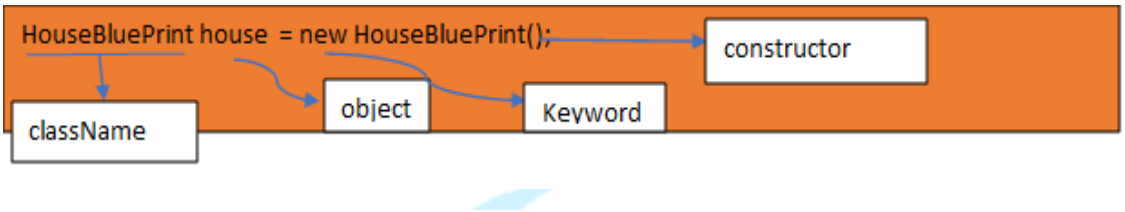
It is instance of class. Object will constructed based on class.

Syntax :

ClassName objectName = new ClassName();

Example :

HouseBluePrint house = new HouseBluePrint();



Constructor:

Constructor is used to create/ allocate a memory for the object. To create a constructor we have to follow the below rules.

Rules :

1. *We need to use the same name as class name.*
2. *We shouldn't mention the return type.*
3. *It will accept the arguments but will not have return type.*

Syntax :

```
Public className(){
    //body of constructor
}
```

NOTE : If you wrote a return type then it will become a method not a constructor. It will only return a warning not error.

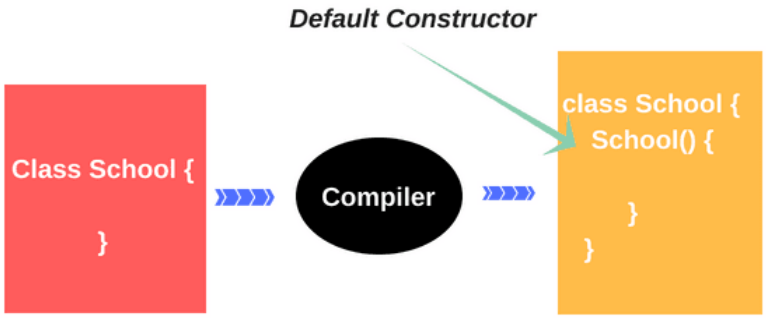
Note : By default jvm will provide the constructor while the object is created.

There are two types of constructors :

1. Default constructor
2. User defined Constructor
    1. *Parameterized Constructor*
    2. *Copy Constructor*
    3. *Constructor Overloading*

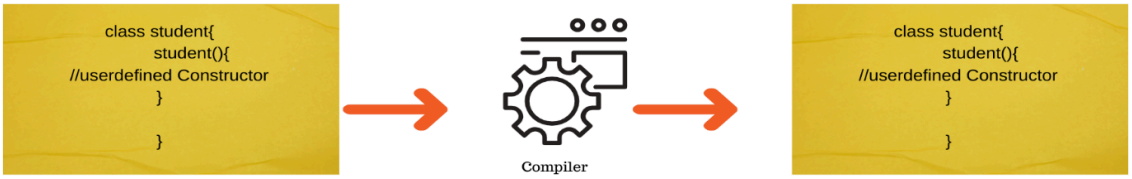
1. Default Constructor :

If there is no constructor inside a class then the JVM will create a constructor automatically during run time.



2. User Defined Constructor :

Here we will write a constructor then the default constructor will get override by the constructer that the user provided.



2.1 . Parameterized Constructor :

A parameterized constructor is a constructor with parameters. It allows you to initialize the instance variables with values provided at the time of object creation.

Example :

```
public class MyClass {

    private int value;

// Parameterized constructor
```

```
    public MyClass(int value) {
    this.value = value;
    }
}
```

2.2 Copy Constructor:

A copy constructor creates a new object by copying the values from an existing object. It helps in creating a new object with the same state as the original object.

Example :

```
    public class MyClass {
    private int value;
    // Copy constructor
    public MyClass(MyClass original) {
    this.value = original.value;
    }
    }
```

2.3 Constructor Overloading:

Constructor overloading is a concept where a class has multiple constructors with different parameter lists. The appropriate constructor is chosen based on the arguments provided during object creation.

Example :

```
public class MyClass {
private int value;
// Parameterized constructor
public MyClass(int value) {
this.value = value;
}
// Default constructor
public MyClass() {
// Default initialization or additional logic
}
}
```

This keyword :

In Java, the `this` keyword is a reference variable that refers to the current object. It is used within an instance method or a constructor to

refer to the current instance of the class. The primary purposes of the `this` keyword are:

1. **To distinguish instance variables from local variables:*

When a local variable in a method or constructor has the same name as an instance variable, the `this` keyword is used to differentiate between the two.

Example :

```
public class MyClass {
private int myVar;
public void setMyVar(int myVar) {
// Use 'this' to refer to the instance variable
this.myVar = myVar;
}
}
```

2. **To invoke the current object's method:*

It can be used to invoke the current object's method, especially when a method parameter has the same name as an instance variable.

Example :

```
public class MyClass {
private int value;
public void setValue(int value) {
// Use 'this' to invoke the method of the current object
this.value = value;
}
}
```

3. **To pass the current object as a parameter to other methods:*

It can be used to pass the current object as a parameter to other methods.

Example :

```
public class MyClass {
private int data;
public void processData() {
// Pass the current object to another method
SomeClass.process(this);
}
}
```

Note ; the `this` keyword is a reference to the current object, and its usage helps in distinguishing between instance variables and local variables, as well as in situations where you need to reference the current object explicitly.

Methods :

Methods are block of code which is used for the reusability of the logic. In java we are using the methods for the code reusability and to reduce the Lines Of Code(LOC).

Method Declaration :

Syntax :

AccessSpecifier returnType methodName(par1,..,parn){

//Block of code for reusability

}

Example :

```
public void Method() {
    //block of code
}
```

Calling a Method :

We use the semi-colon to call method with the help of object.

Syntax :

ObjectName.MethodName(arguments);

NOTE: The no of Arguments will always depends on no of parameters

Example :

```
obj.Method();
```

There are 4 types of methods in java

1. *Without Arguments, without Return Value*
2. *Without Arguments, With Return Values*
3. *With Arguments, Without Return Values*
4. *With Arguments, With Return Values*

1. Without Arguments, without Return Value

Example :

```java
public class MethodWOAWR {
    public void greet() {
        System.out.println("Hello");
    }
    public static void main(String[] args) {
        MethodWOAWR object = new MethodWOAWR();
        object.greet();
    }

}
```

2. Without Arguments, With Return Values

```java
public class MethodWOAWR {
    public String greet() {
        String Message = "Hello";
        return Message;
    }
    public static void main(String[] args) {
        MethodWOAWR object = new MethodWOAWR();
        String Message = object.greet();

        System.out.println(Message);
    }

}
```

3. With Arguments and Without Return Values :

```
public class MethodWOAWR {
    public void greet(String Name) {
        System.out.println("Hello " + Name);
    }
    public static void main(String[] args) {
        MethodWOAWR object = new MethodWOAWR();

        object.greet("Max"); //output : Hello Max
    }

}
```

4. With Arguments and With Return Type :

```
public class MethodWOAWR {
    public String greet(String Name) {
        String Message = "Hello " + Name;
         return Message;
    }
    public static void main(String[] args) {
        MethodWOAWR object = new MethodWOAWR();

        String Sen = object.greet("Max");
        System.out.println(Sen); //output : Hello Max
    }

}
```

Inheritance:

Inheritance is when we are inheriting the properties of a class with another class. In here we will have to use the extends keyword inorder to implement the inheritance.

1. Parent class/ super class

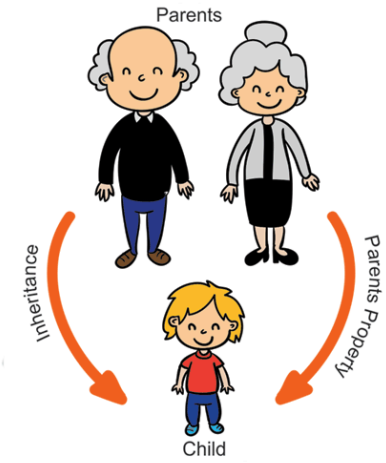
2. Child class/ sub class

1. Super Class :

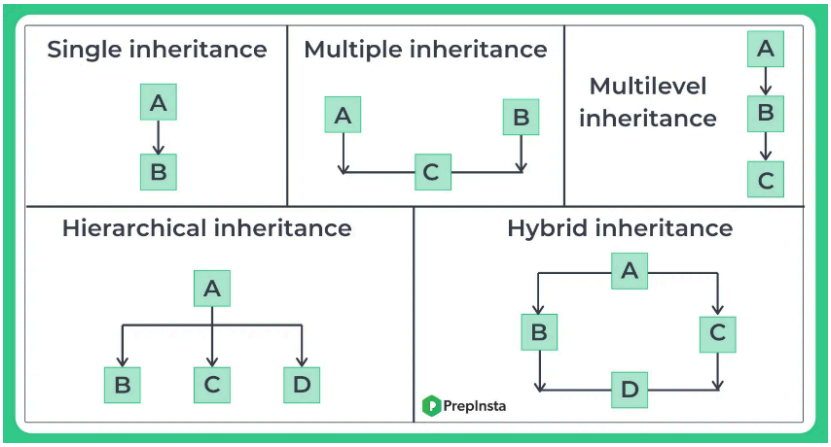
This is the class whose features are inherited. It's also called the base class or parent class.

2. Sub Class :

This is the class that inherits the features from the superclass. It's also called the derived class or child class.



Types of inheritance :



1. ***Single inheritance :***

In single level inheritance we only have one super and one child class where the child class will inherit the methods and variables of the parent class.

Syntax :

```
class parent{
    //Methods and variables of parent/ super class
}
Class child extends parent{
    //Methods and variables of child class/sub class
}
```

Example :

```java
class parent{
    int a;
    void setA(int a) {
        this.a = a;
    }
    void getA() {
        System.out.println(a);
    }
}

public class singleLevel extends parent{

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        singleLevel o = new singleLevel();
        o.setA(10);
        o.getA(); //output : 10


    }
```

2. ***Multilevel Inheritance :***

In multilevel inheritance we will extend one extended class with another class like a ladder.

Syntax :

```
class parent{
    //parent methods and objects
}
Class child1 extends parent{
    //child1 methods and objects
}
```

Class child2 extends child1{

//child2 methods and objects

}

Example :

```
class parent{
    void Method1() {
        System.out.println("Parent Method");
    }
}

class child1 extends parent{
    void Method2() {
        System.out.println("Child1 Method");
    }
}

class child2 extends child1{
    void Method3() {
        System.out.println("Child2 Method");
    }

}

public class singleLevel extends parent{

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        child2 c2 = new child2();
        c2.Method1(); //Parent Method
        c2.Method2(); //Child1 Method
        c2.Method3();//Child2 Method

    }

}
```

3. ***Hierarchical Inheritance :***

Hierarchical Inheritance is where a sub classes will use the same parent class.

Syntax :

```
class parent {
        //parent methods and object
        }
        Class child1 extends parent{
        //child1 methods and object
        }
        Class child2 extends parent{
        //child2 methods and object
        }
```

Example :

```java
class parent{
    void Method1() {
        System.out.println("Parent Method");
    }
}

class child1 extends parent{
    void Method2() {
        System.out.println("Child1 Method");
    }
}

class child2 extends parent{
    void Method3() {
        System.out.println("Child2 Method");
    }

}

public class singleLevel extends parent{

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        child1 c1 = new child1();
        child2 c2 = new child2();
        c2.Method1(); //Parent Method
        c1.Method1(); //parent Method
    }

}
```

Note : To implement the remaining inheritance we need to use the interfaces as the java in not able to support the multiple and Hybrid inheritance.

Abstract classes and Methods :

Abstract is a keyword in java which is used for methods and classes.

1. Abstract Class:

- An abstract class is a class that cannot be instantiated on its own; it serves as a blueprint for other classes.
- It can contain both abstract and non-abstract (concrete) methods.
- Abstract will act as normal class as it will contain all the things that contained with in a class. EX : constructor, instance variables, static variables.

Note : If a class has at least one abstract method, it must be declared as abstract.

NOTE : Subclasses of abstract classes must either implement all the abstract methods declared in the superclass or be declared as abstract themselves.

2. Abstract Methods :

- An abstract method is a method declaration without a body; it provides the method's signature but not its implementation.
- Abstract methods are typically meant to be overridden (implemented) by subclasses.
- Abstract methods are declared using the abstract keyword.

NOTE : Abstract methods can only exist within abstract classes or interfaces.

Syntax :

For class :

```
abstract class className{
```

//abstract Methods and body of class

}

For Methods :

Abstract returnType methodName(){

//method body

}

Example :

```
abstract class a{
    abstract void m();
    void display() {
        System.out.println("Displaying...");
    }
}


public class singleLevel extends a{

    //override
    void m() {
        System.out.println("override method of a class");
    }


    public static void main(String[] args) {
        // TODO Auto-generated method stub
        singleLevel obj = new singleLevel();
        obj.display();
        obj.m();
    }
}
```

Final :

In Java, the final keyword is used to define entities that cannot be changed or extended. It can be applied to variables, methods, and classes, each with its own implications:

1. *Final Variable*
2. *Final class*
3. *Final Method*

1. ***Final variables :***

When applied to a variable, it means that the variable's value cannot be changed once it's initialized. For reference types (objects), it means the reference cannot be changed, but the state of the object can still be modified if it's mutable.

Syntax :

Final datatype variableName = value ;

Example :

Final int PI = 3.14;

2. ***Final Methods :***

When applied to a method, it means that the method cannot be overridden by subclasses. This is commonly used to prevent subclasses from changing the behavior of certain methods, especially critical methods in a class's design.

Syntax :

```
final returnType methodName(){
        //Method body
}
```

Example :

```
Class priate{
    Final void priateKing(){
        //priateKing method body
        }
}

    Class super extends priate{
    @override
    void priateKing(){
    //will throw error as it is a final method
        //priateKing method body
        }

}
```

3. ***Final class :***

When applied to a class, it means that the class cannot be subclassed. This is often used when a class's design is considered complete or when inheritance is not desired for security or design reasons.

Syntax :

```
final class Super{
    //class methods and objects
}
```

Example :

```
class sub extends Super{
    //throws error
}
```

Interface :

An interface in Java is a reference type that is similar to a class but contains only abstract methods, default methods, static methods, and constant declarations (variables that are implicitly public, static, and final). It defines a contract for what a class implementing the interface must do, without providing the implementation details.

Note : interfaces will only have the structure not the body.

Note : To inherit an interface we use the implements method and not the extends

In order to create a interface we need to use the interface keyword before class.

Syntax :

```
Interface interfaceName{
    //Methods structure or final and static variables
}
```

In interfaces we can also provide the variables but they must be final or static.

Add-on's :

Interface will not have the memory in heap as they will share in class memory

Example :

```java
interface PriateKing{
    final String IslandName = "Raftle";

    void priateKings();
}

class onePiece implements PriateKing{

    public void priateKings() {
        System.out.println("People who reached "+IslandName+ " will be titled as priateking");
    }
}

public class MainMethod{

    public static void main(String[] args) {
        onePiece Roger = new onePiece();
        Roger.priateKings(); // output : People who reached Raftle will be titled as priateking

    }
}
```

Multiple Inheritance :

In java we can't use the multiple inheritance as there is a limit to the extends keyword. As extends keyword can only extend one class at a time. Hence we use the interfaces in order to achieve the multiple inheritance.

Syntax :

```
interface A{

//static and final methods and variables

}

Interface B{

// static and final methods and variables

}
```

Class child implements A,B{

//override the methods and create a new one

}

Example :

```
interface parent1{
    public void test();
}

interface parent2{
    public void test();
}
public class Multiple implements parent1, parent2{
    @Override
    public void test() {
        // TODO Auto-generated method stub
        System.out.println("Multiple");
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Multiple m = new Multiple();
        m.test(); //Multiple

    }
}
```

Hybrid Inheritance :

We can't perform the hybrid level inheritance directly with the classes as the multiple inheritance in the java is not possible. We can combine the both classes and interface together.

Syntax :

class A{

//methods and objects

```
}
Interface B{
//abstract methods and objects
}
Interface C{
//abstract methods and objects
}
Class B extends A implements B{
//body
}
```

Example :

```java
class computer{
    void run() {
        System.out.println("running and working...");
    }
    void run(String name) {
        System.out.println(name+" running and working...");
    }
}

interface Laptop{
    final String name = "Laptop";
    void run();
}

interface Desktop{
    final String name = "Desktop";
    void run();
}
class employee extends computer implements Laptop{
    @Override
    public void run() {
        System.out.println("working fine");
    }
}

public class Multiple{

    public static void main(String[] args) {
        employee lap = new employee();
        lap.run(lap.name);
    }
}
```

Super keyword :

Super keyword is used to call the parent values and can only be used in child class. Using super we can call the variables, constructors and methods in parent class.

Syntax :

super.variableName; - To invoke the parent variable

super.methodName(); - To invoke the parent Method

super(); - To invoke the parent constructor

Example :

```
class parent{
    parent(){
        System.out.println("parent");
    }

    void run(){
        System.out.println("Running parent");
    }
}

class child extends parent{
    child(){
        super();//parent
        System.out.println("child"); //child
    }

    void run(){
        System.out.println("Running child"); //Running child
        super.run(); //Running parent
    }
}
public class Multiple{

    public static void main(String[] args) {
        child c = new child();
        c.run();
    }
}
```

Polymorphism :

Polymorphism, in object-oriented programming, refers to the ability of objects to take on multiple forms or behaviors based on their data type or class hierarchy. It allows objects of different classes to be treated as objects of a common superclass.

With the help of method overloading and overriding we can achieve polymorphism. There are 2 types of polymorphism :

1. *Compile-time Polymorphism (Static Binding or Early Binding)*
2. *Runtime Polymorphism (Dynamic Binding or Late Binding)*

1. Compile-time Polymorphism (Static Binding or Early Binding) :

Method overloading is a common example of compile-time polymorphism, where multiple methods in the same class have the same name but different parameter lists.

Example :

```
class Calculator {
        int add(int a, int b) {
        return a + b;
        }
        double add(double a, double b) {
        return a + b;
        }
}
```

2. Runtime Polymorphism (Dynamic Binding or Late Binding) :

Method overriding is a common example of runtime polymorphism, where a subclass provides a specific implementation of a method that is already defined in its superclass.

Example :

```
class Animal {
        void sound() {
```

```
        System.out.println("Animal makes a sound");
        }
}
class Dog extends Animal {
        @Override
        void sound() {
        System.out.println("Dog barks");
        }
}
class Cat extends Animal {
        @Override
        void sound() {
        System.out.println("Cat meows");
        }
}
```

Dynamic Method Dispatch :

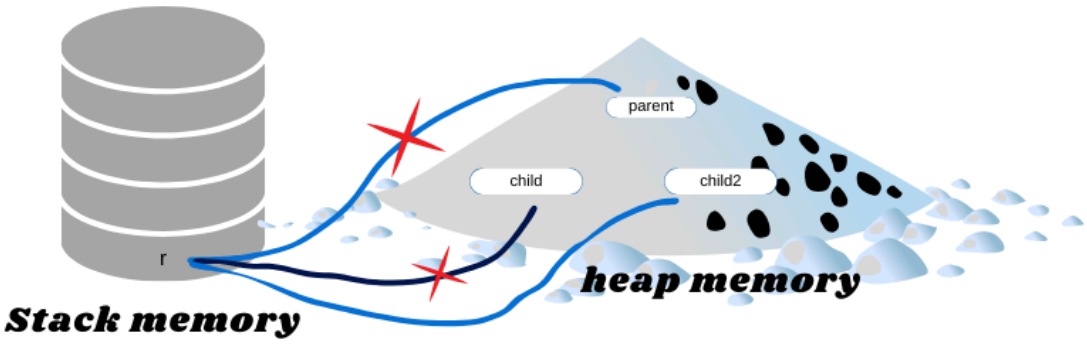
Dynamic method dispatch is a mechanism in Java (and other object-oriented programming languages) through which the method to

be invoked is determined at runtime rather than at compile time. It is also known as runtime polymorphism or late binding.

Example :

```
class parent{
    void run() {
        System.out.println("parent Running");
    }
}

class child extends parent{
        void run() {
        System.out.println("child Running");
    }
}

class child2 extends child{
    void run() {
        System.out.println("child2 Running");
    }
}

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

        parent r = new parent();
        r.run();
        r = new child();
        r.run();
        r = new child2();
        r.run();
        //output : parent Running
        //         child Running
        //         child2 Running

    }

}
```

Explaination :



Upcasting and Downcasting :

Upcasting and downcasting are two types of casting operations used when dealing with inheritance in object-oriented programming languages like Java. Here's an explanation of each:

1. **Upcasting**:

- Upcasting involves converting a reference of a subclass type to a reference of a superclass type.

- It is implicit and does not require any explicit casting syntax.

- Upcasting is always safe because it moves from a more specific type to a more

general type, ensuring that no information is lost.

- Upcasting allows you to treat a subclass object as if it were an instance of its superclass.

- **Example:**

Dog dog = new Dog(); // Creating a Dog object

Animal animal = dog; // Upcasting Dog to Animal

2. **Downcasting**:

- Downcasting involves converting a reference of a superclass type to a reference of a subclass type.

- It is explicit and requires the use of casting syntax. Downcasting is potentially unsafe because it moves from a more general type to a more specific type, which could lead to a `ClassCastException` if the object being casted is not actually an instance of the subclass.

- Downcasting is necessary to access subclass-specific members (methods or fields) that are not present in the superclass.

- **Example:**

Animal animal = new Dog(); // Upcasting Dog to Animal

Dog dog = (Dog) animal; // Downcasting Animal to Dog

Buffered Class :

Before learning the buffer class we will see the IO class which is the reason for both input and output. In java we are having a system class which contains both input and output static objects in IO package.

Input – will read one character at a time and converts the input to ANSII values.

Syntax :

VariableName =

System.in.read();

It asks us to throw IOExeception we need to add the statment at the main method.

Output – Will print the value.

In order to read the multiple set of values we use the buffered reader class which works along with the IO.

NOTE : To use the BufferedReader we need to import the java package of io.

Syntax for BufferReader :

BufferReader objName = new BufferReader(inputStream obj);

It needs a inputStream object which we need to write one more line which is shown in below.

Syntax for inputStream :

InputStreamReader in = new InputStreamReader(System.in);

NOTE : It will only read the values in string if we want to change them to integers then we have to use the parseInt().

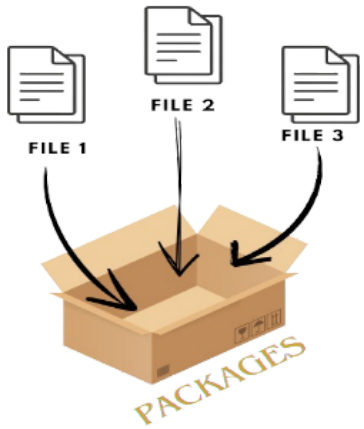
TIP : BufferReader is a versatile reader as it not only read from keyboard can read from files and browser

Package:

In java Packages are used to maintain a group of code/ files together. It helps in managing and organizing large codebases by grouping related classes together. Packages provide two main benefits:

1. Namespace Management

2. Access Control



1. ***Namespace Management :***

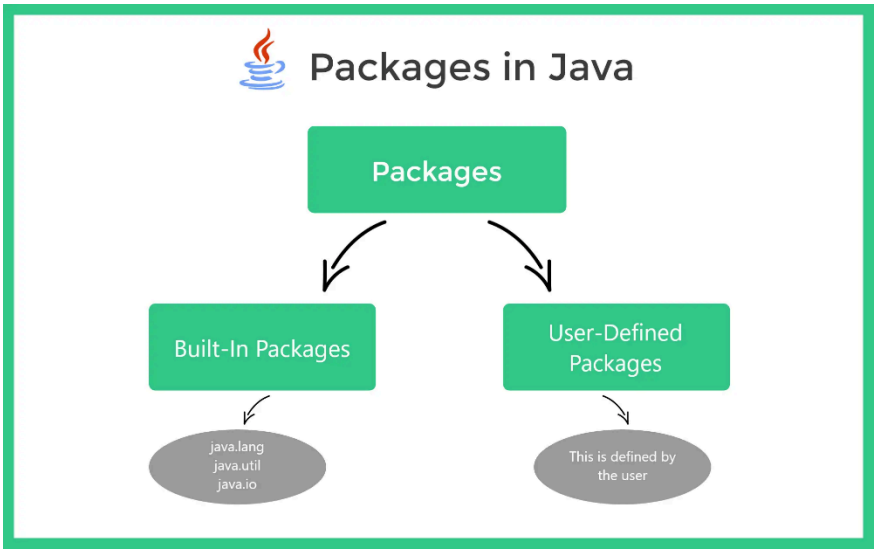
Packages prevent naming conflicts by providing a namespace for the classes they contain. Each class within a package is uniquely

identified by its fully qualified name, which includes the package name followed by the class name.

2. ***Access Control :***

Packages control access to classes and members using access modifiers (public, private, protected, default). Classes and members with default access are accessible only within the same package, while public and protected members can be accessed from other packages.

NOTE : Java is having the built in and user define packages



Creating a package :

Inorder to create a package we use the packageName followed package keyword.

Syntax :

```
package packageName;
```

Importing a Package :

Inorder to import a package to access a class we use the import keyword with the packageName.

Syntax :

```
import packageName.className;
```

Note : If we want to call all the classes then we use * . By default we are importing the java.lang package in our code.

It will be more useful when we are using the meaningful name for the packages

Access specifiers :

Access specifiers, also known as access modifiers, in Java are keywords that define the accessibility or visibility of classes, methods, and variables. Java provides four types of access specifiers:

1. Public

2. Private

3. Protected

4. Default (no Modifier)

1. ***Public :***

Public members are accessible from any other class. There are no restrictions on accessing public members.

2. Protected :

Protected members are accessible within the same package and by subclasses (even if they are in different packages). However, they are not accessible by non-subclasses outside the package.

3. Private :

Private members are accessible only within the same class. They cannot be accessed from outside the class, not even from subclasses.

4. Default :

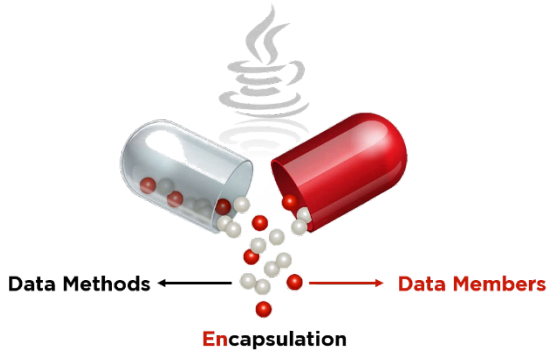
If no access specifier is specified, it is considered as default (also known as package-private). Default members are accessible only within the same package. They are not accessible from outside the package, even by subclasses.

Access Modifiers	Within the class	Within the Same Package	Subclass	In Other Packages
PUBLIC	Access Allowed	Access Allowed	Access Allowed	Access Allowed
PROTECTED	Access Allowed	Access Allowed	Access allowed	Access Denied
DEFAULT (NO ACCESS MODIFIER)	Access Allowed	Access Allowed	Access Denied	Access Denied
PRIVATE	Access Allowed	Access Denied	Access Denied	Access Denied

Encapsulation :

Encapsulation in OOP means grouping together data and the methods that work with it within a class. It keeps the internal details of

an object hidden from outside access and allows only specific interactions through defined interfaces**.**



Exception Handling :

Exception handling in Java is a mechanism used to handle runtime errors or exceptional situations that occur during the execution of a program. Exceptions are objects representing these abnormal conditions, such as division by zero, array index out of bounds, file not found, etc. Java provides a robust exception handling mechanism to deal with such situations, which involves the following key concepts:

1. **Try-Catch Blocks**:

A try-catch block is used to handle exceptions. The code that might throw an exception is enclosed within a try block, and the code that handles the exception is placed within one or more catch blocks.

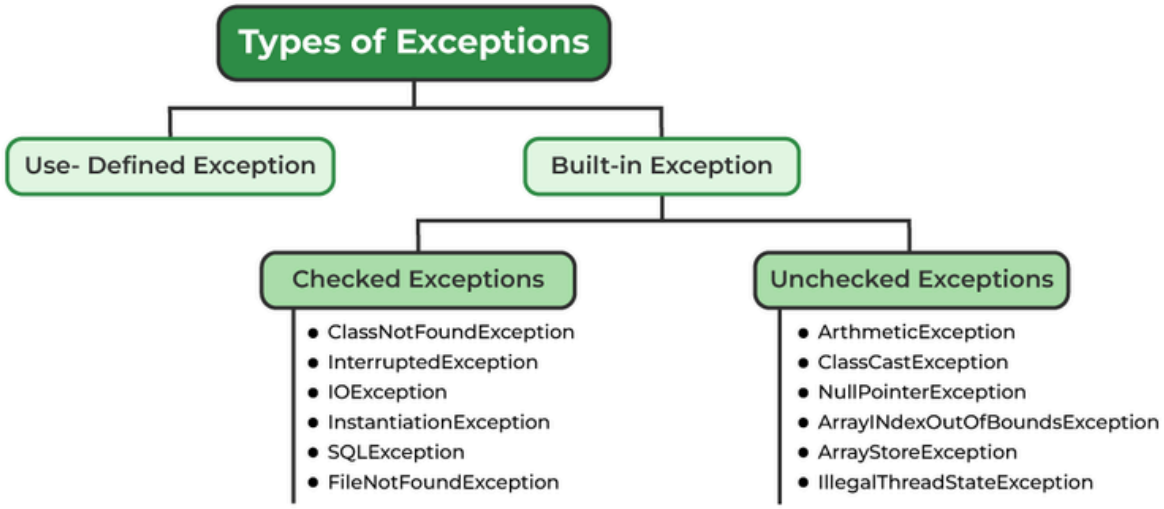
2. **Throwing Exceptions**:

You can manually throw exceptions using the `throw` keyword. This is useful for signaling exceptional conditions explicitly within your code.

3. **Checked and Unchecked Exceptions**:

Java has two types of exceptions:

1. checked
2. unchecked.



1. Checked Exception :

Checked exceptions must be either caught or declared using the `throws` clause in the method signature,

2. Unchecked Exception :

while unchecked exceptions (e.g., `RuntimeException` and its subclasses) do not need to be declared or caught.

*4. **Finally Block**:*

The finally block is used to execute cleanup code, such as closing resources, regardless of whether an exception occurs or not. It is optional and follows the try-catch block.

*5. **Multiple Catch Blocks**:*

You can have multiple catch blocks to handle different types of exceptions. They are checked in the order they appear, and only the first matching catch block is executed.

*6. **Exception Hierarchy**:*

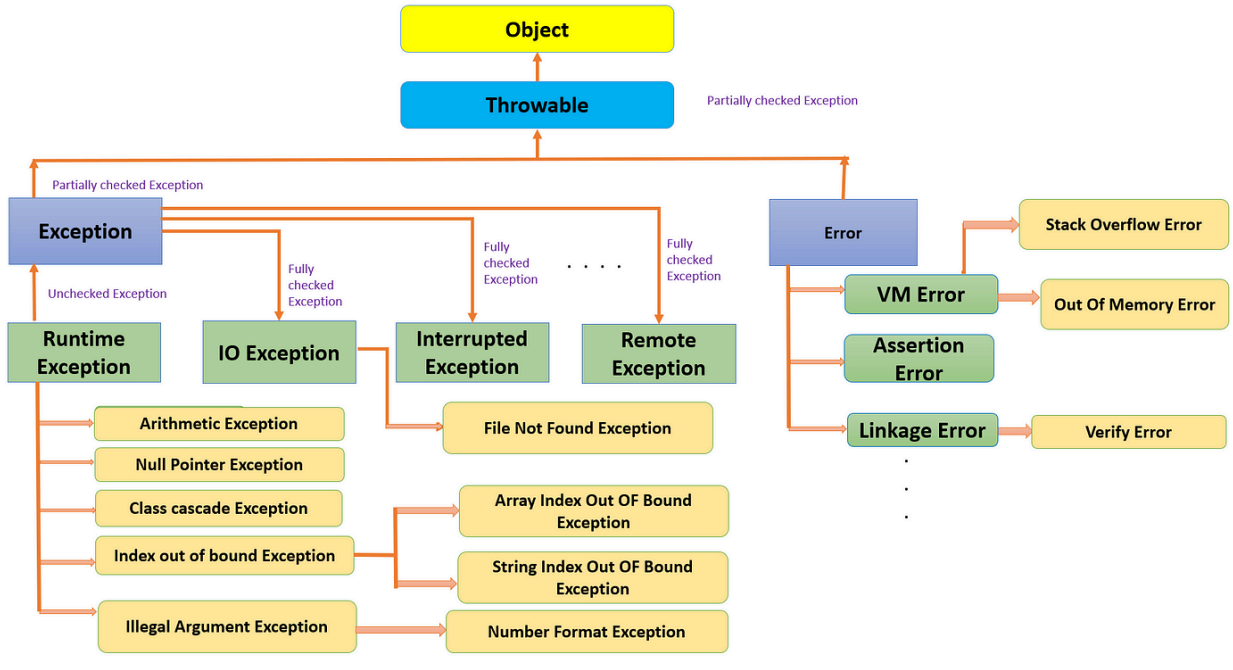


Fig. Exception Hierarchy in Java ~ by Deepti Swain

Exceptions in Java are organized in a hierarchy of classes. At the root of this hierarchy is the `Throwable` class, which has two main subclasses:

1. *`Error` (for severe errors that should not be caught)*
2. *`Exception` (for exceptions that can be caught and handled).*

Example :

```
public class Exception {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int divisor = 0;
        int dividend = 4;

        try {
            System.out.println(dividend/ divisor);
        }
        catch(ArithmeticException e) {
            System.out.println(e);
        }
        finally {
            System.out.println("Finally handled error");
        }
        //output :  java.lang.ArithmeticException: / by zero
        //          Finally handled error

    }
}
```

Threads :

Will make allow us to run the multi tasking at the same time by dividing the process into small threads.

As the CPU will only handle threads and not the process in order to run them simultaneously.

Utilization	Speed	
3%	1.07 GHz	
Processes	Threads	Handles
289	3388	117484
Up time		
0:20:47:45		

Multi-threading in java :

We can run the threads simultaneously without any issues and you can find the image below for the thread life cycle.

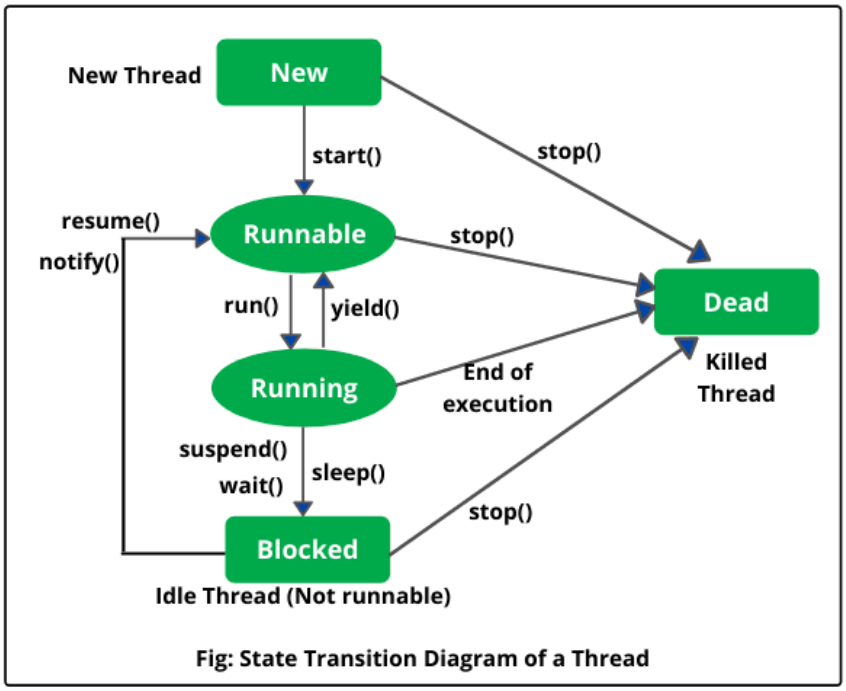


Fig: State Transition Diagram of a Thread

Thread class :

In java we are having the thread class which is responsible for implementing the threads. To code the thread we need to extends the threads class to the class which you want to make as thread.

Syntax :

```
Class className extends Threads{
        Public void run(){
                //threads code
        }
}
```

Example :

```java
class T1 extends Thread{
    public void run(){
        for(int i = 0; i < 100; i++){
            System.out.println("Inside Thread1");
        }
    }
}

class T2 extends Thread{
    public void run(){
        for(int i = 0; i <200; i++){
            System.out.println("Inside Thread2");
        }
    }
}

public class Threads {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        T1 t = new T1();
        T2 t2 = new T2();

        t.start();
        t2.start();
    }

}
```

Setting priorities of a thread :

We can set the priorities to the threads in order to run them. In general the schedular will take of the priorities and how to run the threads as a developer we can't control the schedular as it will work on it's won by using the defined algorithm.

To set the priorities we are having the setPriority and getPriority methods to set and get the priorities of a method.

Info : The priority ranges from 1 to 10, where 1 is the lowest priority and 10 is the highest. However, you can use the constants Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, and Thread.MAX_PRIORITY for clarity instead of hardcoding the values.

Syntax to get the priority :

```
Obj.getPriority();
```

Syntax to set the priority :

```
Obj.setPriority(value);
```

Sleep() :

Sleep in threads are used to place the threads into waiting mode. Sleep will take the millisecounds as input and it will throws an error.

Hence we need to place the sleep in try and catch block.

Syntax :

```
Thread.sleep(value);
```

Example :

```java
class T1 extends Thread{
    public void run(){
        for(int i = 0; i < 100; i++){
            System.out.println("Inside Thread1");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class T2 extends Thread{
    public void run(){
        for(int i = 0; i <200; i++){
            System.out.println("Inside Thread2");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Threads {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        T1 t = new T1();
        T2 t2 = new T2();

        t.start();
        t2.start();

    }

}
```

Implementing Threads using the Runnable Interface :

In java we can implement the thread in two ways.

1. Threads Class

2. Runnable Interface

Runnable interface is used as the java will not able to use the multiple inheritance if the class requires another class to inherit the

Threads can't be used. So as a solution we are having a runnable interface which will contains the run() method in it.

Points to remember :

1. In general when we are using the threads class we are able use the start() as it actually belongs to the Threads thought, Here we are having the interface of runnable which will not have any start().
2. Hence we have to create object for the Runnable and then we need to create thread object and need to pass the Runnable obj to the thread class.
3. Then it will work normally.

Syntax :

```
Class clsName implements Runnable{
    Public void run(){
        // code that needs to run
        }
}
Class Main{
PSVM(){
        Runnable obj = new clsName();
```

```
        Thread t1 = new Thread(obj);
        T1.start();
        }
    }
```

Example :

```java
class T1 implements Runnable{
    public void run(){
        for(int i = 0; i < 10; i++){
            System.out.println("Inside Thread1");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class T2 implements Runnable {
    public void run(){
        for(int i = 0; i <10; i++){
            System.out.println("Inside Thread2");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Threads {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Runnable task1 = new T1();
        Runnable task2 = new T2();

        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);

        thread1.start();
        thread2.start();

    }

}
```

We can also use the anonymous classes when creating object for interfaces. To do that we need to follow the below the below syntax :

Syntax :

```
interface obj = new interface(){
        //class body;
        };
```

Example :

```
Runnable task1 = new Runnable() {
        public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println("Inside Thread1");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

};
```

Join() :

Join() method is a special method in thread which allows the threads to reach the main method and then only main method will continue executing instead of finishing it 1st and wait for the threads to reach.

Sytnax :

```
threadObj.join();
```

threadObj.join();

Example :

```
public class Threads {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Runnable task1 = new Runnable() {
                public void run() {
                for(int i = 0; i < 10; i++){
                    System.out.println("Inside Thread1");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

        };
        Runnable task2 = new T2();

        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);

        thread1.start();
        thread2.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        try {
            thread2.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("Hello");

    }

}
```

Synchronized :

Synchronized is a keyword which helps the threads to work on a single method one at a time and make the code synchronize.

Syntax :

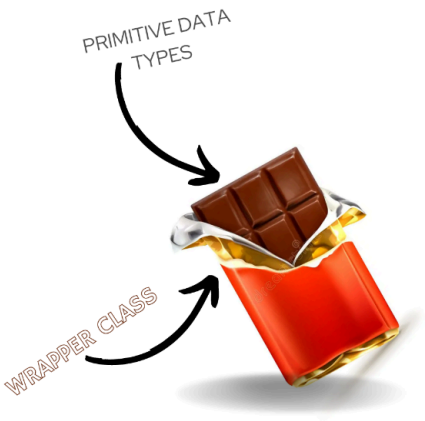
```
accessSpecifier synchronized returnType methodName(){
        //method
}
```

Wrapper Classes :

Wrappers classes are used to make the java 100% OOP. Why some of the frameworks will only works on objects and we are using the primitive data types inorder to store them. To correct that we are having this wrapper classes.

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

This is called wrapper as we are making the primitive type look like a object by wrapping the primitive type value just like a gift.



Autoboxing :

Auto boxing is when we are converting the primitive type to the wrapper class automatically then it is called as autoboxing.

Syntax :

wrapperClass obj = primitiveType;

Example :

```
int a = 5;
Integer aWrapper = a;
System.out.println("This is a wrapper class "+aWrapper);
```

Auto - Unboxing :

Unboxing is when we are manually converting the wrapper class to the primitive type then it is called as unboxing.

Syntax :

primitiveType variable = warpperObject;

Example :

```
int a = 5;
Integer aWrapper = a;
System.out.println("This is a wrapper class "+aWrapper);
int b = aWrapper;
System.out.println("This is a unBoxing wrapper class "+b);
```

Boxing :

In boxing we use the normal way of creating the object to a class, here we create the object for the wrapper class and will pass the value through constructor.

Syntax :

WrapperClass obj = new WrapperClass(primitiveValue);

Example :

```
int a = 5;
Integer a = new Integer(a);
}
```

The constructor Integer(int) has been deprecated since version 9 and marked for removal Java(67110276)

Unboxing :

In unboxing we use the built in methods to unbox the elements.

Syntax :

primitiveType variable = WrapperObj.PTValue();

Example :

```
float a = 5.8f;
Float a1 = a;
float b = a1.floatValue();
System.out.println(b);
```

Methods:

- ***valueOf**:** Static method that returns an instance of the wrapper class representing the specified primitive value.

Integer intValue = Integer.valueOf(10);

- ***parseXxx**:** Static method that parses the specified string and returns the primitive value. (`Xxx` represents the primitive data type).

int intValue = Integer.parseInt("10");

- ***xxxValue**:* Instance method that returns the primitive value of the wrapper object.

int intValue = intValueObject.intValue();

- ***toString**:* Instance method that returns a string representation of the object.

String stringValue = intValueObject.toString();

- ***compareTo**:* Instance method that compares the value of the current object with another object of the same type.

int comparisonResult = intValueObject.compareTo(anotherIntValueObject);

- ***equals**:* Instance method that checks if the current object is equal to another object.

boolean isEqual = intValueObject.equals(anotherIntValueObject);

- ***hashCode***: Instance method that returns the hash code value of the object.

int hashCodeValue = intValueObject.hashCode();

```
Integer a;
a = Integer.parseInt("23");
System.out.println(a);
```

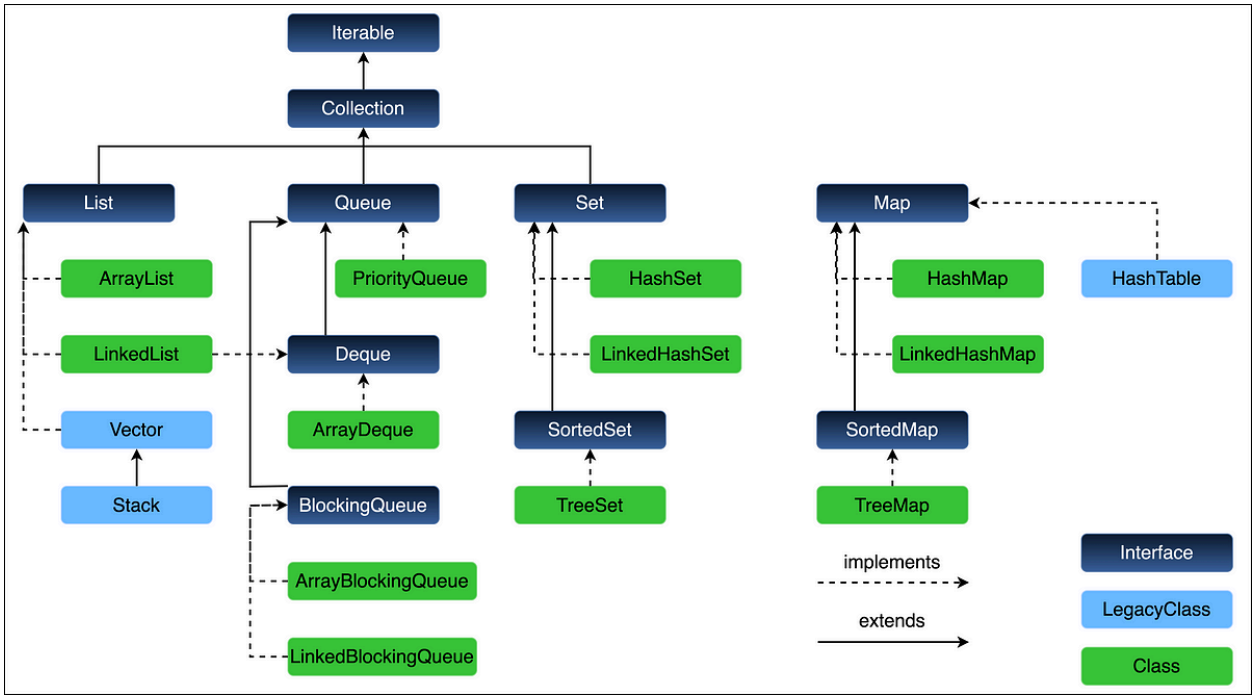
Collection API :

The Collection API in Java provides a framework for representing and manipulating collections of objects. Collections are groups of objects, typically stored in data structures like lists, sets, or maps. The Java Collections Framework includes interfaces, implementations, and algorithms that allow for the manipulation and management of collections of objects.

Here's an overview of some key interfaces in the Java Collections Framework:

- **Collection:**

  This is the root interface of the collections hierarchy. It defines the basic operations that a ll collections support, such as add, remove, contains, and size.



List :

A collection that maintains the order of its elements. Some common implementations include ArrayList, LinkedList, and Vector.

List is a interface which can be implements by different classes. Unlike static collections lists are dynamic and can shrink and grow according to the values automatically.

Interface Methods :

Add() : Used to add elements to the list

Remove() : Used to remove an element from list using the index value

Get() : Used to get the element from the list using the index value

Size() : used to get the size of the list.

IsEmpty() : Will check if the list is empty or not.

Array List :

ArrayList is a class which can implement the lists or collection interface.

Implements a dynamic array that can grow as needed. It provides fast random access and is good for scenarios where elements are frequently added or removed from the end of the list.

Syntax :

```
Collection obj = new ArrayList();
```

Example :

```java
public class ArrayCollection {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Collection c = new ArrayList();
        c.add(1);
        c.add(2);
        c.add(3);
        c.add(4);
        System.out.println(c);
    }
}
```

The problem of the collections are it will take the values as object values and it will not fall under any specific datatype. So inorder to provide the data type to the values we are adding in the array we will use the wrapper class after the collections and array list constructor we call them generics.

Syntax :

collection <wrapperClass> obj = new ArrayList<wrapperClass>();

NOTE : With the help of the specificity we are removing errors in the arrayList as normally the array list is a object hence it will accept the Strings as well. If I need to perform any mathematical operations then the error will occur as it is a String.

ArrayList Methods:

1. **add(E e)**:

Appends the specified element to the end of this list.

2. **addAll(Collection<? extends E> c)**:

Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

3. **clear()**:

Removes all of the elements from this list.

6. **contains(Object o)**:

Returns `true` if this list contains the specified element.

7. **get(int index)**:

Returns the element at the specified position in this list.

8. **indexOf(Object o)**:

Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

9. **isEmpty()**:

Returns `true` if this list contains no elements.

10. **iterator()**:

Returns an iterator over the elements in this list in proper sequence.

11. **remove(Object o)**:

Removes the first occurrence of the specified element from this list, if it is present.

12. **remove(int index)**:

Removes the element at the specified position in this list.

13. **removeAll(Collection<?> c)**:

Removes from this list all of its elements that are contained in the specified collection.

14. **retainAll(Collection<?> c)**:

Retains only the elements in this list that are contained in the specified collection.

15. **size()**:

Returns the number of elements in this list.

16. **toArray()**:

Returns an array containing all of the elements in this list in proper sequence.

17. **toArray(T[] a)**:

Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

NOTE: We can directly print the collections without the help of a loops

Linked List :

Linked list is similar to arrays but doubly-linked list. It's efficient for adding or removing elements from the beginning, middle, or end of the list.

Syntax :

List<wrapperClass> obj = new LinkedList<wrapperClass>();

Linked list also uses the same methods as ArrayList();

Example :

```
List<Integer> list = new LinkedList<Integer>();
list.add(5);
list.add(6);
list.add(7);
list.add(8);
System.out.println(list);
list.remove(0);
System.out.println(list);
```

Stack:

Represents a last-in, first-out (LIFO) stack of objects. It provides push and pop operations for adding and removing elements from the top of the stac

Creating a Stack:

You can create a stack using the Stack class(Legacy) or by initializing a Deque with one of its implementations (ArrayDeque or LinkedList).

Stack Methods :

Push: To add an element to the top of the stack, you use the push() method.

Pop: To remove and return the element at the top of the stack, you use the pop() method.

Peek: To view the element at the top of the stack without removing it, you use the peek() method.

Empty Check: To check if the stack is empty, you use the isEmpty() method. k.

Creating stack using the arraydeque :

```
package streamAPIs;

import java.util.ArrayDeque;
import java.util.Deque;
```

```
	public class StackCLass {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Deque d = new ArrayDeque();
		d.push(1);
		d.push(2);
		d.push(3);
		d.push(4);
		d.push(5);
		System.out.println(d);
		System.out.println(d.pop());
		System.out.println(d);

	}
}
```

Creating stack using Stack class :

```
	package streamAPIs;
	import java.util.ArrayDeque;
	import java.util.Deque;
	import java.util.Stack;

	public class StackCLass {
		public static void main(String[] args) {
		// TODO Auto-generated method stub
			Stack s = new Stack();
```

```
                    s.push(1);
                    s.push(2);
                    s.push(3);
                    s.push(4);
                    s.push(5);
                    System.out.println(s);
                    s.pop();
System.out.println(s);

        }

}
```

NOTE : It is recommended to use the deque as the Stack class is a Legacy one and for more features use the deque interface.

Set :

- Set: A collection that does not allow duplicate elements. Common implementations include HashSet, TreeSet, and LinkedHashSet.

Set is similar to the List as it is also interface but unlike list it will not store dulipcate values and also don't deal with the index values.

Interface Methods :

Add() : will add the elements into the set

Remove() : will remove the elements from the set, if present

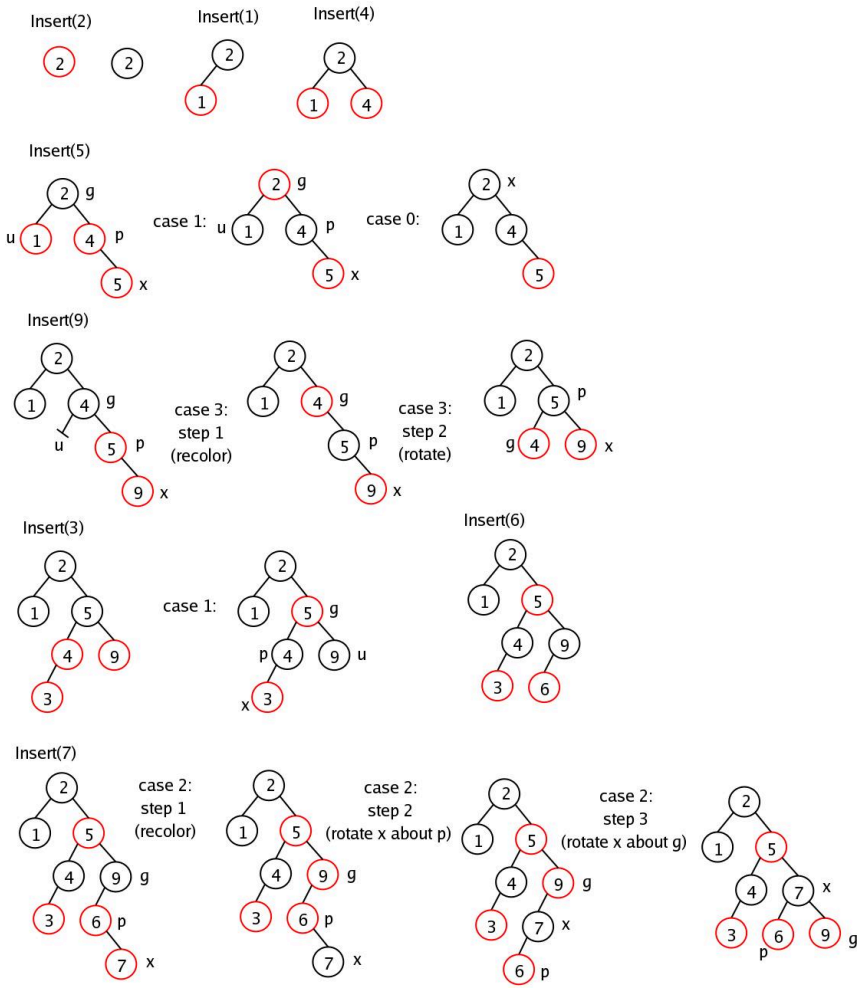
Contains() : will check if the elements are present inside the set or not.

Size() : will provide the size of the set.

IsEmpty() : will check if the set is empty or not.

TreeSet :

TreeSet follows tree structure in order to sort the values in the set. It follows the red black binary tree in order to store the values.



Additional Methods :

1. *pollFirst() – retrieves and removes the lowest element*
2. *pollLast() – retrieves and removes the highest element*
3. *first() – returns the lowest element*
4. *last() – returns the highest element*

Sytnax :

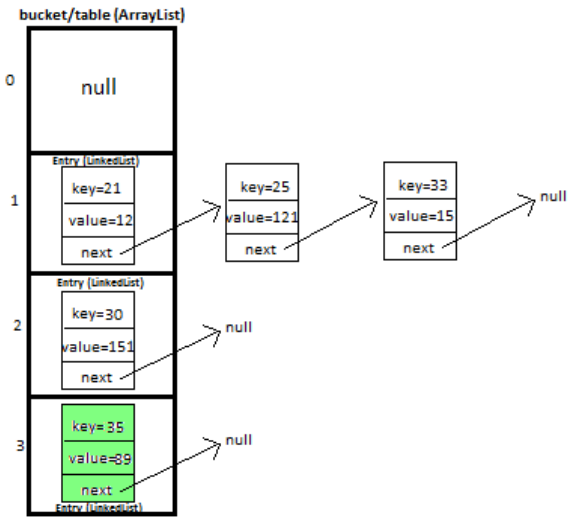
set<wrapperClass> obj = new TreeSet<wrapperClass>();

Example :

```
Set<Integer> sets = new TreeSet<Integer>();
sets.add(1);
sets.add(2);
sets.add(3);
sets.add(1);
System.out.println(sets);
```

HashSet :

It is also similar to the TreeSet but the only difference is it will not show data in a sorted order.



Additional Methods :

hashCode() – used to return the hashcode

Syntax :

Set<wrapperClass> obj = new HashSet<WrapperClass>();

Example :

```
Set<Integer> sets = new HashSet<Integer>();
sets.add(31);
sets.add(24);
sets.add(13);
sets.add(02);
sets.add(11);
System.out.println(sets);
```

LinkedHashSet :

LinkedHashSet is a class in Java that combines the features of both HashSet and LinkedHashMap. It implements the Set interface and internally uses a hash table along with a linked list to maintain the insertion order of elements.

Syntax :

Set<wrapperClass> obj = new LinkedHashSet<Integer>();

Example :

```
Set<Integer> sets = new LinkedHashSet<Integer>();
sets.add(31);
sets.add(24);
sets.add(13);
sets.add(02);
sets.add(11);
System.out.println(sets);
```

Map:

A collection that maps keys to values. Each key can map to at most one value. Common implementations include HashMap, TreeMap, and LinkedHashMap, HashTable.

Methods :

put(K key, V value) : Used to insert the values

get(Object key) : Used to retrieve the values from map.

containsKey(Object key) : Used to check the values in map

containsValue(Object value) : Used to check the values in map

remove(Object key) : Used to remove the values from the map

keySet() : Used to get the keyset from the maps

values() : Used to get the values from the maps

entrySet() : Used to gets the entire sets

size() : Used to get the size of the sets

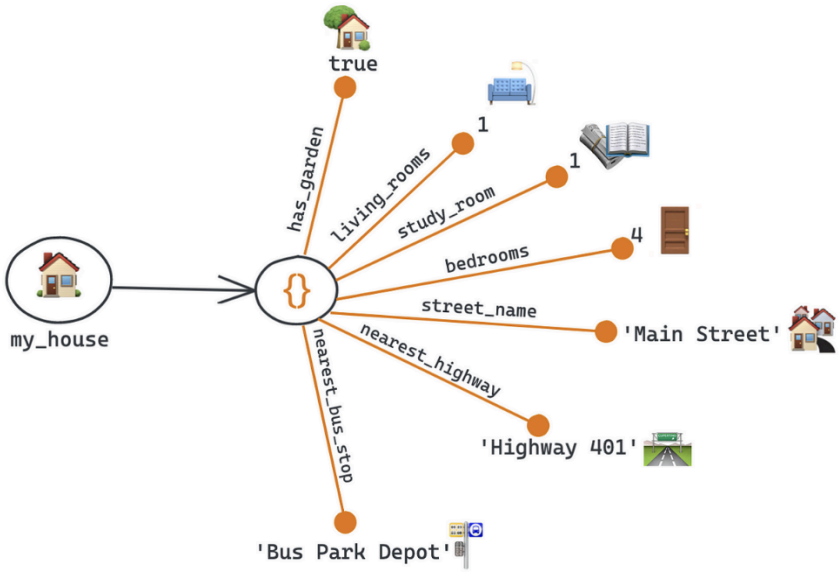
isEmpty() : Used to check the set is empty or not

Syntax :

Map<wrapperClass for Key, wrapperClass for Value> obj = new HashMap<>();

HashMap :

As in the above example we can save the values with the help of keys which will help us to track the elements inside the sets.



Syntax :

Map<wrapperClass, WrapperClass> sets = new HashMap<>();

Example :

```
Map<String, Integer> sets = new HashMap<>();
sets.put("a", 1);
sets.put("b", 2);
sets.put("c", 3);
sets.put("d", 4);
System.out.println(sets);
```

TreeMap :

Implements the Map interface using a Red-Black tree. It maintains the elements in sorted order.

Syntax :

Map<wrapperClass, WrapperClass> sets = new TreeMap<>();

Example :

```
Map<String, Integer> sets = new TreeMap<>();
sets.put("a", 1);
sets.put("b", 2);
sets.put("c", 3);
sets.put("d", 4);
System.out.println(sets);
```

LinkedHashMap :

Extends HashMap and maintains the insertion order of elements. LinkedHashMap provides constant-time performance for basic operations like get and put, with slightly higher memory overhead due to maintaining the linked list.

Syntax :

Map<wrapperClass, WrapperClass> sets = new LinkedHashMap<>();

Example :

```
Map<String, Integer> sets = new LinkedHashMap<>();
sets.put("a", 1);
sets.put("b", 2);
sets.put("c", 3);
sets.put("d", 4);
System.out.println(sets);
```

Hashtable :

A legacy implementation of the Map interface that is synchronized (thread-safe), but it's generally slower than HashMap due to the overhead of synchronization.

Syntax :

Map<wrapperClass, WrapperClass> sets = new Hashtable<>();

Example :

```
Map<String, Integer> sets = new Hashtable<>();
sets.put("a", 1);
sets.put("b", 2);
sets.put("c", 3);
sets.put("d", 4);
System.out.println(sets);
```

- ***Queue:*** *A collection used to hold elements prior to processing. Common implementations include LinkedList and PriorityQueue.*
- ***Deque:*** *A double-ended queue that supports element insertion and removal at both ends. Common implementations include ArrayDeque and LinkedList.*
- ***Iterator:*** *An interface that provides a way to iterate over elements in a collection.*
- ***Comparator:*** *An interface used to define custom ordering for objects.*

Iteratable :

Iteratable is a interface which is the superior than collection class and it is a interface and it contains the iterator method which will throw the elements inside the collections.

Syntax :

Iterator<wrapperClass> obj = collection.Iterator();

Example :

```
Iterator<Integer> it = sets.iterator();
while (it.hasNext()) {
  System.out.println(it.next());
}
```

It uses the two built in methods**.**

1. Next()

2. HasNext()

1. Next :

Will return the value of the collections one at a time.

2. HasNext :

Will return boolean values and it check if the collection is having the next element or not.

true – if elements are there

false - if elements are not present

Sort :

Sort is a built in method to sort the collections.

Syntax :

Collections.sort(obj);

Stream API :

Provides a way to process collections of objects in a functional style. It allows you to perform operations such as filtering, mapping, reducing, and iterating over collections easily and efficiently. Streams enable you to write expressive and concise code for data manipulation and transformation.

Basic Stream Operations:

1. **Creating Streams:**

- You can create a stream from a collection using the `stream()` method:

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
```

- You can also create a stream from an array using the `Arrays.stream()` method:

```
String[] array = { "a", "b", "c" };
Stream<String> stream = Arrays.stream(array);
```

2. **Intermediate Operations:**

- Intermediate operations are used to transform, filter, or manipulate the elements of the stream. Examples include `map()`, `filter()`, `distinct()`, `sorted()`, etc.

Map() – used to point the data without using index values and it will return streams

Filter() – used to use a condition and if it is returning true then we will store it inside new stream or else ignore

Distinct() – will remove the duplicates from the collection

Sorted() – Will sort the collections

3. **Terminal Operations:**

- Terminal operations consume the elements of the stream and produce a result or side-effect. Examples include `forEach()`, `collect()`, `reduce()`, `count()`, `anyMatch()`, `allMatch()`, `noneMatch()`, etc.

Example Usage:

Filters :

```
ArrayList<Integer> a = new ArrayList<>();
        a.add(1);
        a.add(2);
        a.add(3);
        a.add(4);
        a.add(5);
        Stream<Integer> s = a.stream();

        Stream<Integer> i = s.filter(
                                value ->{
                                        if((int)value % 2 == 0)
                        return true;
                else
                        return false;
```

```
                    }
                );
        i.forEach(value -> System.out.println(value));
```

map :

```
    LinkedList ls = new LinkedList();
            ls.add(1);
            ls.add(2);
            ls.add(3);
            ls.add(4);
            ls.add(5);
            System.out.println(ls);
            Stream s = ls.stream();
            Stream a = s.map( value -> value);
            a.forEach(value -> System.out.println(value));
```

Characteristics of Streams:

- **Pipelining**:

Stream operations can be chained together to form a pipeline, where the output of one operation is the input for the next operation.

- **Lazy Evaluation**:

Stream operations are evaluated only when necessary. Intermediate operations are usually lazy and will not be executed until a terminal operation is invoked.

- **Stateless and Stateful Operations**:

Intermediate operations like `map()` and `filter()` are stateless (they do not depend on the state of other elements), while operations like `sorted()` are stateful (they may depend on the state of other elements).

- **Parallel Execution**:

Streams can be processed in parallel by utilizing the `parallelStream()` method, which can improve performance for large datasets on multi-core systems.

NOTE : The stream() will allow us to use the stream methods simultaneously.

Files Handling :

In java we can handle files by using the buffered reader. To do that we need to follow some steps in order to achieve this

To add read a file we need to mention the file name inside the constructors of file.

Syntax :

File fileObj = new File(“ FileName.extension “);

**Note : We need to include the \\ or can use one forward slash / for the address as that is the escape sequence for the slash **

We can check if the file exist or not with the help of built in method exist().

Syntax :

fileObj.exists(); //returns the true or false

To get the location we can use the built in method called getPath().

Syntax :

fileObj.getPath(); // returns file name

To get the absolute path we use the getAbsolutePath() method to get the complete address.

Syntax :

fileObj.getAbsolutePath(); // returns the entire location of the file

To get the know if the file is a file or not we use the isFile() built in method to check.

Syntax :

fileObj.isFile(); // returns boolean values

To delete the file we need to use the delete built in method

Syntax :

```
fileObj.delete();
```

Writing a File :

To write the file we need to use the FileWriter class

Syntax :

```
FileWriter obj = new FileWriter( " path or fileName",
boolean );
```

Where Boolean = true or false : to append à true

To write in file we use the write method to write.

Syntax :

```
Obj.write("information ");
```

We can append the text to the files using the append method

Syntax :

```
obj.append("text/ info");
```

Reading a file :

To read a file we use the FileReader class

Syntax ;

FileReader FRobj = new FileReader(“path”);

To read we need the help of bufferedReader

Syntax :

BufferedReader BRObj = new BufferedReader(FRobj);

To get the text from the file we use the readLine() built in method.

Syntax :

String varName;

VarName = BRObj.readLine();

We need to read them until we get the null value or the –1

Syntax :

String varName;

While(varName != null){

System.out.println(varName);

VarName = BRObj.readLine();

}

Example :

```
String path = "D:/Secound.txt";
        try {
                File file = new File(path);
```

```
            FileWriter writer = new FileWriter(file, true);
            writer.write("\nHello");
            writer.write("\nChecking");
            writer.close();

            FileReader reader = new FileReader(path);
            BufferedReader br = new BufferedReader(reader);

            String line;

            while( (line = br.readLine()) != null ) {
                System.out.println(line);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
```

Serialization :

Serialization in Java refers to the process of converting an object into a stream of bytes so that it can be easily stored to a file, sent over a network, or persisted in a database. This process allows the object's state to be saved and later reconstructed when needed. In Java, serialization is achieved by implementing the `Serializable` interface, which marks the class as serializable. Once marked as serializable, objects of that class

can be converted into a byte stream using Java's built-in serialization mechanisms, typically through `ObjectOutputStream`.

Example :

```
package com.File;
 import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Serializable;
import java.util.Scanner;
class student implements Serializable{
        private String name;
        private int rollno;
        student(String name, int rollno){
                this.name = name;
                this.rollno = rollno;
        }
        public String getName() {
```

```
            return name;
        }
        public int getRollno() {
            return rollno;
        }
}
public class MainMethod {
        public static void main(String[] args){
            String path = "D:/Student.ser";
            try {
                File file = new File(path);
                FileWriter writer = new FileWriter(file, true);
                student s1 = new student("mike", 34);
                student s2 = new student("nike", 4);
                student s3 = new student("moye moye", 24);
                student s4 = new student("Ante adhi! Ram", 34);
                student[] students = {s1, s2, s3, s4};
                for(student data : students) {
```

```
                writer.write(data.getName() + " " +
            data.getRollno() + "\n");
            }
            writer.close();
            FileReader reader = new FileReader(path);
            BufferedReader br = new BufferedReader(reader);

            String line;
            while( (line = br.readLine()) != null ) {
                System.out.println(line);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Enumerations :

Enums are similar as class. We can have methods, constructors everything in enum, except for the inheritance. We can't inherit the enum to any class.

Creating an Enum :

```
Enum enumName{
    //values and methods
}
```

In Enum in we have a fixed set of values, which are typically represented as public static final constants.

Example :

```
enum test{
    working, failed, warning, error;
}
```

To access the enum is also very simple it will looks like how we actually stores an value to a variable.

Syntax :

```
enumName obj = enumName.constantValue;
```

Example :

```
test t1 = test.working;
        System.out.println(t1);
        test[] t2 = test.values();
```

```
        List<test> values = Arrays.asList(t2);

        values.forEach((value) -> System.out.println(value));
```

As like a normal data we can use the conditional statements to check or compare the data.

Comparing data using the conditional Statements :

Using if :

```
        test t1 = test.working;

        if(t1 == test.error) {
                System.out.println("error");
        }
        else if(t1 == test.warning) {
System.out.println("warning");
}
        else if(t1 == test.failed) {
System.out.println("failed");
}
        else if(t1 == test.working) {
System.out.println("working");
}
        else {
                System.out.println("404 : NotFound");
        }
```

Output :

Working

Using the switch :

```
        test t1 = test.working;

                switch (t1) {
                case working:
                        System.out.println("working");
        break;
                case failed :
                        System.out.println("failed");
        break;
        case warning:

System.out.println("warning");
        break;
        case error:

System.out.println("error");
        break;
        default:

System.out.println("default");
```

```
        break;
```

In java enum will hold the constant values which are nothing but the objects. Hence we can use the constructor as well.

Example :

```
        enum test{
        admin("admin rights"), user("user rights");
        final String msg;
        test(String msg){
                this.msg = msg;
        }
}

public class MainMethod {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                        test t1 = test.admin;
                        System.out.println(t1);//admin
                        System.out.println(t1.msg); //admin rights
                        test t2 = test.user;
                        System.out.println(t2);//user
                        System.out.println(t2.msg); //user rights
                }
        }
```

NOTE : We can specify the constructor and we can use the any constructor based on the parameters we are passing.// Constructor Overloading

NOTE : We can even use the private constructor as we are creating the constructors in the same class itself.

Annotation :

In order to provide some extra info to the compiler with the help of annotations. Annotations will help us in debugging the logical part.

Example :

```
	class A{
	public void check1() {
		System.out.println("Checking");
	}
}

class B extends A{
	@Override
	public void check() {
	System.out.println("Checking B");
	}
```

```
}
public class MainMethod {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
			B b = new B();
			b.check1();
		}
	}
```

In the above code we are trying to override and we mentioned that with the help of annotation that we are looking to override it and compiler will check If we are making any mistakes or not. If we are then it will throw some errors.

NOTE : In general we don't use the annotations in java much though we them in frameworks more.

Var :

Var is a Local Variable Type Inference which was introduced in java 10 to make the java more easy and understandable.

Syntax :

Var variableName;

Example :

```
        var i = 5;
    var s = "Hello";
    var f = 2.5;
    var list = new ArrayList();
    var sets = new TreeSet();
    var map = new HashMap();
```

Note : When we are using the var we need to initialize the value.

Sealed Classes :

In java we use the final or the abstract classes to hide the data from the others. But if we want to inherit the class for some classes the we can use the sealed class. In sealed class the programmer can provide the permission to extend this class to child class.

Syntax :

```
Sealed class className{
    //body
}
```

Example :

```
sealed class A permits B{

}
```

```
non-sealed class B extends A{

}
```