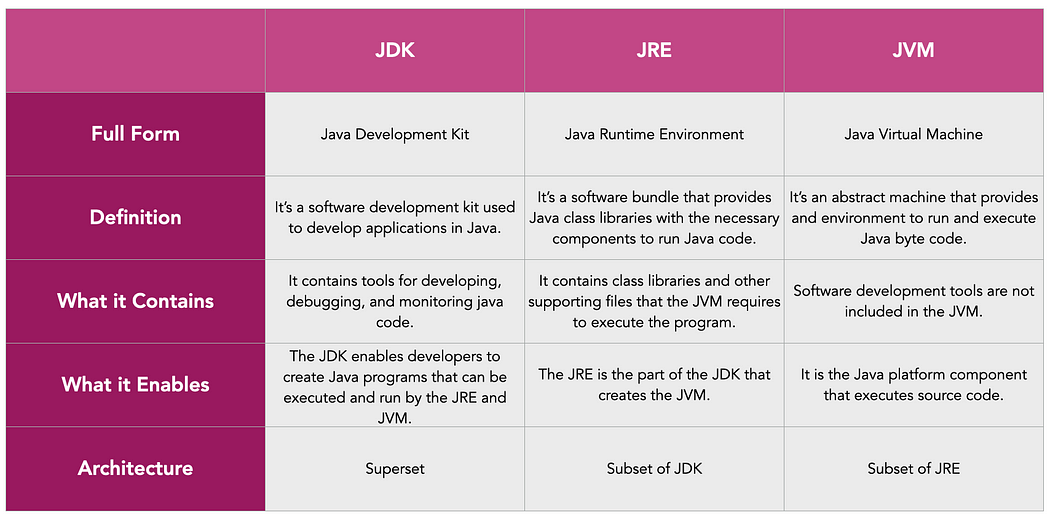
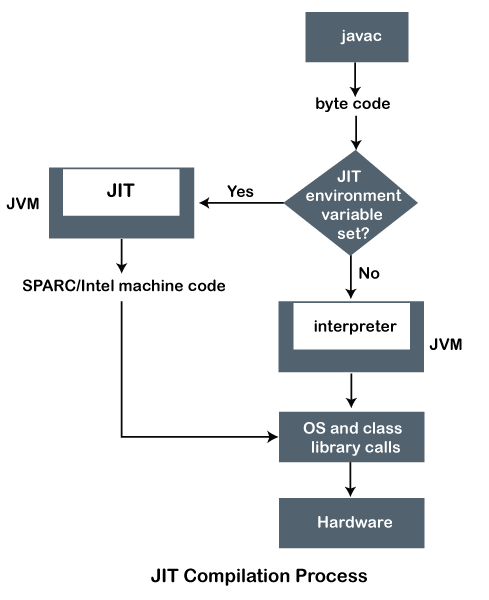
Core java:

* Package : Containers that groups related java elements [ classes, interfaces]
* Types of packages:
* Built-in : prewritten packages included into java
* User defined: defined by the developer
* Features: -
* Classes and objects : properties or state and behavior or functions
* Access modifiers: private, public, protected, default
* Data Abstraction: Security and confidentiality
* Inheritance: only multiple inheritance is not allowed due to **diamond problem**
* Encapsulation: bundles the data and the code working on that data in a single unit.(loosely coupled code.)
* Abstraction
* Polymorphism
* Portable [WORA: write once run anywhere]
* Object Relationships:
* Is-a relationship: Achieved through inheritance (parent child relationship)
* Has-a relationship: Whenever an object is used in other class. (could be one-one, one-many, many-many). Eg class A has object of class B.
* Association: relationship between 2 different objects.

1. Aggregation : both objects. Can survive individually, means ending of one object will not end other object.
2. Composition: Ending of one object will end another object.

* JDK= JRE+ [Programming language +Compiler+ Debugger]
* JRE= JVM+ [Class path Libraries]
* Byte Code is platform independent but JDK, JVM, JRE are platform dependent
* JSE: Java Standard edition, core java
* JEE: Java Enterprise edition/ Jakarta EE, includes APIs like transaction, rollback, servlets, JSP, persistentAPI
* JME: Java Micro/Mobile edition: APIs for mobile applications
* 
* JIT: Just In Time Compiler , part of JVM
* Follows 2 Approaches AOT( Ahead- Of-Time Compilation) , Interpretation
* AOT: Compiles code into a native machine language, bytecode -> to machine code
* Interpretation: code to machine code (.class file bytecode -> machine code)
* 
* Java components: JVM,JDK,JRE are platform dependent only bytecode is platform independent
* JVM: abstract machine[ does not exist physically], it converts bytecode[Compiler generates bytecode/ .class file] into machine code so it can be read by CPU then we get the output. Not platform independent. IT has JIT(just in time compiler)
* JRE: Java Runtime Environment, it has JVM and Class libraries
* JDK: Java Development Kit, consists JRE+program language(.java file), compiler(javac) (.class file), Debugger
* OOP’s
* Object Oriented programming
* Object: Real world entity
* Loosely coupled
* Data hiding (data binding)

1. Objects: has 2 things: 1) Properties or state, Behaviour or function
2. Classes: template for object
3. Data Abstraction: for Security confidentiality purpose, achieved through interface and abstract class
4. Data Encapsulation: Bundles the data and the code working on that data in a single unit, Data hiding, loosely coupled code
5. Inheritance:

* Capability of class to inherit properties from their parent class
* Types: Single, Multilevel, Hierarchical, Multiple [not supported, Diamond problem: Solved using Interface]
* Object Relationships:

1. Is- a: Achieved through inheritance
2. Has-a: Object of other class used in Different class, It can be One-one, one-many, many-one

Has a includes: Association: relationship between two different objects

[Weak Relation: Aggregation: both objects can serve individually, means ending of one object will not end of another object. Eg. Student- school]

[Strong Relation: Composition: Ending of one object will end another object. Eg. School - classroom]

1. Polymorphism

* Poly means many and morphism means form
* It behaves differently in different situations
* Types:

1. Compile time/ Static / Method overloading: In same class with same name of method name but different parameters

[Note: Using different return type is not correct way to achieved overloading, because it does not check return type at compile time]

1. Run Time/ Dynamic Polymorphism/ method Overriding: parent child relationship, where child extends method from parent which has same name, same return type also has same parameter list so at the run time it decides which method need to call depending on the object which class object we have created. [Inheritance]

* Variables:
* Static Type language: which has data type for variable
* Strong type language: there is specific limit to for variable to store value
* Types :

A] Primitive types

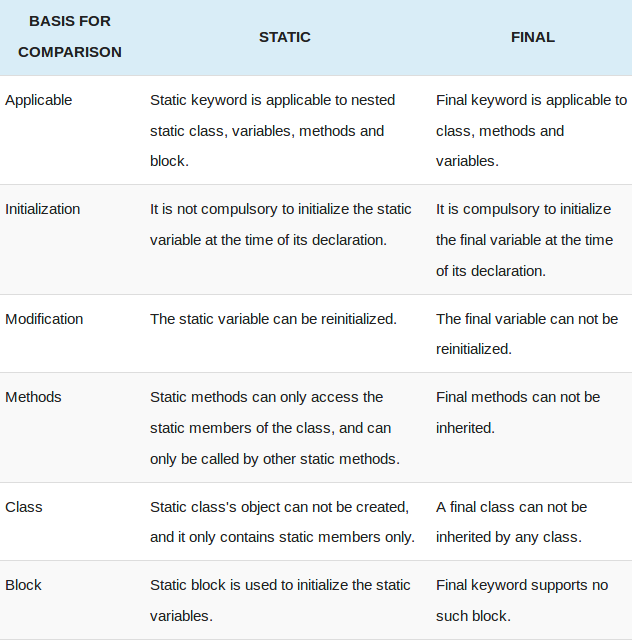
1. char
2. byte
3. short
4. int
5. long
6. float
7. double
8. Boolean

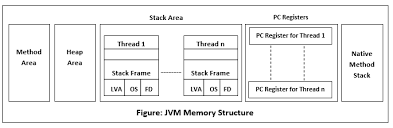
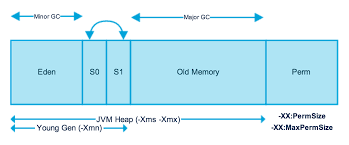
B] Reference Data Type/ Non- primitive: Wrapper classes(AutoBoxing [Primitive to its wrapper class] and Unboxing [Wrapper class to it primitive type] ) : In java everything is pass by value

Object is created using new then it will be stored on heap

Class, String, Integer, Double, Character: These objects hold the reference of actual memory [ It is used to achieve what we achieved using pointer in c++]

String: Immutable, inside heap there is String constant pool stores string literals.



* Constants: Capital letters variables, final eg. static final VAR\_NAME=10
* Kind of Variables:
* Local: inside methods
* Instance/ member variable
* Static/ class variable: associated with class, can be accessed without object or using Classname.static\_varName
* Method parameters
* Constructor parameters
* How Java stores Float vs Double, [IEEE 742]
* Type of Conversion:
* Widening/ Automatic Conversion
* Narrowing/ Down casting/ Explicit Conversion: (Datatype) variable\_name;
* Promotion during expression: when datatype var is reaches max limit it will promote datatype accordingly. eg. Byte a=127, byte b=1; [ byte ans= a+b; //not possible] so int ans= a+b;
* Explicate casting during expression
* Methods: collection of instructions to perform certain task
* Types of methods: System Defined
* User Defined
* Overloaded
* Overridden
* Static method: associated with the class, can be called using class name, cannot access non static instance variable and methods, cannot be overridden.[ Use when methods which do not modify the state of the object or computation purpose, when calculation is totally depends on arguments]
* Final method: Can not be overridden, because child or any other class cannot be able to change final method’s implementation.
* Abstract method: Defined only in Abstract class, only method declaration is done, child class need to implement overridden method.
* Variable arguments (Varargs): variable number of inputs in the parameter, only one variable argument can be present in the method, It should be the last argument in the list. Eg. int sum(int …var)
* Access Specifiers:
* Public: can be access through any class in any package
* Private: can be access by methods only in the same class
* Protected: can be access by other classes in same package or other sub-classes in different package
* Default: if we do not mention anything, then default access specifier is used by Java. It can be only accessed by classes in same package
* Constructor: same as class name but without return type, [can not be static , final , abstract or synchronised]
* Types: new used to tell runtime that we have to call the constructor
* Default
* No Args
* Parameterized
* Constructor overloaded
* Private constructor:
* Constructor chaining: this [within one class], super[ between parent and child class]
* Memory management in java:
* 
* 
* Types: both created by JVM and stored in RAM.

1. Stack

* Stores temporary variables and separate memory block for methods
* Primitive data types
* Stores reference of the heap objects [ strong reference, weak reference: soft reference(created by using WeakReferance class) ]
* Each thread has its own stack memory
* When stack memory goes full, its throws “java.lang.StackOverFlowError”
* Variables within a SCOPE is only visible and as soon as any variable goes out of the scope, it get deleted from the stack (in LIFO order)

1. Heap:

* Stores object
* There is no order of allocating the memory
* GC used to collect unreferenced objects from the heap
* Mark and sweep algo
* It’s shared with all the threads
* Contains string pool
* Heap full-> “java.lang.OutofMemoryError”
* Heap divided into:

Young Generation (minor GC happens here) : Eden, survivor

Old / Tenured GC ( major GC happens here)

Permanent generation

Meta space: non-heap, extendable, stores class variables, meta data, static or final variables, constant data

* GC is used to delete the unreferenced objects from the heap
* -mark and sweep algorithm
* -types of GC: Single GC, parallel GC, CMS( concurrent mark sweep), G1
* Heap shared with all the threads
* Contains the string pool
* Heap full then it throws “java.lang.OutofMemoryError”
* Heap memory is further divided into:
* -young Generation (minor GC happens here): Eden, survivor
* -Old/tenured Generation (major GC happens here)
* -permanent generation(meta space): separated form heap space
* Garbage collector:
* Types of GC: single GC: only 1 thread works, slow
* Parallel GC: default version in java 8, little bit fast
* CMS (concurrent mark sweep): runs concurrent gc but not 100%
* G1: 100% concurrent gc run
* Class Loading: process of the storing the class specific information in memory.
* JVM contains Class loaders: which is responsible for class loading.
* Class loaders are program that run inside JVM.
* Types:

1. Static: Compiler decides which class to load
2. Dynamic: Runtime developer told which class needs to load

* Whenever inheritance is done statically parent class is loading in memory
* During association also the associated class is being loaded in the memory
* Memory is allocated for static data-members in class area.
* Default values are assigned to static data-members.
* Static block is executed in given order.
* There are 3 types of class loaders which is responsible for loading class in Java

1. Bootstrap [ Primodial Classloader] : Not a Java class, called by JVM. It loads the .java classes dynamically from the location rt.jar [path:jdk/jre/lib/rt.jar]
2. Extension: Child of bootstrap, It loads the file from jre/lib/ext directory
3. Application: Loads the classes found in environment variable classpath

* Delegation Hierarchy Model: JVM and classloaders use an algorithm to called delegation model. JVM checks whether the class is loaded or not from method area. If .class file is not loaded, JVM request classloader subsystem to load it. Classloader forward this request application classloader-> It will delegate the request to bootstrap class loader.-> Bootstrap class loader search it into bootstrap classpath. [jdk/jre/lib/rt.jar] if not found the java class it will deligates the request to extension classloader.-> extension classloader search in its path jdk/jre/ext/\*.jar if not found here-> it will delegates the request to application classloaders. And if class file is not even there in application classloader. Runtime exception occur: ClassNotFoundException
* Types of classes:

1. Concrete class:

* those we can create an instance using NEW keyword
* all method should have implementation
* can also be your child class from interface or extend abstract class
* a class access modifier can be “public” or “package private” ( no explicit modifier defined)

1. Abstract class:

* Declared using: abstract” keyword,
* Can have both abstract (without body) and non-abstract methods’
* Can not create an instance of this class
* Child of abstract class can be abstract or concrete class
* Constructor can be created inside them and with super keyword from child classes we can access them.

1. Super class and sub class:

* A class that is derived from another class -> subclass
* From class through which subclass is derived it’s called -> superclass
* In java, in the absence of any other explicit superclass, every class is implicitly a subclass of Object class

1. Object class:
2. – Present in java.lang , topmost class in java [parent of all class], which has methods like clone(), toString(), equals(), notify(), wait(), hashCode(), finalize(), getClass(), notify(), notifyAll()
3. Nested class: Use when, if you know that, a class A will be used by only one another class B, then instead of created new file(a.java) for it, we can create nested class inside B class itself. Helps to group logically related classes in one file

Scope: same as of its outer class

* Inner class (non-static nested class): have access to all the instance variable and method of outer class, obj can be initiated on after initiating the object of outer class.

1. Anonymous inner class: Inner class without name, need when we want to override the behaviour of the method without even creating any subclass. End with (;)
2. Member inner class: can be private, public, protected, default
3. Local inner class: inside if, while or loop method declare, only can be default not private, public, protected

* Static nested class/ static class:

Do not have access to the non-static instance var of outer class method

Can be initiated without initiating the object of outer class

Can be private, public protected or package-private(default, no explicit declaration)

1. Generic class:

* Used to handle Object class related type casting
* Generic type can be any non-primitive object . eg. Print<String> vehicleobj= new Print<String>();
* Apart primitive type we can use anything as generic type
* Can have as many as generic type parameters. Eg class Pair<K,V,….,N>
* Inheritance with generic class possible
* Generic Method: type parameter should be before the return type of the method declaration, type parameter scope is limited to method only. Eg. public <K,V> void printValue( Pair<K,V> p1, Pair<K,V> p2)
* Raw Type: name of the generic class or interface without any type argument
* Bounded Generics : used at generic class and method: upper bound (<T extends Number>) means T can be of type Number or its subclass only. Here superclass ( in this example Number) we can have interface too., Multibound: <T extends superclass & Interface1 & interface N> , The 1st restrictive type should be concrete class. 2,3 and so on… can be interfaces.
* WildCards: upper bounded wildcard : <? Extends UpperBoundClassName> i.e. class Name and below, Lower bounded wildcard: <? Super LowerBoundClassName> i.e. class Name and above, Unbounded wildcard <?> only you can read
* Type Erasure

1. POJO class: Plain old java object

* Contains variables and its getter and setter methods
* Class should be public
* Public default constructor
* No annotations should be used like @Table, @Entity, @ID
* It should not extend any class or implement any interface

1. Enum Class :

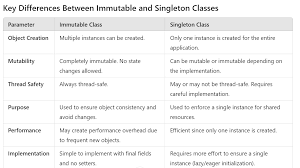
* Collection of CONSTANTS
* Created using static and final
* Internally extends java.lang.Enum class
* Implements interfaces
* Can have variables, constructors, method
* Can not be initiated( because it private , default- bytecode make it private)
* Can not be extended by other class
* Can have abstract method, and all the constants should implement that abstract method
* Can be define with custom values, method overriding by constant, with abstract method, enum implements interface

1. Final class: class declared as final cannot be inherited.
2. Singleton class: objective is to create only one and one object

* Different ways of creating Singleton class:

1. Eager Initialization
2. Lazy Initialization
3. Synchronization block
4. Double check lock (there is a memory issue, resolved through volatile instance variable)
5. Bill pugh solution
6. Enum singleton
7. Immutable class:

* We can not change the value of an object once it is created
* Declare class as ‘final’ so that it can not be extended
* All class members should be private. So that direct access can be avoided
* Class members are initialized only once using constructor
* There should not be any setter methods , which is generally use to change the value
* Just getter methods. And returns copy of the memory variable
* Eg. String, Wrapper class



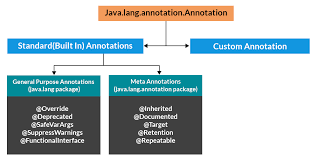
1. Reference Data type(Non-premitive) : in java everything is pass by value there is no concept pass by reference in java.

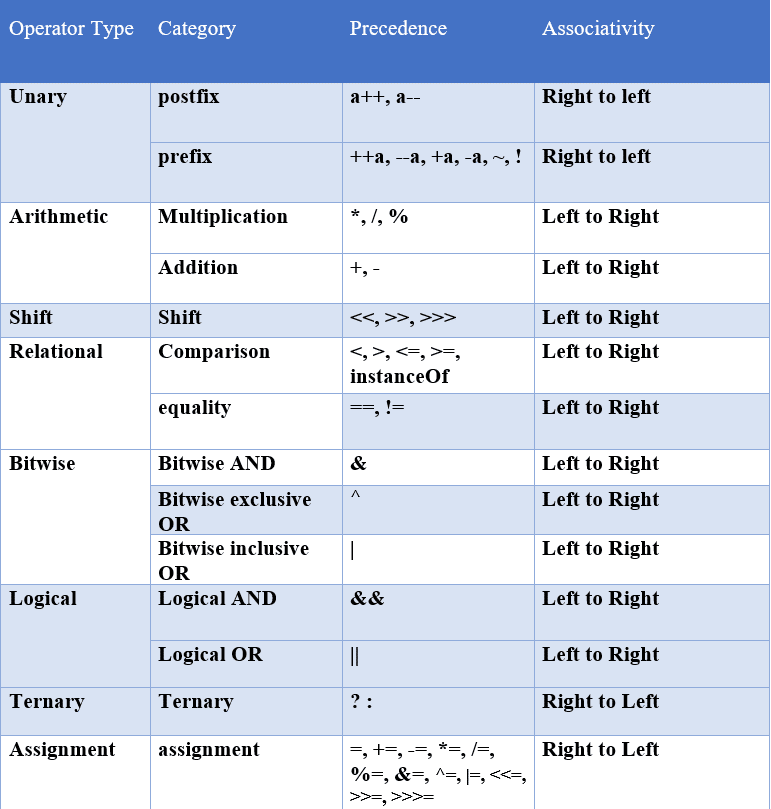
* Class: obj reference stored inside heap memory
* String: Immutable, String Literals Inside heap memory -> String constant pool: stores literals, using new it will create new reference inside heap not inside string constant pool, equals() checks value, == checks reference/address
* Interface: can’t create obj of ineterface

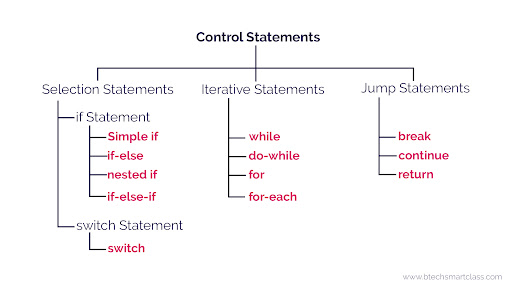
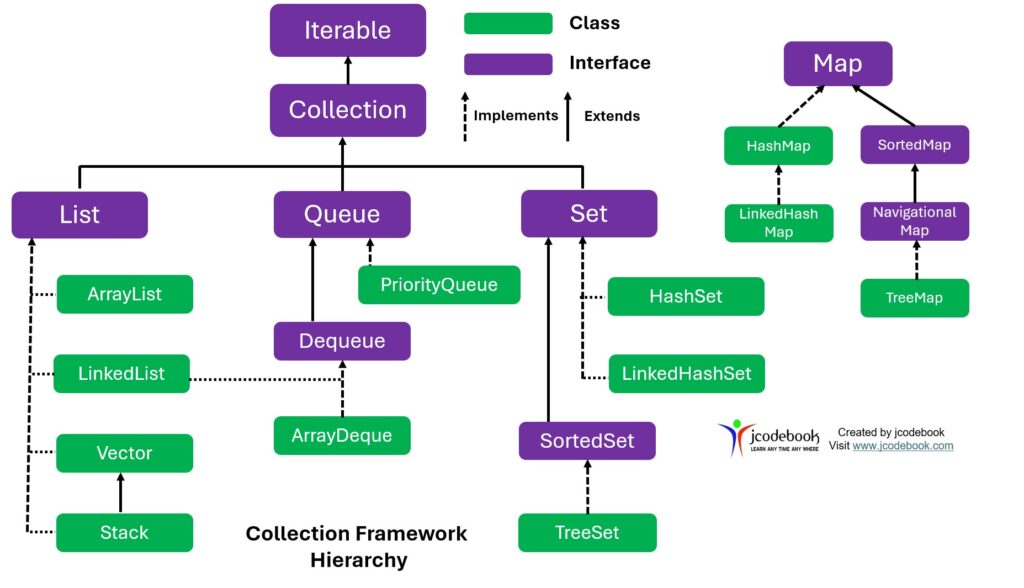
1. Wrapper class: Primitive data types not stored in heap stored in stack(it works top to bottom so once method execution over it will wannish output which is store on topn and it will return old value) so we need wrapper class to modify value because it uses heap because its reference type, Collections works on Wrapper class/reference type hence for each primitive type wrapper class is provided by java

* Autoboxing: primitive to wrapper class
* Unboxing: Wrapper class to primitive type
* Interface: it helps two system to interact with each other, without one system has to know the details of other
* Helps to achieve Abstraction
* Only public and default modifiers are allowed( Protected and private are not allowed)
* Need of Interface: Abstraction, Polymorphism, multiple inheritance [ solves diamond problem]
* Can not create object of interface
* All method inside interface should be implicit public only, can not be declared as final
* Fields in Interface: are public, static and final implicitly (Constant), can not make field private or protected
* Nested Interface: can be declared within another interface, can be declared within a class, Used to group, logical related interface
* Java 8 features : Default method [Before java 8 m interface can have only abstract method . and all child class has to provide its implementation, so default method introduced 3for when new requirement comes then we can provide this method inside interface with bodyonly need to provide implementation in child class if multiple interface are implemented and they have same default class] eg. stream in collection interface , Static method” [ we can provide implementation of the method in interface, but it can not be overridden by classes which implement the interface, can access it using interface name itself, by default public]
* Java 9 features: Private method , Private static method: To avoid redundant code and more re-usability, Oracle Corp. introduced private methods in Java SE 9 Interfaces. From Java SE 9 on-wards, we can write private and private static methods too in an interface using ‘private’ keyword.
* Functional Interface:
* If an interface contains only one abstract method that is known as Functional Interface
* Also known as SAM interface ( Single Abstract Method)
* @FunctionalInterface keyword can be used at top of the interface ( but its optional), it restrict us to create only one abstract method
* Can contain default , static and object class method along with abstract method
* Can be implemented using “implements” keyword and using “anonymous class”, using “lambda Expression”
* Types Of Functional Interface:

1. Consumer: Represents an operation, that accept a single input parameter and returns no result, present inside java.util.function;
2. Supplier: Represent the supplier of the result. Accepts no input parameter but produce a result. Present in package : java.util.function;
3. Function: Represent function , that accepts one argument process it and produce a result, [ java.util.function]
4. Predicate: represent function that accept one argument and return the Boolean, [java.util.function]

* Lambda Expression: is a way to implement the functional interface [ “()->” ]
* Java Reflection: used to examine the classes, methods, fields , interfaces at runtime and also possible to change the behaviour of the class too. [java,lang.reflect;], used to details of metadata
* Annotaions: kind of adding meta data to the java code.
* Can use use this meta data information at runtime and can add certain logic in our code if wanted. Can be applied to class, methods , interfaces , fields , parameters
* Types: pre defined [ 1. Meta data annotations, 2. Used on java code] , custom annotations/user defined
* 
* Operators in java:



* Control flow statements
* 
* Collection framework
* 
* **Java Collection Framework - Short Notes**

| **Aspect** | **Details** |
| --- | --- |
| **Package** | java.util |
| **Definition** | The **Java Collection Framework (JCF)** is a unified architecture for handling and manipulating groups of objects efficiently. |
| **Purpose** | Provides reusable data structures and algorithms to store, manipulate, and process collections of objects. |
| **Core Interfaces** | 1. Collection - Root interface of collection framework 2. List - Ordered collection (e.g., ArrayList, LinkedList) 3. Set - Unique elements only (e.g., HashSet, TreeSet) 4. Queue - FIFO structure (e.g., PriorityQueue, Deque) 5. Map - Key-value pairs (e.g., HashMap, TreeMap) |
| **Hierarchy** | Collection → List, Set, Queue Map is a separate interface, not extending Collection. |
| **Key Implementations** | **List:** ArrayList, LinkedList, Vector, Stack **Set:** HashSet, LinkedHashSet, TreeSet **Queue:** PriorityQueue, Deque, ArrayDeque **Map:** HashMap, LinkedHashMap, TreeMap, Hashtable, ConcurrentHashMap |
| **Ordering** | **List:** Maintains insertion order **Set:** Unordered (HashSet), Sorted (TreeSet) **Queue:** Ordered based on priority (PriorityQueue) **Map:** Unordered (HashMap), Sorted (TreeMap) |
| **Thread Safety** | **Non-Synchronized:** ArrayList, HashMap, HashSet **Synchronized:** Vector, Hashtable, Collections.synchronizedList() |
| **Java 8 Enhancements** | 1. **forEach()** - Iterates using lambda expressions. 2. **stream() API** - Performs operations like filter, map, reduce. 3. **removeIf()** - Removes elements based on a condition. |
| **Example Usage** (ArrayList) | java <br> List<String> list = new ArrayList<>(); <br> list.add("A"); list.add("B"); <br> list.forEach(System.out::println); |
| **When to Use?** | **List:** When order matters & duplicates are allowed. **Set:** When uniqueness is required. **Queue:** When FIFO processing is needed. **Map:** When key-value association is needed. |

* **Iterable Interface in Java (Including Java 8 Features)**

| **Aspect** | **Details** |
| --- | --- |
| **Package** | java.lang |
| **Definition** | Iterable<T> is an interface that allows a collection to be iterated sequentially. |
| **Purpose** | Provides a way to iterate over a collection using an Iterator, enhanced for-loop, or forEach(). |
| **Implemented By** | All major collection classes like ArrayList, HashSet, LinkedList, etc. |
| **Key Methods** | 1. Iterator<T> iterator() - Returns an iterator over elements. 2. void forEach(Consumer<? super T> action) *(Java 8)* - Iterates using lambda expressions. 3. Spliterator<T> spliterator() *(Java 8)* - Supports parallel iteration using Spliterator. |
| **Java 8 Enhancements** | **1. forEach(Consumer<? super T> action):** Allows iteration using lambda expressions or method references. **2. spliterator():** Enables parallel stream processing and performance optimization. |
| **Usage - forEach() (Java 8)** | java <br> List<String> list = Arrays.asList("A", "B", "C"); <br> list.forEach(System.out::println); |
| **Usage - spliterator() (Java 8)** | java <br> Spliterator<String> spltr = list.spliterator(); <br> spltr.forEachRemaining(System.out::println); |
| **Custom Implementation** | A class can implement Iterable<T> to define custom iteration logic. |
| **Introduced In** | Java 5 |
| **Enhancements** | Java 8 introduced forEach() and spliterator() for better iteration support. |

* Comparable vs comparator

| **Feature** | **Comparable<T>** | **Comparator<T>** |
| --- | --- | --- |
| **Package** | java.lang | java.util |
| **Purpose** | Defines natural ordering of objects. | Defines custom ordering of objects. |
| **Implementation** | Implemented within the class (implements Comparable<T>). | Implemented as a separate class (implements Comparator<T>). |
| **Method to Implement** | int compareTo(T obj) | int compare(T obj1, T obj2) |
| **Comparison Logic** | Defined inside the entity class. | Defined outside the entity class. |
| **Used For** | Default sorting order. | Sorting based on multiple criteria. |
| **Modifies Original Class?** | Yes (must modify class to implement Comparable). | No (works externally without modifying the class). |
| **Example Use Case** | Sorting students by roll number (natural order). | Sorting students by name, age, or marks (custom order). |
| **Sorting Order** | Single sorting sequence (natural order). | Multiple sorting sequences possible. |

* **Map Interface** in Java:

| **Feature** | **Description** |
| --- | --- |
| **Definition** | A collection that maps **keys** to **values**, where each key is unique. |
| **Package** | java.util |
| **Key Characteristics** | - No duplicate keys allowed - Allows one null key (in HashMap, LinkedHashMap) - Multiple null values allowed |
| **Common Implementations** | HashMap, LinkedHashMap, TreeMap, Hashtable, ConcurrentHashMap |
| **Important Methods** | - put(K key, V value): Adds key-value pair - get(Object key): Retrieves value by key - remove(Object key): Removes entry by key - containsKey(Object key): Checks if key exists - containsValue(Object value): Checks if value exists - keySet(): Returns a set of keys - values(): Returns a collection of values - entrySet(): Returns a set of key-value pairs |
| **Performance** | - HashMap: O(1) for get/put - TreeMap: O(log n) (Red-Black Tree) - LinkedHashMap: O(1), maintains insertion order |
| **Thread Safety** | - HashMap is **not** thread-safe - ConcurrentHashMap is thread-safe |
| **Sorting** | - TreeMap sorts by **natural order** or custom Comparator - LinkedHashMap maintains insertion order |
| **Best Use Cases** | - HashMap: Fast lookups, non-thread-safe scenarios - TreeMap: Sorted keys - LinkedHashMap: Maintain insertion order - ConcurrentHashMap: Multi-threaded environments |

* Steams:

Here’s a **comprehensive table** covering **Java Streams** with key details, categorized by **Creation, Intermediate Operations, and Terminal Operations**:

**Java Stream Notes (Table Format)**

| **Category** | **Method / Operation** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Stream Creation** | Stream.of(T... values) | Creates a stream from values. | Stream.of(1, 2, 3, 4, 5); |
|  | Arrays.stream(T[] array) | Creates a stream from an array. | Arrays.stream(new int[]{1, 2, 3}); |
|  | List.stream() | Creates a sequential stream from a collection. | list.stream(); |
|  | List.parallelStream() | Creates a parallel stream. | list.parallelStream(); |
|  | Stream.generate(Supplier<T>) | Generates an infinite stream. | Stream.generate(Math::random).limit(5); |
|  | Stream.iterate(T seed, UnaryOperator<T>) | Generates an infinite stream with a function. | Stream.iterate(1, n -> n + 2).limit(5); |
|  | Files.lines(Path path) | Reads file lines as a stream. | Files.lines(Paths.get("file.txt")); |
|  | Pattern.compile(",").splitAsStream(String) | Splits a string into a stream using regex. | Pattern.compile(",").splitAsStream("A,B,C"); |
|  | IntStream.range(int start, int end) | Creates a stream of numbers within a range. | IntStream.range(1, 5); // 1,2,3,4 |
|  | Stream.builder() | Creates a stream using a builder pattern. | Stream.<String>builder().add("A").add("B").build(); |
| **Intermediate Operations** | filter(Predicate<T>) | Filters elements based on a condition. | stream.filter(n -> n % 2 == 0); |
|  | map(Function<T,R>) | Transforms elements in the stream. | stream.map(n -> n \* 2); |
|  | flatMap(Function<T,Stream<R>>) | Flattens nested structures. | stream.flatMap(Collection::stream); |
|  | distinct() | Removes duplicate elements. | stream.distinct(); |
|  | sorted() | Sorts elements in natural order. | stream.sorted(); |
|  | sorted(Comparator<T>) | Sorts elements using a comparator. | stream.sorted(Comparator.reverseOrder()); |
|  | peek(Consumer<T>) | Performs an action without modifying the stream. | stream.peek(System.out::println); |
|  | limit(long n) | Limits the number of elements in the stream. | stream.limit(5); |
|  | skip(long n) | Skips the first n elements. | stream.skip(2); |
| **Terminal Operations** | forEach(Consumer<T>) | Iterates over each element. | stream.forEach(System.out::println); |
|  | collect(Collector<T,A,R>) | Collects stream elements into a collection. | stream.collect(Collectors.toList()); |
|  | toArray() | Converts stream elements into an array. | stream.toArray(); |
|  | reduce(BinaryOperator<T>) | Performs reduction (e.g., sum, min, max). | stream.reduce(0, Integer::sum); |
|  | count() | Counts the number of elements. | stream.count(); |
|  | min(Comparator<T>) | Finds the minimum element. | stream.min(Comparator.naturalOrder()); |
|  | max(Comparator<T>) | Finds the maximum element. | stream.max(Comparator.naturalOrder()); |
|  | anyMatch(Predicate<T>) | Returns true if any element matches the predicate. | stream.anyMatch(n -> n > 10); |
|  | allMatch(Predicate<T>) | Returns true if all elements match the predicate. | stream.allMatch(n -> n > 0); |
|  | noneMatch(Predicate<T>) | Returns true if no elements match the predicate. | stream.noneMatch(n -> n < 0); |
|  | findFirst() | Returns the first element (if present). | stream.findFirst(); |
|  | findAny() | Returns any element (useful in parallel streams). | stream.findAny(); |

* Multithreading

Here are **comprehensive notes on Multithreading in Java** in **table format**:

**Java Multithreading Notes (Table Format)**

**1. Basics of Multithreading**

| **Concept** | **Description** |
| --- | --- |
| **Thread** | A lightweight sub-process that runs independently within a Java program. |
| **Multithreading** | Running multiple threads concurrently to achieve parallel execution and improved performance. |
| **Process vs. Thread** | A **process** is an independent program execution, whereas a **thread** is a lightweight sub-task within a process. |
| **Thread Lifecycle** | **New → Runnable → Running → Blocked/Waiting → Terminated** |
| **Main Thread** | Every Java program starts with the main() method, which runs in the **main thread**. |

**2. Creating a Thread**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| **Extending Thread class** | Create a new thread by overriding run() method in Thread class. | class MyThread extends Thread { public void run() { System.out.println("Thread running"); } } |
| **Implementing Runnable interface** | Implement Runnable interface and define run(). | class MyRunnable implements Runnable { public void run() { System.out.println("Thread running"); } } |
| **Using ExecutorService (Thread Pool)** | Use Executors.newFixedThreadPool(n) for efficient thread management. | ExecutorService executor = Executors.newFixedThreadPool(2); executor.submit(new MyRunnable()); |
| **Using Callable and Future** | Allows threads to return results. | class MyCallable implements Callable<Integer> { public Integer call() { return 10; } } |

**3. Thread Lifecycle & States**

| **State** | **Description** |
| --- | --- |
| **New** | Thread is created but not started (new Thread()). |
| **Runnable** | Thread is ready to run but waiting for CPU. |
| **Running** | The thread is executing its run() method. |
| **Blocked** | Waiting for a locked resource. |
| **Waiting** | Waiting indefinitely for another thread's signal. |
| **Timed Waiting** | Waiting for a specific time (sleep() or join(timeout)). |
| **Terminated** | Thread has completed execution. |

**4. Thread Methods**

| **Method** | **Description** |
| --- | --- |
| start() | Starts a new thread (calls run() internally). |
| run() | Defines the thread's task (must be overridden). |
| sleep(ms) | Puts the thread to sleep for a specified time. |
| join() | Forces the current thread to wait until another thread finishes execution. |
| isAlive() | Checks if a thread is still running. |
| yield() | Hints the scheduler to switch execution to another thread. |
| setName(name) | Sets a thread's name. |
| getName() | Returns a thread's name. |
| setPriority(int) | Sets the thread priority (MIN\_PRIORITY = 1, MAX\_PRIORITY = 10, NORM\_PRIORITY = 5). |

**5. Thread Synchronization**

| **Concept** | **Description** | **Example** |
| --- | --- | --- |
| **Synchronized Block** | Allows only one thread to access a block of code at a time. | synchronized(this) { // Critical section } |
| **Synchronized Method** | Ensures only one thread can execute a method at a time. | synchronized void method() { } |
| **Static Synchronization** | Synchronizes methods at the class level. | synchronized static void method() { } |
| **Lock (ReentrantLock)** | Explicit lock mechanism with manual unlocking. | Lock lock = new ReentrantLock(); lock.lock(); try { } finally { lock.unlock(); } |
| **Volatile** | Ensures variable visibility across threads. | volatile int sharedVar; |
| **Atomic Variables** | Provides atomic operations to avoid synchronization overhead. | AtomicInteger count = new AtomicInteger(0); |

**6. Inter-Thread Communication**

| **Method** | **Description** |
| --- | --- |
| wait() | Causes a thread to wait until it is notified. |
| notify() | Wakes up a single waiting thread. |
| notifyAll() | Wakes up all waiting threads. |
| **Example** | synchronized(obj) { obj.wait(); } obj.notify(); |

**7. Deadlock & Avoidance**

| **Concept** | **Description** |
| --- | --- |
| **Deadlock** | When two or more threads are waiting for each other indefinitely. |
| **Avoidance** | Use proper **lock ordering**, **tryLock()**, or **timeouts**. |

**8. Thread Pooling**

| **Executor Service** | **Description** |
| --- | --- |
| newFixedThreadPool(n) | Fixed number of threads in a pool. |
| newCachedThreadPool() | Dynamic thread pool that creates new threads as needed. |
| newSingleThreadExecutor() | A single-threaded executor. |
| submit(Callable<T>) | Submits a task that returns a result. |
| shutdown() | Stops accepting new tasks. |

**9. Fork/Join Framework**

| **Concept** | **Description** |
| --- | --- |
| **ForkJoinPool** | A thread pool optimized for recursive parallelism. |
| **ForkJoinTask** | Splits tasks into smaller subtasks for parallel execution. |
| **Example** | ForkJoinPool pool = new ForkJoinPool(); pool.invoke(new RecursiveTask<Integer>() { ... }); |

**10. Concurrency Utilities (java.util.concurrent)**

| **Class / Interface** | **Description** |
| --- | --- |
| Semaphore | Controls thread access using permits. |
| CountDownLatch | Blocks threads until a countdown reaches zero. |
| CyclicBarrier | Synchronizes threads at a common barrier point. |
| ConcurrentHashMap | Thread-safe alternative to HashMap. |
| BlockingQueue | Thread-safe queue implementation. |

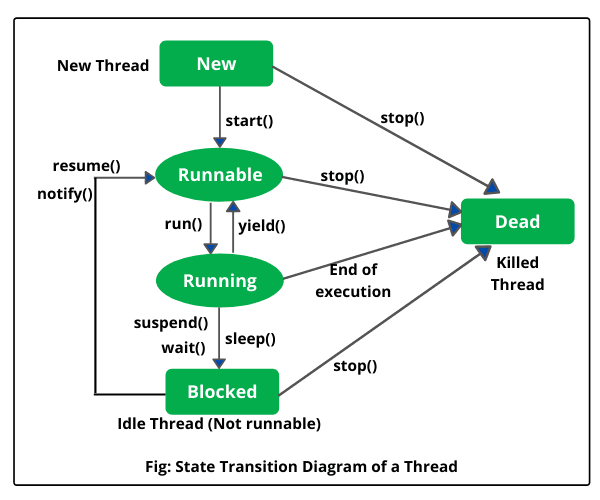
**11. Key Differences**

| **Feature** | **Thread Class** | **Runnable Interface** |
| --- | --- | --- |
| Inheritance | Extends Thread (cannot extend another class). | Implements Runnable (can extend other classes). |
| Reusability | Less flexible (tight coupling). | More flexible (better for thread pooling). |
| Recommended? | Not preferred. | Preferred in real-world applications. |

**12. Real-World Use Cases**

| **Use Case** | **Why Use Multithreading?** |
| --- | --- |
| **Web Servers** | Handle multiple client requests simultaneously. |
| **Background Tasks** | Run tasks like logging, data processing, or monitoring. |
| **Parallel Processing** | Perform large computations using multi-core processors. |
| **GUI Applications** | Prevent UI freezing by running background operations in separate threads. |

* Thread Life cycle



* Concurrent collection, lock, atomic operations, CMS, thread pool & lifecycle