**Basic of Java**

JVM is abstract virtual machine. It is virtual because it does not exist physically. It is platform dependent. **It provides runtime environment** to execute java Bytecode. ***Performs tasks: Loads code, verify code, execute code, provide runtime environment.***

JRE is Java runtime Environment also known as Java RTE. It is platform dependent. **It is used to provide Run time Environment.** It is a set of software tools which are used for developing java applications.

JDK is Java development kit. It physically exists. It is platform dependent. **It contains JRE + Development tools.** JDK ***contains JVM, other resources like interpreter/loader(java), compiler(javac), an archive(jar) and a document generator(javadoc) etc.*** For java application development.

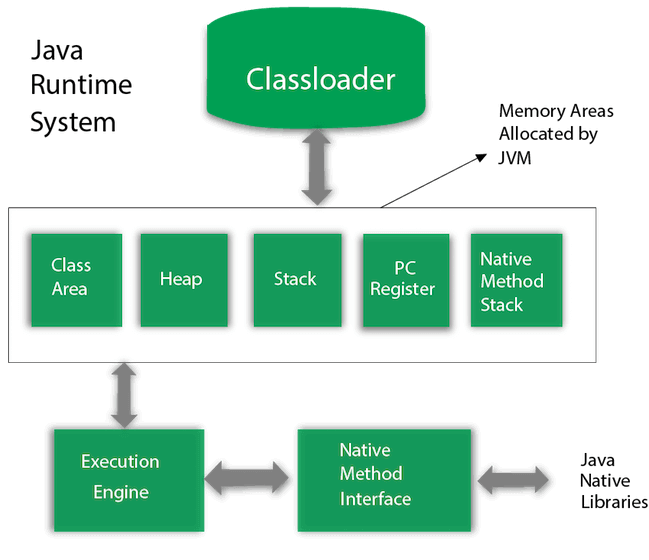
**JVM Architecture**

JVM is:

* Specification where working of jvm is specified. But implementation provider is independent to choose the algorithm.
* Implementation is known as JRE.
* Runtime Instance, when you write java command to CMD to run java class, an Instance of JVM is created.

It provides Definitions for:

* Memory Area
* Class file format
* Register set
* Garbage collected heap
* Fatal error reporting



1. Classloader: Used **to load the class files**. There are three classloaders in Java.

* Bootstrap Loader: super class of Extension classloader.
* Extension classloader: child class of Bootstrap classloader.
* System/Application classloader: child class of Extension classloader.

1. Class(method) area: **stores per class structure.**
2. Heap: run time **data area in which objects are allocated.**
3. Stack: stores frame. Holds local variables and partial results. Plays part in method invocation and return. **Frame is created when method invoked and destroy frame when invocation is done.**
4. Program Counter register: PC Register **contain address of jvm instruction** currently being executed.
5. Native method stack: contains all methods used in application.
6. Execution engine: It contains

* Virtual processor
* Interpreter: **read bytecode** and **execute instruction**
* Just-In-Time (JIT) compiler: To **improve the performance**. Compiles parts of byte code which have similar functionalities.

1. Java native Interface (JNI): framework used to communicate with app written in another language like c, c++, assembly.

**JIT Compiler**

To **improve the performance**. Compiles parts of byte code which have similar functionalities.

**OOPS Concepts**

**Encapsulation:** Bundling Variables and methods into a single class. Helps in **Data hiding**. Encapsulation itself is not Data hiding but it is used to achieve Data hiding.

**Inheritance:** Allows to create new class (Sub-class/ Child class) from existing class (Super-class/ Parent class). Use **extends** keyword in java to inherit a class.

**Method Overriding:** Overriding a method of super-class in sub-class. Both methods have **same method name**, the **same return type** and the s**ame parameter list. Static** and **Final** methods can not be override. **Run Time Polymorphism.**

**Polymorphism:** Means **more than one form**. One method can perform different tasks in different scenarios.

**Method Overloading:** method name same but **different no. of parameters** of **different type of parameters** or **both**. It is not method overloading if we only change the return type of methods. There must be differences in the number of parameters. **Compile Time Polymorphism.**

**Constructor Overloading:** If we have defined any parameterized constructor, then compiler will not create default constructor. and vice versa if we don’t define any constructor, the compiler creates the default constructor (also known as no-arg constructor) by default during compilation. **Recursive constructor calling is invalid in java.**

**Abstraction:** deals with **hiding the internal details** and showing the essential things to the user. Two ways to achieve abstracion: 1. Abstract Class or 2. Interface.

Interface **can only have abstract methods**(methods without body). **Exceptions are Static and Default methods.**

**Memory Management in Java**

Memory is divided into two parts: the **Stack** and the **Heap**.

**Stack:** responsible for holding references to the heap objects and stores value types(Primitive Types). Variables in the stack have certain visibility known as **Scope.**

**Heap:** Stores the actual object in the memory. Exists only one heap memory for each JVM process.

The Maximum size of stack and heap are not predefined, it depends on the running machine.

**Reference types:**

**1. Strong reference**: Object on the heap it is not garbage collected while there is a strong reference pointing to it, or if it is strongly reachable through a chain of strong references.

**2. Weak reference:** A weak reference to an object from the heap is most likely to not survive after the next garbage collection process.

**3. Soft reference:** Used for more memory-sensitive scenarios, since those are going to be garbage collected only when your application is running low on memory. Therefore, as long as there is no critical need to free up some space, the garbage collector will not touch softly reachable objects.

**4. Phantom reference:**

**Packages in Java**

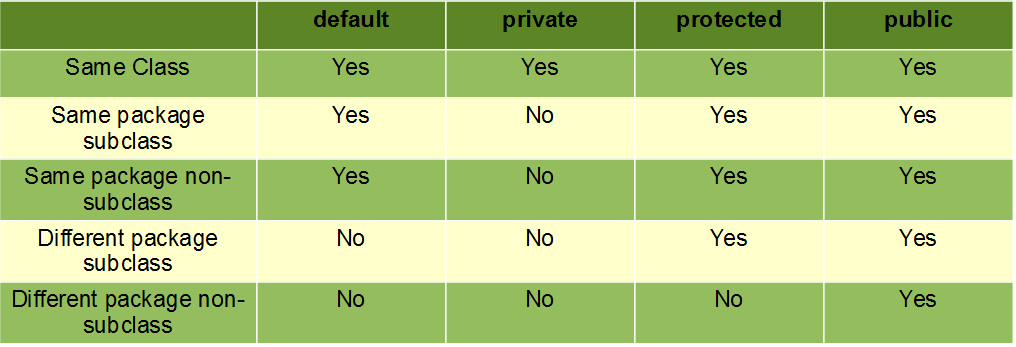
Packages in java are a mechanism to encapsulate a group of classes, sub-packages and interfaces.

Sub-packages are packages which are inside another package.

There are two types of packages:

1. Build-in packages: Consists large no. of classes which are part of Java API.
2. User defined packages: Packages which are created and defined by user.

**Access Modifiers in Java**



**If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.**

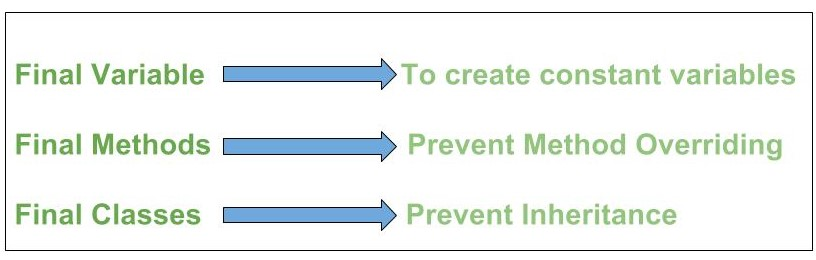
**Ex. If method was protected then you can’t use default in overriding, you can use public.**

**Overriding in Java**

1. The access modifier for an overriding method can allow more, but not less, access than the overridden method.
2. If we don’t want a method to be overridden, we declare it as final. **Final methods can not be overridden.**
3. When you define a static method with same signature as a static method in base class, it is known as method hiding. *When a static method of the superclass is hidden, it requires the subclass to use a fully qualified class name of the superclass to invoke the hidden method.*
4. **Private methods cannot be overridden** as they are bonded during compile time.
5. **The overriding method must have same return type (or subtype)** : From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. This phenomena is known as **covariant return type.**
6. We can call parent class method in overriding method using **super keyword.**
7. We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
8. **Overriding and Exception-Handling:**
   1. If the *super-class overridden method does not throw an exception,* *subclass overriding method can only throws the unchecked exception,* throwing checked exception will lead to compile-time error.
   2. If the *super-class overridden method does throws an exception, subclass overriding method can only throw same,* subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is *no issue if subclass overridden method is not throwing any exception.*
9. **Abstract methods in an interface or abstract class are meant to be overridden** in derived concrete classes otherwise a compile-time error will be thrown.

**Final Keyword in Java**

final is a non-access modifier applicable only to a variable, a method or a class.



**Final Variables:** When a variable is declared with final keyword, its value **can’t be modified**, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e., *you can add or remove elements from the final array or final collection or appending a string.* It is good practice to represent final variables in all uppercase, using underscore to separate words.

A final variable **can be assigned value later**, but only once. **final with for-each statement is a legal statement.**

There are three ways to initialize a final variable:

1. You can initialize a final variable when it is declared.
2. A blank final variable can be initialized inside an instance-initializer block or inside the constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise, a compile-time error will be thrown.
3. A blank final static variable can be initialized inside a static block.

**Final Classes:** A final class *cannot be extended(inherited)*. The other use of final with classes is *to create an immutable class* like the predefined String class. One can not make a class immutable without making it final.

**Final Methods:** A final method cannot be overridden.

**Constructor can not be made final.**

**Static Keyword in Java**

The static keyword is non-access modifier in java applicable to Blocks, Variables, methods and classes. *When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.*

**Static Blocks:**  If you need to do the computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

**Static variables:** We can create static variables at the class level only. static block and static variables are executed in the order they are present in a program.

**Static methods:** They can **only directly call other static methods**. They can only directly access static data. They **cannot refer to this or super** in any way.

**Static class:** A class can be **made static only if it is a nested class**. We cannot declare a top-level class. Nested static class doesn’t need a reference of Outer class. A static class **cannot access non-static members of the Outer class**.

**Enum in Java**

representing a group of named constants in a programming language. we can also add variables, methods and constructors to it.

The main objective of enum is to define our own data types(Enumerated Data Types). *Enum declaration can be done outside a Class or inside a Class but not inside a Method.*

**Every enum is internally implemented by using Class.**

Every enum constant represents an object of type enum.

enum type can be passed as an argument to switch statement.

Every enum constant is **always implicitly public static final**. Since it is static, we can access it by using the enum Name. Since it is final, we can’t create child enums.

We can declare the main() method inside the enum. Hence we can invoke enum directly from the Command Prompt.

All enums **implicitly extend java.lang.Enum class**. As a class can only extend one parent in Java, so an enum cannot extend anything else. toString() method is overridden in java.lang.Enum class, which returns enum constant name.

enum can implement many interfaces.

**values() method** can be used to return all values present inside enum.

Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.

**valueOf() method** returns the enum constant of the specified string value, if exists.

enum can contain a constructor and it is executed separately for each enum constant at the time of enum class loading.

We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

enum can contain both concrete methods and abstract methods. If an enum class has an abstract method, then each instance of the enum class must implement it

**String, StringBuilder, StringBuffer**

The Java **String is immutable** which means it cannot be changed. Whenever we change any string, a new instance is created. For **mutable strings, you can use StringBuffer and StringBuilder classes.**

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable.

Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

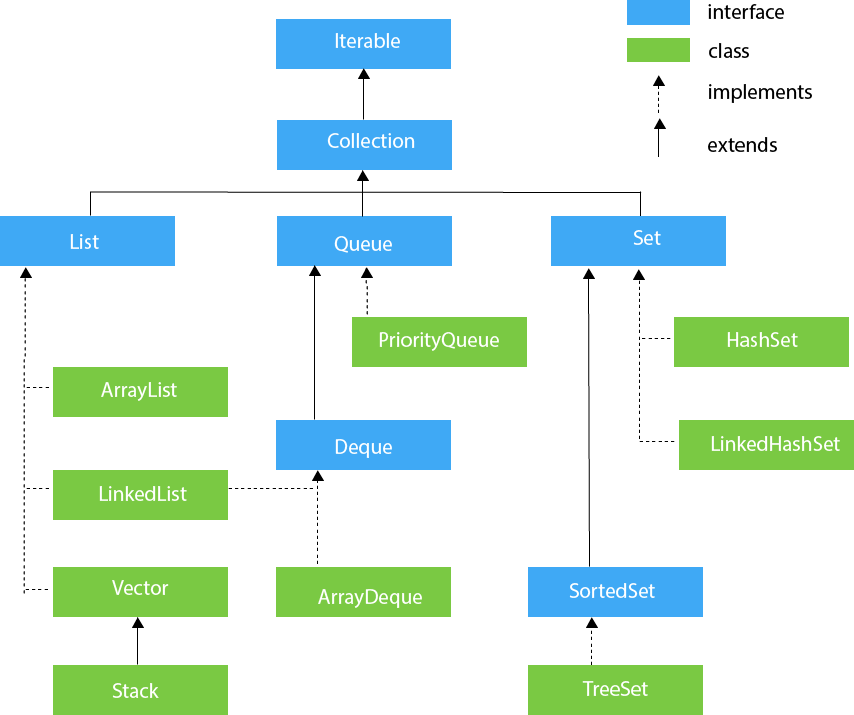
Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized (not Thread safe).

**Lambda Expression**

[Lambda Expressions](https://www.javatpoint.com/java-lambda-expressions)

**Collections in Java**

**A Collection represents a single unit of objects**, i.e., a group.



**Iterator interface:** Iterator interface provides the facility of iterating the elements in a forward direction only. Has Three methods *hasNext(), next() and remove().*

**Iterable interface:** The Iterable interface is *the root interface for all the collection* classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface. It contains only one abstract method.

**Collection interface:** The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have.

**List interface:** List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

* **ArrayList:** The ArrayList class implements the List interface. It uses a dynamic array to *store the duplicate element* of different data types. The ArrayList class maintains the insertion order and is *non-synchronized*. The elements stored in the ArrayList class can be randomly accessed.
* **LinkedList:** LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is *not synchronized*. In LinkedList, the manipulation is fast because no shifting is required.
* **Vector:** Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is ***synchronized*** *and contains many methods* that are not the part of Collection framework.
* **Stack:** The stack is the *subclass of Vector*. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

**Queue interface:** Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

* **Priority Queue:** The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue *doesn't allow null values* to be stored in the queue.
* **Deque interface:** Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.
* **ArrayDeque:** ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends. ArrayDeque is *faster than ArrayList and Stack* and has no capacity restrictions.

**Set interface:** It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can *store at most one null value* in Set.

* **HashSet:** It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It *contains unique items.*
* **LinkedHashSet:** LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and *permits null elements.*
* **SortedSet interface:** SortedSet is the alternate of Set interface that provides a total ordering on its elements. The *elements of the SortedSet are arranged in the increasing (ascending) order*. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.
* **TreeSet:** Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the *access and retrieval time of TreeSet is quite fast*. The elements in TreeSet stored in ascending order.

**Sorting in Java**

**Java Comparable interface:** Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains *only one method named compareTo(Object)*. It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. Need to *implement comparable interface* in class.

**Java Comparator interface:** A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of the same class.

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

**Java Streams**

[**Java Streams**](https://stackify.com/streams-guide-java-8/)

**Java IO**

[Java IO](https://www.javatpoint.com/java-io)

**Java Lang**

[Java Lang Package](https://www.geeksforgeeks.org/java-lang-package-java/)

**Java Reflection**

We can say its class Modifier. Mostly used for debugging tools. Can be used to get all information about a class.

[Java Reflection](https://www.geeksforgeeks.org/reflection-in-java/)

[Java Reflection Tutorial](https://www.youtube.com/watch?v=agnblS47F18&list=PLF206E906175C7E07&index=10)

**Java Multi Threading**

Allows concurrent execution of two or more parts of a program for *maximum utilization of CPU*. Each part of such program is called a thread. So, threads are **light-weight processes** within a process.

Threads can be created by using two mechanisms :

**1.Extending the Thread class:**

We create a class that extends the java.lang.Thread class. This class **overrides the run()** method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and **call start()** method to start the execution of a thread. **Start() invokes the run() method** on the Thread object.

**2.Implementing the Runnable Interface:**

We create a new class which implements java.lang.Runnable interface and **override run()** method. Then we instantiate a Thread object and **call start()** method on this object.

**Thread Class vs Runnable Interface**

1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, **if we implement the Runnable interface, our class can still extend other base classes.**
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

[Java Callable and Future](https://www.callicoder.com/java-callable-and-future-tutorial/)

[Java Callable and Future Tutorial](https://youtu.be/NEZ2ASoP_nY)

[Java Synchronized](https://www.youtube.com/watch?v=RH7G-N2pa8M)

**Java Concurrency**

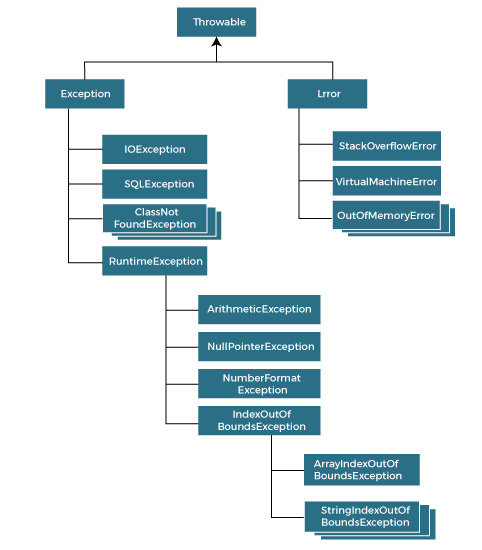
[Java Executors](https://www.baeldung.com/java-executor-service-tutorial)

[Java Scheduler](https://www.journaldev.com/2340/java-scheduler-scheduledexecutorservice-scheduledthreadpoolexecutor-example)

**Java Exceptions**

Exception Handling is a mechanism to handle runtime errors.

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



There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. **Checked Exception:** The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time. *By default Unchecked Exceptions are forwarded in calling chain (propagated).*
2. **Unchecked Exception:** The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime. *By default, Checked Exceptions are not forwarded in calling chain (propagated).*
3. **Error:** Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

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

**Java Object Cloning**

[Java Object Cloning](https://www.javatpoint.com/object-cloning)