**Important Points of HashMap:**

* HashMap has Key and Value pairs.
* HashMap is not Synchronized (i.e. Not Thread Safe)
* HashMap has no guarantees as to the order of the map. HashMap does not guarantee that the order will remain constant over time.
* HashMap allows only one Null Key, Which will always be stored at 0th position in the bucket
* HashMap can have multiple null values, but only one null key is allowed
* HashMap can be Synchronized externally using Collections.synchronizedMap(HashMap)

**Contract between equals() and hashCode() methods :**

* If two objects are equal by equals() method then their hash code values must be same.
* If two objects are not equal by equals() method then their hash code may be same or different.

**Employee emp1 = new Employee("One");**

**Employee emp2 = new Employee("One");**

**How it works without Overriding equals() and hashcode()?**

first it compares two objects (eventhough they are equals, since we have not overridden the equals()) it shows both the objects are not equals and will be considered as Unqiue Keys and with values.

From the hashing, it goes to different buckets – duplicate entries for same object.

****Override equals() and Not hashcode:**** Now after overriding only equals() method, we can see both objects are compared and identified as equals objects (Both Objects are  Equal: true)

But here the problem is, though both the objects are equal() the hashcode() is different and so the both objects will be stored in different buckets.

If both objects are equal and if it points to same bucket then the value will be overridden.

****Override hashcode() and Not equals():**** Since we have overridden only hashcode() and not equals method() –> The objects comparison becomes false, means the objects are unique. So even though the hashcode points to same bucket, the objects are considered as unique keys and both the values will be stored in the bucket.

****override both equals() and hashcode():**** Since we have overridden the equals() and hashcode(), when inserting in map — It has identified both objects are equal and both has same hashcode values. So when trying to insert into the bucket, only one value will be inserted that is the reason we have only one element in hashmap (Key is: Employee@13665 Value is: Two)

****Association, Composition and Aggregation in Java :****

****Association**** is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.

Example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

* It is a special form of Association where:
* It represents**Has-A** relationship.
* It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents part-of relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object cannot exist without the other entity.

**Aggregation vs Composition**

* **Dependency:**Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don’t exist separate to a Human
* **Type of Relationship:** Aggregation relation is “has-a” and composition is “part-of” relation.
* **Type of association:**Composition is a strong Association whereas Aggregation is a weak Association.

**Is finally block always get executed in Java? :** Yes, the finally block is always get executed unless there is an abnormal program termination either resulting from a JVM crash or from a call to System.exit().

* A finally block is always get executed whether the exception has occurred or not.
* If an exception occurs like closing a file or DB connection, then the finally block is used to clean up the code.
* We cannot say the finally block is always executes because sometimes if any statement like System.exit() or some similar code is written into try block then program will automatically terminate and the finally block will not be executed in this case.
* A finally block will not execute due to other conditions like when JVM runs out of memory when our java process is killed forcefully from task manager or console when our machine shuts down due to power failure and deadlock condition in our try block.

try {

System.out.println("I am in try block");

System.exit(1);

} catch(Exception ex){

ex.printStackTrace();

} finally {

System.out.println("I am in finally block");

}

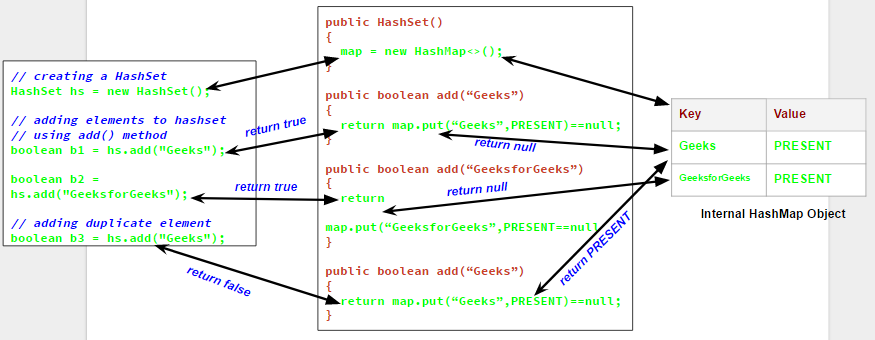
In the above example, the finally block will not execute due to the System.exit(1) condition in the try block.

**LinkedList class in Collection :**

* ***LinkedList***class extends ***AbstractSequentialList***and implements ***List***, ***Deque***and ***Queue*** interface.
* It can be used as ***List***, stack or ***Queue***as it implements all the related interfaces.
* It can contain duplicate elements and is not synchronized.
* maintains insertion order.
* not synchronized.
* No random access.
* manipulation fast because no shifting needs to be occurred.

**How LinkedList work Internally? :** Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the [doubly linked list data structure](https://www.geeksforgeeks.org/doubly-linked-list/). The main difference between a normal linked list and a doubly LinkedList is that a doubly linked list contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

**Internal working of Set/HashSet in Java : S**et is a well-defined collection of distinct objects. Each member of a set is called an element of the set. So in other words, we can say that a set will never contain duplicate elements. But how in java Set interface implemented classes like HashSet, LinkedHashSet, TreeSet etc. achieve this uniqueness.



// creating a HashSet

        HashSet hs = new HashSet();

        // adding elements to hashset

        // using add() method

        boolean b1 = hs.add("Geeks");

        boolean b2 = hs.add("GeeksforGeeks");

           // adding duplicate element

        boolean b3 = hs.add("Geeks");

**Output :** b1 = true;b2 = true;b3 = false;[GeeksforGeeks, Geeks]

Now from the output, it is clear that when we try to add a duplicate element to a set using add() method, it returns false, and element is not added to hashset, as it is already present. Now the question comes, how add() method checks whether the set already contains the specified element or not. It will be more clear if we have a closer look on the add() method and default constructor in HashSet class.

// predefined HashSet class

public class **HashSet**{

// A HashMap object

private transient HashMap<e, object="" style="box-sizing: border-box;"> map;

// A Dummy value(PRESENT) to associate with an Object in the Map

private static final Object PRESENT = new Object();

// default constructor of HashSet class

// It creates a HashMap by calling default constructor of HashMap class

**public HashSet() {**

**map = new HashMap<>();**

**}**

// add method it calls put() method on map object and

//then compares it's return value with null

**public boolean add(E e) {**

**return map.put(e, PRESENT)==null;**

**}**

// Other methods in Hash Set

}

Now as you can see that whenever we create a HashSet, it internally creates a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it’s key and constant Object called **“PRESENT”** as it’s value. So we can say that **a Set achieves uniqueness internally through HashMap**.

As we know in a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) each key is unique and when we call put(Key, Value) method, it returns the previous value associated with key, or null if there was no mapping for key. So in add() method we check the return value of map.put(key, value) method with null value.

1. If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and element is added to the HashSet(internally HashMap).
2. If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will return false and element is not added to the HashSet(internally HashMap).

As LinkedHashSet extends HashSet, so it internally calls constructors of HashSet using [super()](https://www.geeksforgeeks.org/super-keyword/). Similarly creating an object of [TreeSet](https://www.geeksforgeeks.org/treeset-class-java-examples/) class internally creates object of [Navigable Map](https://www.geeksforgeeks.org/navigablemap-interface-in-java-with-example/) as backing map.

**Difference between Singly linked list and Doubly linked list**

|  |  |
| --- | --- |
| **Singly linked list** | **Doubly linked list** |
| **Internal implementation :** A singly linked list is a set of nodes where each node has two fields ‘data’ and ‘link’. The ‘data’ field stores actual piece of information and ‘link’ field is used to point to next node. Basically ‘link’ field is nothing but address only. | DLL has nodes with a data field, a previous link field and a next link field.While doubly linked list has some more complex implementation where the node contains some data and a pointer to the next as well as the previous node in the list |
| SLL has nodes with only a data field and next link field. | DLL has nodes with a data field, a previous link field and a next link field. |
| **Order of elements :** In SLL, the traversal can be done using the next node link only.  Singly linked list allows traversal elements only in one way. | In DLL, the traversal can be done using the previous node link or the next node link.Doubly linked list allows element two way traversal. |
| **Memory consumption :** The SLL occupies less memory than DLL as it has only 2 fields. | The DLL occupies more memory than SLL as it has 3 fields. |
| Less efficient access to elements. | More efficient access to elements. |
| **Complexity :** In singly linked list the complexity of insertion and deletion at a known position is O(n) | In case od doubly linked list the complexity of insertion and deletion at a known position is O(1) |
| Singly linked list are generally used for implementation of stacks | On other hand doubly linked list can be used to implement stacks as well as heaps and binary trees. |
| Singly linked list is preferred when we need to save memory and searching is not required as pointer of single index is stored. | If we need better performance while searching and memory is not a limitation in this case doubly linked list is more preferred. |





**What is the garbage collector in Java? :**

* Garbage Collector is part of JRE that makes sure that object that are not referenced will be freed from memory.
* Garbage collector can be viewed as a reference count manager. if an object is created and its reference is stored in a variable, its reference count is increased by one. during the course of execution if that variable is assigned with NULL. reference count for that object is decremented. so the current reference count for the object is 0.
* Now when Garbage collector is executed, It checks for the objects with reference count 0. and frees the resources allocated to it.

**Advantage of Garbage Collection**

* It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
* It is automatically done by the garbage collector(a part of JVM) so we don’t need to make extra efforts.

**How can an object be unreferenced?**

There are many ways:

* **By nulling the reference**Student s=new Student();  
  s=null;
* **By assigning a reference to another**Student s1=new Student();  
  Student s2=new Student();  
  s1=s2;//now the first object referred by s1 is available for garbage collection
* **By annonymous object etc.**  
  new Student();

**finalize() method :**The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:  
protected void finalize(){}

The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

**gc() method :** The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

## **public static void gc(){}**

Many people think garbage collection collects and discards dead objects.  
In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage.

When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system. To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a mark-and-sweep algorithm.

**How work Garbage Collector in Java :**

**Java Memory Management :**Java Memory Management, with its built-in garbage collection, is one of the language’s finest achievements. It allows developers to create new objects without worrying explicitly about memory allocation and deallocation, because the garbage collector automatically reclaims memory for reuse. This enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. At least in theory.

Ironically, Java garbage collection seems to work too well, creating and removing too many objects. Most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system. In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

**Java Garbage Collection GC Initiation :**Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. System.gc() and Runtime.gc() are hooks to request the JVM to initiate the garbage collection process.

Though this request mechanism provides an opportunity for the programmer to initiate the process but the onus is on the JVM. It can choose to reject the request and so it is not guaranteed that these calls will do the garbage collection. This decision is taken by the JVM based on the eden space availability in heap memory. The JVM specification leaves this choice to the implementation and so these details are implementation specific.

Undoubtedly we know that the garbage collection process cannot be forced. I just found out a scenario when invoking System.gc() makes sense. Just go through this article to know about this corner case when System.gc() invocation is applicable.

**Java Garbage Collection Process :** Garbage collection is the process of reclaiming the unused memory space and making it available for the future instances.

**Eden Space:** When an instance is created, it is first stored in the eden space in young generation of heap memory area.

**Survivor Space (S0 and S1):**As part of the minor garbage collection cycle, objects that are live (which is still referenced) are moved to survivor space S0 from eden space. Similarly the garbage collector scans S0 and moves the live instances to S1.

Instances that are not live (dereferenced) are marked for garbage collection. Depending on the garbage collector (there are four types of garbage collectors available and we will see about them in the next tutorial) chosen either the marked instances will be removed from memory on the go or the eviction process will be done in a separate process.

**Old Generation:**Old or tenured generation is the second logical part of the heap memory. When the garbage collector does the minor GC cycle, instances that are still live in the S1 survivor space will be promoted to the old generation. Objects that are dereferenced in the S1 space is marked for eviction.

**Major GC:**Old generation is the last phase in the instance life cycle with respect to the Java garbage collection process. Major GC is the garbage collection process that scans the old generation part of the heap memory. If instances are dereferenced, then they are marked for eviction and if not they just continue to stay in the old generation.

**Memory Fragmentation:**Once the instances are deleted from the heap memory the location becomes empty and becomes available for future allocation of live instances. These empty spaces will be fragmented across the memory area. For quicker allocation of the instance it should be defragmented. Based on the choice of the garbage collector, the reclaimed memory area will either be compacted on the go or will be done in a separate pass of the GC.

**HashMap Interview Questions**

**What is Hashmap?**Hashmaps store items in key-value pairs. It is part of the Java collection. It extends AbstractMap class and implements the map interface and is found in java.util package. The most important condition of the hashmap is that the key-value pair should be unique. If there are any duplicates found in the key then the latest value will overwrite the previous value. You can perform operations like insertion, deletion, updation, etc. in hashmaps.

**How does it work in Java?**

Here are some points to remember for Hashmaps in Java:

* It does not maintain any order.
* It is non-synchronized.
* It may contain one or more null values.
* Initially, the default capacity of the Java Hashmap class is 16.

The operation of storing and retrieving the object is done at a constant time, provided that the key is known. The method to store the object is put(key, value) and the method to retrieve the object is get(key).

There are following parameters in Java Hashmap:  
K: the type of keys maintained by the map  
V: the type of values mapped.

****public Hashmap()****: This is the default constructor which creates an instance for Hashmap with an initial capacity of 16.

**public Hashmap(int initialCapacity, float loadFactor):**

This constructor creates an instance with the specified initial capacity and the load factor.

**What do you mean by load factor?**

It is the measure of how much rehashing is to be done. It is initially kept higher so rehashing doesn’t take place, but this also increases the iteration time. The most common load factor value is 0.75.

**Rehashing**is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size hashmap when the threshold is reached/crossed. Rehashing of a hash map is done when the number of elements in the map reaches the maximum threshold value

The **Load factor** is a measure that decides when to increase the HashMap capacity to maintain the get() and put() operation complexity of O(1). The default load factor of HashMap is 0.75f (75% of the map size).

**Is it possible to use any custom object as a key in Hashmap?**

Yes, we can use the custom object as a key in a hashmap by implementing hashcode() and equals() in the custom class. But the condition is that the hashcode should not vary once the object is inserted in the map.

**Hashmap handles collisions in Java. Justify.**

In the worst-case scenario, it can happen that all keys might have the same hashcode, which will result in the hash table turning into a linked list. In this case, searching a value will take O(n) complexity as opposed to O(1) time due to the nature of the linked list.

if the new values with the same key are attempted to be pushed, then these values are stored in a linked list stored in a bucket of the key as a chain along with the existing value.

**What will happen if two different HashMap key objects have the same hashcode?**

They will be stored in the same bucket but no next node of the linked list. And keys equals () method will be used to identify the correct key-value pair in HashMap.

**What are things an object needs to be used as a key or value in the Hashmap?**

The key and value along with its implementation should have the two functions hashcode() and equals(). The function that has the name of hashcode() is used when we insert the value of the key in any HashMap. At the same time, the function of equals() is called only when we are trying to get back the value that was already stored in the HashMap.

**Can we use Hashmaps in the case where we need to store null values?**

Yes, you can use the hashmaps to store null values without any issues. You can store one or multiple null values as per the requirement.

**How can you handle null keys in hashmaps?**

There are two separate methods for that putForNullKey(V value) and getForNullKey(). Null keys always map to index 0. The equals() and hashcode() methods are not used in the case of null keys in HashMap.

**Is it possible to store multiple values under the same key using Java Hashmaps?**

No, you will not be able to store duplicate keys in the hashmap. If you try to store a new value in a key already present in the hashmap, then the hashmap would simply remove the value that was earlier stored in that key and replace it with the new one.  
The size of the hashmap, in this case, would not change, meaning there will be no addition of keys into the hashmap. This feature is one of the reasons we use the function keyset() to get back all the keys of a hashmap and that this function returns a set and not a collection (because in a set all the values have to be unique).

**What are Instance initialization block & Static initialization block in java**

**Instance initialization block** in java can be used to initialize instance variables in java.   
**[Static](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) initialization block** in java can be used to initialize static variables in java.

**Features of static initialization block in java >**

* Static blocks are also called Static initialization blocks in java.
* Static block executes when class is loaded in java.
* static blocks executes before instance blocks in java.
* Only static variables can be accessed inside static block in java
* static blocks can be used for initializing static variables or calling any static method in java.
* this **[keyword](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html)** cannot be used in static block in java.

**Features of instance initialization block in java >**

* Instance blocks are also called instance initialization blocks in java
* Instance block executes when instance of class is created in java.
* Also known as non-static initialization block in java.
* instance blocks executes after static blocks in java.
* Static and non-static variables (instance variables) can be accessed inside instance block in java.
* instance blocks can be used for initializing instance variables or calling any instance method in java.
* this **[keyword](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html)** can be used in instance block in java.

**class** MyClass {

      /\* Static block \*/

**static** {

          System.*out*.println("static block");

   }

**/\* Non-Static block (Instance block) \*/**

**{**

**System.*out*.println("instance/non-static block");**

**}**

   /\* Constructor \*/

   MyClass() {

          System.*out*.println("MyClass constructor");

   }

}

/\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/

**public** **class** BlockTest {

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

          //Create instance of MyClass.

**new** MyClass();

   }

}

/\*OUTPUT

static block

**instance/non-static block**

MyClass constructor

\*/

When you using super & sub class

SuperClass ----------> static block

SubClass > static block

**SuperClass ----------> Instance/non-static block**

SuperClass ----------> constructor

**SubClass > Instance/non-static block**

SubClass > constructor

<https://howtodoinjava.com/java/collections/hashmap/design-good-key-for-hashmap/>

**Serialization** is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when **needed**. The reverse process is called deserialization.

**What type of members are not serialized**? Explanation: All static and transient variables are **not serialized.**

**Can we serialize static fields? :** In Java, **serialization** is a concept using which **we can** write the state of an object into a byte stream so that **we can** transfer it over the network (using technologies like JPA and RMI). But, **static variables** belong to class therefore, you cannot **serialize static variables** in Java.

**What is the use of @transient? :** The **transient** keyword in **Java** is used to avoid serialization. If any object of a data structure is defined as a transient , then it will not be serialized. Serialization is the ​process of converting an object into a byte stream.

At the time of serialization, if we don't want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient** keyword, it ignores original value of the variable and save default value of that variable data type.

**What is volatile and transient in Java? :** The **volatile** keyword flushes the changes directly to the main memory instead of the CPU cache. ... On the other hand, the **transient** keyword is used during serialization. Fields that are marked as **transient** can not be part of the serialization and deserialization.

**Why does Java have transient fields? : Transient** in **Java is** used to mark the member variable not to be serialized when it **is** persisted to streams of bytes. This keyword plays an important role to meet security constraints in **Java**. It ignores the original value of a variable and saves the default value of that variable data type.

**Why is serialization required? : Serialization** is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when **needed**. The reverse process is called deserialization.

**How do you serialize an object? :** To **serialize an object** means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the **object**. A Java **object** is **serializable** if its class or any of its superclasses implements either the java. io. **Serializable** interface or its subinterface, java.

**Are methods serialized in Java? :** Since **Method** does not implement **Serializable**, it cannot be **serialized** using the standard **Java Serialization** API. A workaround would be to manually **serialize** just the name of the class and **method** and its parameter types. You can then recreate the **Method** instance during deserialization.

**What serialVersionUID should I use? :** During serialization, java runtime associates a version number with each serializable class. This number called serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

**Why so we use SerialVersionUID :**SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded.

**Is serialization necessary? :** Serialization is usually used When the **need** arises to send your data over network or stored in files. By data I mean objects and not text. ... **Serialization** is the translation of your Java object's values/states to bytes to send it over network or save it.

**Serialization :** At the time of serialization, with every object sender side JVM will save a Unique Identifier. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system.

**Deserialization:** At the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise we will get Runtime Exception saying InvalidClassException. This unique Identifier is nothing but SerialVersionUID.

**Problem of depending on default SerialVersionUID generated by JVM :**

* Both sender and receiver should use the same JVM with respect to platform and version also. Otherwise receiver unable to deserialize because of different SerialVersionUID.
* Both sender and receiver should use same .class file version. After serialization if there is any change in .class file at receiver side then receiver unable to deserialize.
* To generate SerialVersionUID internally JVM may use complex algorithm which may create performance problem.

We can solve the above problem by configuring our own SerialVersionUID. We can configure our own SerialVersionUID as follows:

private static final long SerialVersionUID=10l;

**JDK :**

* JDK contains tools required to write Java programs, and JRE to execute them.
* It contains development tools such as a compiler, debugger, Java application launcher etc .
* Compiler converts code written in Java into byte code.
* Java application launcher opens a JRE, loads the necessary class, and executes its main method.
* JDK includes all features that JRE has.
* JDK provides the environment to develop and execute Java source code.
* It can be installed on Windows, Unix, and Mac operating systems.

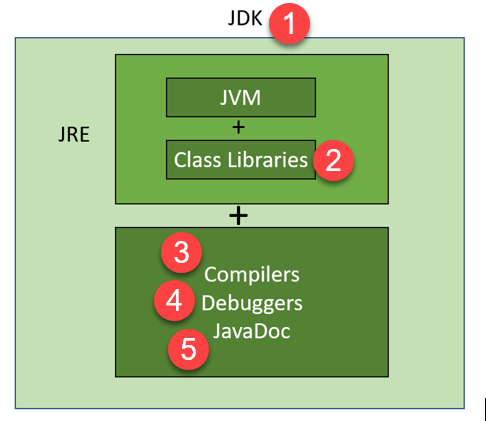
**JRE :**

* JRE contains class libraries, JVM, and other supporting files. It does not contain any tool for Java development like a debugger, compiler, etc.
* Java Runtime Environment is a set of tools using which the JVM actually runs.
* JRE contains deployment technology, including Java Web Start and Java Plug-in.
* Developers can easily run the source code in JRE, but Developers cannot write and compile the Java program.
* It includes integration libraries like Java Database Connectivity (JDBC), Remote Method Invocation (RMI), Java Naming and Directory Interface (JNDI), and more.
* JRE has JVM and Java HotSpot virtual machine client.

**JVM :**

* JVM provides a platform-independent way of executing Java source code.
* It has numerous libraries, tools, and frameworks.
* Once you run Java program, you can run on any platform and save lots of time.
* JVM comes with JIT(Just-in-Time) compiler that converts Java source code into low-level machine language. Hence, it runs more faster as a regular application.
* Java Virtual Machine converts byte code to the machine-specific code.
* It provides basic java functions like memory management, security, garbage collection, and more.
* JVM runs the program by using libraries and files given by Java Runtime Environment.
* JDK and JRE both contain Java Virtual Machine.
* It can execute the java program line by line hence it is also called as interpreter.
* JVM is easily customizable for example, you can allocate minimum and maximum memory to it.
* It is independent from hardware and the operating system. So, you can write a java program once and run anywhere.

**How JDK Functions? :**



Here are the important components of JDK:

* **JDK and JRE:** The JDK enables programmers to create core Java programs that can be run by the JRE, which included JVM and class libraries.
* **Class Libraries:**It is a group of dynamically loadable libraries that Java program can call at run time.
* **Compilers:** It is a Java program that accepts text file of developers and compiles into Java class file. It is the common form of output given by compiler, which contains Java byte code. In Java, the primary compiler is Javac.
* **Debuggers:** Debugger is a Java program that lets developers test and debug Java programs.
* **JavaDoc:**JavaDoc is documentation made by Sun Microsystems for the Java. JavaDoc can be used generating API documentation in HTML file from the source program.

**How JRE Functions?**

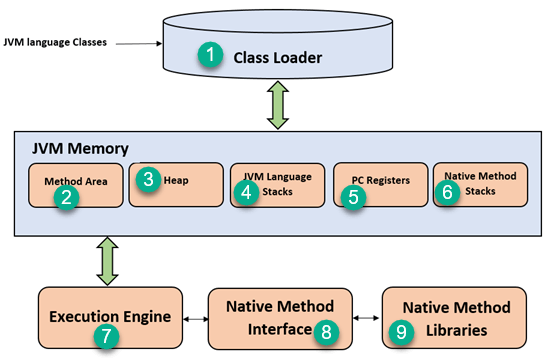
JRE has an instance of JVM with it, library classes, and development tools. Once you write and compile Java code, the compiler generates a class file having byte code.

Here are the important components of JRE:

* **Class loaders:**The class loader loads various classes that are necessary for running a Java program. JVM uses three class loaders called the bootstrap class loader, extensions class loader, and system class loader.
* **Byte code verifier:** Byte code verifier verifies the bytecode so that the code doesn't disturb the interpreter.
* **Interpreter:** Once the classes get loaded, and the code is verified, the interpreter reads the code line by line.
* **Run-time:**Run-time is a system used mainly in programming to describe time period during which a particular program is running.
* **Hardware:** Once you compile Java native code, it runs on a specific hardware platform.

In this way, the Java program runs in JRE.

**How JVM Functions? :**



Here are the important components of JVM:

**1) Class Loader :** The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

**2) Method Area :** JVM Method Area stores structure of class like metadata, the code for Java methods, and the constant runtime pool.

**3) Heap :** All the Objects, arrays, and instance variables are stored in a heap. This memory is shared across multiple threads.

**4) JVM language Stacks :** Java language Stacks store local variables, and its partial results. Each and every thread has its own JVM language stack, created concurrently as the thread is created. A new frame is created when method is invoked, and it is removed when method invocation process is complete.

**5) PC Registers :** PC registers store the address of the Java virtual machine instruction, which is currently executing. In Java, each thread has its separate PC register.

**6) Native Method Stacks :**Native method stacks hold the instruction of native code depends on the native library. It allocates memory on native heaps or uses any type of stack.

**7) Execution Engine :**It is a type of software that is used to test software, hardware, or complete systems. The test execution engine never carries any information about the tested product.

**8) Native Method interface :** The Native Method Interface is a programming framework. It allows Java code, which is running in a JVM to call by libraries and native applications.

**9) Native Method Libraries :** Native Libraries is a collection of the Native Libraries (C, C++), which are needed by the Execution Engine.

**Difference between JDK, JRE and JVM :**

|  |  |  |
| --- | --- | --- |
| ****JDK**** | ****JRE**** | ****JVM**** |
| The full form of JDK is Java Development Kit. | The full form of JRE is Java Runtime Environment. | The full form of JVM is Java Virtual Machine. |
| JDK is a software development kit to develop applications in Java. | It is a software bundle which provides Java class libraries with necessary components to run Java code. | JVM executes Java byte code and provides an environment for executing it. |
| JDK is platform dependent. | JRE is also platform dependent. | JVM is platform-independent. |
| It contains tools for developing, debugging, and monitoring java code. | It contains class libraries and other supporting files that JVM requires to execute the program. | Software development tools are not included in JVM. |
| It is the superset of JRE | It is the subset of JDK. | JVM is a subset of JRE. |
| The JDK enables developers to create Java programs that can be executed and run by the JRE and JVM. | The JRE is the part of Java that creates the JVM. | It is the Java platform component that executes source code. |
| JDK comes with the installer. | JRE only contain environment to execute source code. | JVM bundled in both software JDK and JRE. |

**HashSet :**

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.



**The important points about Java HashSet class are:**

* HashSet stores the elements by using a mechanism called hashing.
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

**Constructor of HashSet :**

1) **HashSet()** It is used to construct a default HashSet.

2) **HashSet(int capacity)** It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.

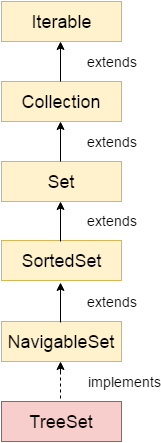
3) **HashSet(int capacity, float loadFactor)** It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.

4) **HashSet(Collection<? extends E> c)** It is used to initialize the hash set by using the elements of the collection c.

1. boolean **add(E e) ->**  It is used to add the specified element to this set if it is not already present.
2. void **clear() ->** It is used to remove all of the elements from the set.
3. object **clone()->** It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4. boolean **contains(Object o) ->** It is used to return true if this set contains the specified element.
5. boolean **isEmpty() ->** It is used to return true if this set contains no elements.
6. Iterator<E> **iterator() ->** It is used to return an iterator over the elements in this set.
7. boolean **remove(Object o) ->** It is used to remove the specified element from this set if it is present.
8. int **size() ->** It is used to return the number of elements in the set.

9) Spliterator<E> **spliterator() ->** It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

**TreeSet class :**



As shown in the above diagram, Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

**The important points about Java TreeSet class are:**

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

**TreeSet()** It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.

**TreeSet(Collection<? extends E> c)** It is used to build a new tree set that contains the elements of the collection c.

**TreeSet(Comparator<? super E> comparator)** It is used to construct an empty tree set that will be sorted according to given comparator.

**TreeSet(SortedSet<E> s)** It is used to build a TreeSet that contains the elements of the given SortedSet.

**boolean add(E e)** It is used to add the specified element to this set if it is not already present.

**boolean addAll(Collection<? extends E> c)** It is used to add all of the elements in the specified collection to this set.

**boolean contains(Object o)** It returns true if this set contains the specified element.

**boolean isEmpty()** It returns true if this set contains no elements.

**boolean remove(Object o)** It is used to remove the specified

element from this set if it is present.

**void clear()** It is used to remove all of the elements from this set.

**Object clone()** It returns a shallow copy of this TreeSet instance.

**E first()** It returns the first (lowest) element currently in this sorted set.

**E last()** It returns the last (highest) element currently in this sorted set.

**int size()** It returns the number of elements in this set.

**Iterator iterator()** It is used to iterate the elements in ascending order.

**E lower(E e)** It returns the closest least element of the specified element from the set, or null there is no such element.

**E pollFirst()** It is used to retrieve and remove the lowest(first) element.

**E pollLast()** It is used to retrieve and remove the highest(last) element.

**Java LinkedHashSet class :**

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class is non synchronized.
* Java LinkedHashSet class maintains insertion order.

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

**Rules for Exception handling w.r.t Method Overriding in Java**

1. If parent-class method ***doesn’t***declare any exception
2. If parent-class method declares***unchecked***exception
3. If parent-class method declares***checked***exception
4. If parent-class method declares both***checked***&***unchecked***exceptions

**Rule 1: If parent-class method doesn’t declare any exception**

1. Then child-class overriding-method can declare any type of unchecked exception  
   Note: this is the only possibility
2. If child-class overriding-method declares checked-exception, then compiler throws compile-time error stating  
   CTE – “Exception IOException is not compatible with throws clause in ParentClass.testMethod()”
3. Then child-class overriding-method can declare no exception in the overriding-method of child-class  
   This is very much same as that of overridden-method of parent-class (exactly same method-signature)

**Rule 2: If parent-class method declares unchecked–exception**

1. Then child-class overriding-method can declare any type of unchecked exception  
   Not necessarily same exception as that of parent-class’ method  
   (only for unchecked exception)
2. If child-class overriding-method declares any checked-exception, then compiler throws compile-time error stating  
   CTE – “Exception IOException is not compatible with throws clause in ParentClass.testMethod()”
3. Then child-class overriding-method can declare no exception in the overriding-method of child-class

**Rule 3: If parent-class method declares checked exception**

1. Then child-class overriding-method can declare any type of unchecked exception
2. Then child-class overriding-method can declare same type of checked exception or one of its sub-class or no exception

OR, sub-type of declared checked exception

1. Then child-class overriding-method can declare no exception in the overriding-method of child-class

**Rule 4: If parent-class method declares combination of both checked & unchecked exceptions**

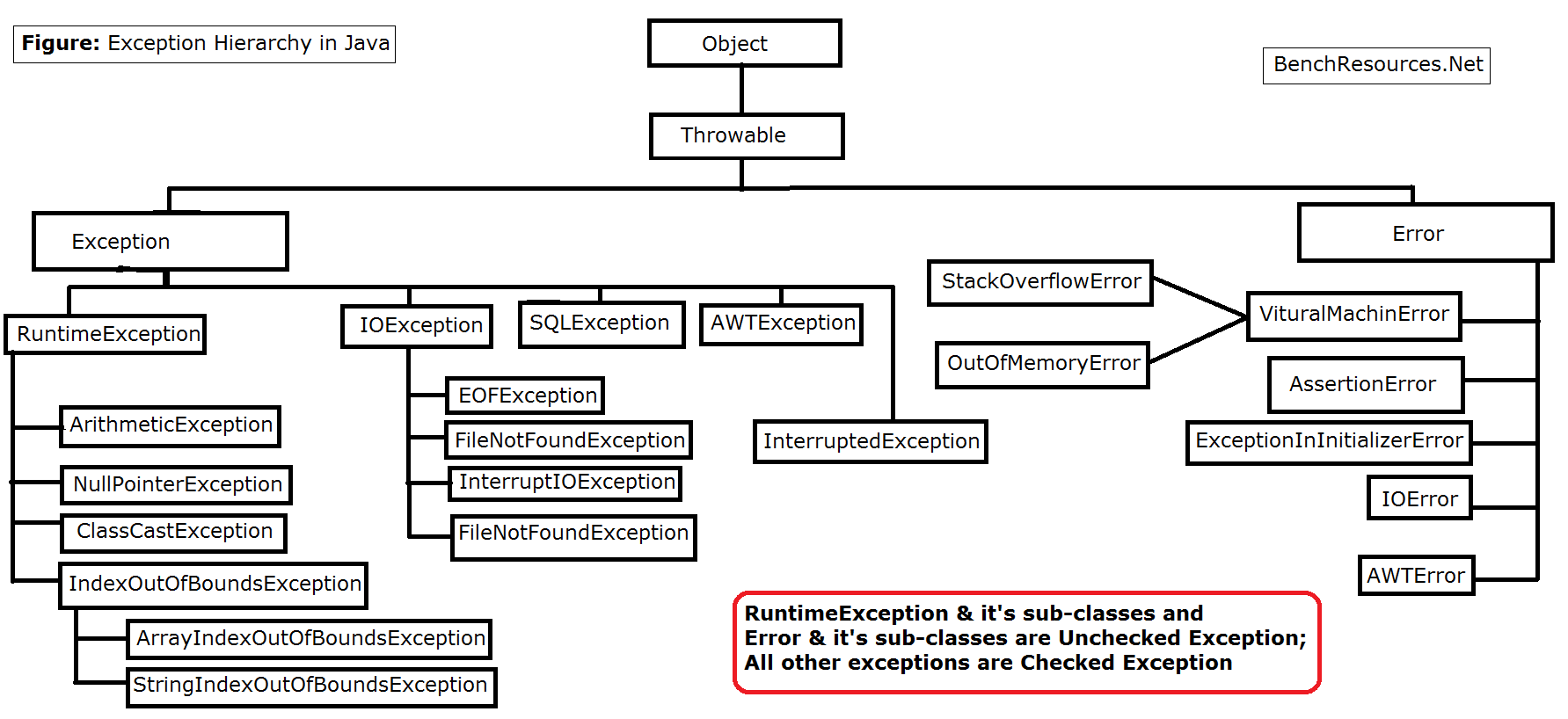
1. Then child-class overriding-method can declare any type of unchecked exception.
2. Then child-class overriding-method can declare same type of checked-exception or one of its sub-class or no exception.
3. Then child-class overriding-method can declare no exception in the overriding-method of child-class.

* When parent-class method declares no exception, then child-class overriding-method can declare,  
  1. No exception or  
  2. Any number of unchecked exception  
  3. but strictly no checked exception
* When parent-class method declares unchecked exception, then child-class overriding-method can declare,  
  1. No exception or  
  2. Any number of unchecked exception  
  3. but strictly no checked exception
* When parent-class method declares checked exception, then child-class overriding-method can declare,  
  1. No exception or  
  2. Same checked exception or  
  3. Sub-type of checked exception or  
  4. any number of unchecked exception
* All above conclusion hold true, even if combination of both checked & unchecked exception is declared in parent-class’ method

**Exception Hierarchy in Java:**

**Throwable class** is the root class for every exception and it branches out to 2 main categories i.e.;

* Exception
* Error



**java.lang.Throwable:**

* Throwable is the root class for exception & it’s sub-type and error & it’s sub-types
* In other words, it is super class for exception & error
* java.lang.Throwable class extends java.lang.Object class (as shown in the above figure)
* It defines 2 sub classes i.e.; Exception and Error

**java.lang.Exception**:

* java.lang.Exception is super class for all types of Exception
* It extends java.lang.Throwable class
* Exception are due to programmatic logic And it is recoverable
* Exception are categorized into checked exception and unchecked exception
* Example: RuntimeException, SQLException, IOException, FileNotFoundException, ArithmeticException, NullPointerException

**java.lang.Error:**

* java.lang.Error is super class for all types of Error
* It extends java.lang.Throwable class
* Error are due to lack of system resources
* And it is non-recoverable
* All error fall into unchecked exception category, as it is raised due to lack of system resources at runtime
* It is out of programming scope as such type of error can’t predicted, may be well planned care can be taken to avoid these kind of Error
* Example: VirtualMachineError, AssertionError, ExceptionInInitializerError, StackOverflowError, OutOfMemoryError, LinkageError, InstantiationError

**Checked Exception v/s Unchecked Exception**

**Checked Exception:**

* Exception which are checked at compile-time during compilation is known as Checked Exception
* Alternate definition: any line of code that could possibly throw exception, and if it is raised to handle during compilation is said to be checked exception
* For example, accessing a file from remote location could possibly throw file not found exception
* It is the programmer’s responsibility to handle the checked exception for successful compilation
* This way, if any exception is raised during execution then respective handling code will be executed
* Note: if it isn’t handled then program will throw compile-time error
* Example: IOException, FileNotFoundException, InterruptedException, SQLException, etc
* Except Runtime exception & its child classes and error & its child classes, all other exception falls under the category of Checked Exception

**Unchecked Exception:**

* Exception which are NOT checked at compile-time is known as Unchecked Exception
* Alternate definition: any line of code that could possibly throw exception at runtime is said to be unchecked exception
* Unchecked exception are because of programming-error
* For example, accessing out of index-position to assign some value during execution could possibly throw exception at runtime
* So, it is again programmer’s responsibility to handle unchecked exception by providing alternate solution in the exception handling code
* Note: if it isn’t handled properly then program will terminate abnormally at runtime
* Example: Runtime exception & its child classes and error & its child classes are examples of Unchecked Exception
* Like ArithmeticException, NullPointerException, NumberFormatException, ArrayIndexOutOfBoundsException, StatckOverflowError, etc

**Misconception about checked and unchecked exception:**

* Sometimes, checked exception are also referred as compile-time exception and unchecked exception are referred as runtime exception
* But this is mis-leading because every exception (whether it is checked or unchecked) occurs/raised only at the runtime i.e.; during program execution only
* Reason: during compilation; checked exception are caught and raises compile-time error, due to which programmer has to handle the exception by providing either try-catch blocks or using throws keyword
* Whereas unchecked exception aren’t caught during compilation, rather it raises exception during execution because of programming error

**Difference between SynchronizedMap and ConcurrentHashMap classes:**

|  |  |
| --- | --- |
| **SynchronizedMap** | **ConcurrentHashMap** |
| This is thread-safe version of Map | ConcurrentHashMap is newly introduced thread-safe class |
| Only one thread is allowed to operate on synchronized map, by locking over complete map object | Multiple threads are allowed to operate on concurrent map, by locking over portion of map object i.e.; at segment-level or bucket-level |
| Every operations like read and update requires lock over complete map object before operating on map object | Read operation doesn’t require lock but update operation require definitely lock on the portion of map object i.e.; at segment-level or bucket-level |
| While one thread iterating Map items, if any other thread tries to modify Map items then ConcurrentModificationException is thrown | While one thread iterating ConcurrentHashMap items, other thread are happily can modify Map items  And it never throws ConcurrentModificationException |
| That’s it is fail-fast iterator | That’s it is fail-safe iterator |
| NULL insertion is possible for key but maximum of one null key and any number of null values against any key | NULL insertion isn’t allowed for both keys and values |
| This is introduced in original collection framework in Java 1.2 version | This is introduced in Java 1.5 version |

**When to use SynchronizedMap ?**

* This is generally used to convert map object into thread-safe Map object
* But only one thread is allowed to operate on map object, as lock is required over complete map object
* So, performance degrades comparatively in a multi-threaded environment
* So, use this only when it is required to convert into thread-safe version of Map object

**When to use ConcurrentHashMap ?**

* This is the best suit to store key-value pairs in a multi-threaded environment
* And also one thread iterating never stops other threads to modify
* And it never throws ConcurrentModificationException

**[ClassNotFoundException vs. NoClassDefFoundError :](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

**[ClassNotFoundException:](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

* [ClassNotFoundException occurs when you try to load a class at runtime using Class.forName() or loadClass() methods and requested classes are not found in classpath.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)
* [This exception is a checked Exception derived from java.lang.Exception class.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)
* [ClassNotFoundException occurs when classpath is does not get updated with required JAR files](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)
* [public static void main(String args[]) {](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

[try     {](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

[Class.forName("GeeksForGeeks");](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

[}    catch (ClassNotFoundException ex)        {](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

[ex.printStackTrace();](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

[}](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

[}](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**[NoClassDefFoundError :](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

* [Java Virtual Machine is not able to find a particular class at runtime which was available at compile time.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)
* [If a class was present during compile time but not available in java classpath during runtime.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)
* [NoClassDefFoundError is an error.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)
* [Missing JAR files are the most basic reason to get NoClassDefFoundError.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**[What is try-with-resources in Java?](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

[In Java, the try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program. The try-with-resources statement ensures that each resource is closed at the end of the statement execution.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**[Why we use try-with-resources?](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

[Support for try-with-resources – introduced in Java 7 – allows us to declare resources to be used in a try block with the assurance that the resources will be closed when after the execution of that block. The resources declared must implement the AutoCloseable interface.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**[Does try-with-resources need finally?](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

[In try-with-resources method there is no use of finally block. the file resource is opened in try block inside small brackets. Only the objects of those classes can be opened within the block which implements AutoCloseable interface and those object should also be local.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**[How try resources work internally?](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

**[Throwable.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)** [That is the concept of try-with-resources. If an exception is thrown in the try block, then the control will be transferred to catch. In between the jump to catch block the close() will be internally invoked for the registered resources.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**[What are the resources used in exception handling?](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)**

[The Java try with resources construct, AKA Java try-with-resources, is an exception handling mechanism that can automatically close resources like a Java InputStream or a JDBC Connection when you are done with them. To do so, you must open and use the resource within a Java try-with-resources block.](https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro" \l ":~:text=ClassNotFoundException is an exception that,was missing at run time.)

**Writing your own exception class**

Here are the steps:

* Create a new class whose name should end with Exception like ClassNameException. This is a convention to differentiate an exception class from regular ones.
* Make the class extends one of the exceptions which are subtypes of the java.lang.Exception class. Generally, a custom exception class always extends directly from the Exception class.
* Create a constructor with a String parameter which is the detail message of the exception. In this constructor, simply call the super constructor and pass the message.

That’s it. The following is a custom exception class which is created by following the above steps:

public class StudentNotFoundException extends Exception {

public StudentNotFoundException(String message) {

super(message);

}

}

And the following example shows the way a custom exception is used is nothing different than built-in exception:

public class StudentManager {

public Student find(String studentID) throws StudentNotFoundException {

if (studentID.equals("123456")) {

return new Student();

} else {

throw new StudentNotFoundException(

"Could not find student with ID " + studentID);

}

}

}

**Marshalling** is the process of writing **Java** objects to XML file. Unmarshalling is the process of converting XML content to **Java** objects.

**What is JAXB marshalling and Unmarshalling?**

**JAXB** stands for Java Architecture for XML Binding. ... As part of this process, **JAXB** provides methods for **unmarshalling** (reading) XML instance documents into Java content, and then **marshalling** (writing) Java content back into XML instance documents. **JAXB** also provides a way to generate XML schemas from Java objects.

<https://attacomsian.com/blog/java-read-write-xml>

**Can we override a main method state the reasons? :** NO, we can not override main method in java. Reason is very simple. As main method is static and we know very well that we can not override static methods in Java, hence main method could not be overridden.

**Can a constructor be final?**

No, a constructor can't be made final. A final method cannot be overridden by any subclasses. ... But, in inheritance sub class inherits the members of a super class except constructors. In other words, constructors cannot be inherited in Java therefore, there is no need to write final before constructors.

**Why is main method static?**

Java main() method is always static, so that compiler can call it without the creation of an object or before the creation of an object of the class. ... Static method of a class can be called by using the class name only without creating an object of a class.

**POJO vs Java Bean :**

|  |  |
| --- | --- |
| **POJO** | **Java Bean** |
| It doesn’t have special restrictions other than those forced by Java language. | It is a special POJO which have some restrictions. |
| It doesn’t provide much control on members. | It provides complete control on members. |
| It can implement Serializable interface. | It should implement serializable interface. |
| Fields can be accessed by their names. | Fields are accessed only by getters and setters. |
| Fields can have any visiblity. | Fields have only private visiblity. |
| There may/may-not be a no-arg constructor. | It must have a no-arg constructor. |
| It is used when you don’t want to give restriction on your members and give user complete access of your entity | It is used when you want to provide user your entity but only some part of your entity. |
| **public** **class** Employee{      // default field      String name;        // public field  **public** String id;        // private salary  **private** **double** salary;        //arg-constructor to initialize fields  **public** Employee(String name, String id,  **double** salary)      {  **this**.name = name;  **this**.id = id;  **this**.salary = salary;      } | class Bean  {      // private field property      private Integer property;      Bean()      {          // No-arg constructor      } |

**Covariant return type** refers to return type of an overriding method. It allows to narrow down return type of an overridden method without any need to cast the type or check the return type. Covariant return type works only for non-primitive return types.

From Java 5 onwards, we can override a method by changing its return type only by abiding the condition that return type is a subclass of that of overridden method return type.

In object-oriented programming, a covariant return type of a method is one that can be replaced by a "narrower" type when the method is overridden in a subclass. ... This usually implies that the return types of the overriding methods will be subtypes of the return type of the overridden method.

# **[Overriding vs. Overloading in Java](https://www.programcreek.com/2009/02/overriding-and-overloading-in-java-with-examples/)**

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Overriding means having two methods with the same method name and parameters (i.e., method signature). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.

* Polymorphism applies to overriding, not to overloading.
* Overriding is a run-time concept while overloading is a compile-time concept.
* Overloading happens at [compile-time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/" \t "https://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/_blank) while Overriding happens at [runtime](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/" \t "https://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/_blank): The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
* Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
* The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
* [Static binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/" \t "https://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/_blank) is being used for overloaded methods and [dynamic binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/" \t "https://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/_blank) is being used for overridden/overriding methods.
* Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
* private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
* Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type ([refer this](https://stackoverflow.com/questions/14694852/can-overridden-methods-differ-in-return-type" \t "https://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/_blank)).
* Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

**Immutable class with mutable member fields in Java :**

Immutable class is a class which once created, it's content can not be changed. Immutable classes are good choice for HashMap key as their state cannot be changed once they are created. Objects of immuable classes are also thread safe as threads can not change the value of its fields once it is created.

Creating a basic immutable class is very easy, but as you dig deep you will start facing many challenges. This makes Immutability a very famous interview topic for mid level java developers.

where we have discussed about creating basic immutable class using traditional approach as well as using Builder Design Pattern approach.

we will discuss about How we will achieve immutability, if it has member variables of any third party class which is mutable. Or what will you do if you have reference of any built-in java collection class which is mutable like ArrayList, LinkedList, etc.

class User which has three fields firstName, lastName and Address of type, . To make this class immutable,

We will declare class as final and all the fields as private final

We will provide one parameterized constructor and getter methods.

We will not provide any setter method, so that field values can not be changed after object creation.

Class User{

String firstName,

String lastName

String address

}

Above class is immutable as we can not change field values after object creation. Now, lets assume that there is a change in requirement and instead of storing address in object, we have to store it in more organized way. We have to use rich third party Address class which has options to store address very efficiently using different fields for firstLine, secondLine and city.

Class Address {

String firstLine;

String secondLine;

String city;

//Parameterized constructor and Setter & getter

}

Lets change type of address from String to Address in User class.

Class User{

String firstName,

String lastName

Address address

}

As of now all the fields of User class were String and itself tring itself is immutable. But Address class has setter methods and hence now User class has one mutable member field.  Can this break immutability of User class?

 Lets try to break immutability of User class.

Here, first we got reference of address object using getAddress() method. We have stored this reference in new local variable, it was still pointing to the same address instance. So, when we changed value of firstLine, secondLine and city fields of address instance, it updated the address instance being used by User object. And hence, when we tried to get address of user, it printed updated address.

How can we achieve immutablility in such case?

Option 1)    
We will not provide any setter methods in Address class so that nobody can change properties of address class. If you will answer this, interviewer will counter you saying Address is a third-part java class i.e. It is being referred from third party jar file and we do not have access to the source code of Address.

Option 2)  We can create child class of Address class, override all the setter methods and then explicitely throw UnsupportedOperationException  from those setter methods.

Is there any problem with this approach?   
Well yes. what if some of the reference variables inside Address class is also Mutable Objects. In that case we need to override their  setter methods as well. This approch becomes more complex when there are many nested Mutable class references.

Option 3)   
Another option is to modify getAddress method of User class. Instead of returning the original Address instance, we will return deep cloned copy of that Adress instance. Even if third party user makes any changes to this cloned address object, it will not affect the original address object of User object.

clone() method only works if Address has implemented Cloneable interface. If it has not implemented it, then we have to manually deep copy all the fields of Address class. But most of the user library has support for Cloneable and Serializable interfaces.

Even if we changed values inside cloned address object, it has not affected original Address value inside User object. If interviewer ask you this question, you can directly tell him about this solution, but you should also be aware about other available options.

Class User{

String firstName,

String lastName

Address address

Public Adress getAdress(){

Return address.clone();

}

}

Mutable Collections as field of Immutable Object : Assume that instead of storing just one Address, we are storing List of Addresses inside User class i.e. Primary Address, Secondary Address, Work Address etc.

Option 1)  
When we have Collection like ArrayList or LinkedList as member variables, we should not use their in-built clone() method. Their clone() method returns shallow copy of this ArrayList instance. The elements themselves are not copied. So in such case, we can write our own method to deep copy ArrayList object.

Option 2)   
We can also use the copy-constructors – new ArrayList(originalList) to create a deep cloned ArrayList object. It would be easier than writing our own deep copy method.

Option 3)   
Collection framework also provides implementation of Unmodifiable Collection classes in Collections utility class. These Unmodifiable collections are just a wrapper around normal Collection classes which throws UnsupportedOperationException from all the methods which tries to modify Collection i.e. add() remove() , etc. You can check implementation of UnmodifiableList.

So how we will use this Unmodifiable collections? Insterad of returning plain ArrayList object, we now return UnmodifiableList of Addresses from getAddress() method of User class.

**To create a custom immutable class we have to do the following steps :**

* Declare the class as final so it can’t be extended.
* Make all fields private so that direct access is not allowed.
* Do not provide setter methods (methods that modify fields) for variables, so that it can not be set.
* Make all mutable fields final so that their values can be assigned only once.
* Initialize all the fields through a constructor doing the deep copy.
* Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.
* If the instance fields include references to mutable objects, don’t allow those objects to be changed
* Don’t provide methods that modify the mutable objects.
* Don’t share references to the mutable objects. Never store references to external, mutable objects passed to the constructor. If necessary, create copies and store references to the copies. Similarly, create copies of our internal mutable objects when necessary to avoid returning the originals in our methods.

**Wildcards in Java :** The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.

**Types of wildcards in Java :**

**Upper Bounded Wildcards:**These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this  using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the extends keyword, followed by its upper bound.

public static void add(List<? extends Number> list)

**Lower Bounded Wildcards:** It is expressed using the wildcard character (‘?’), followed by the super keyword, followed by its lower bound: <? super A>.

Syntax: Collectiontype <? super A>

**Unbounded Wildcard:**This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown type. These are useful in the following cases

* When writing a method which can be employed using functionality provided in Object class.
* When the code is using methods in the generic class that don’t depend on the type parameter

**class** unboundedwildcardemo{

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

        //Integer List

        List<Integer> list1= Arrays.asList(1,2,3);

        //Double list

        List<Double> list2=Arrays.asList(1.1,2.2,3.3);

        printlist(list1);

        printlist(list2);

    }

**private** **static** **void** printlist(**List<?> list**)     {

        System.out.println(list);

    }

}