**Java Interview Questions**

1. What are **OOPS concepts**?

**Abstraction** – Abstraction shows only the relevant details/features to the object **and hides the irrelevant or implementation details/features**. You just need to know which methods of the object are available to call and which input parameters are needed to trigger a specific operation. But you don’t need to understand how this method is implemented and which kinds of actions it has to perform to create the expected result. It reduces the complexity of viewing the things. Abstraction hides the complexity from the user and shows only relevant information.

**In java, abstraction is achieved by interfaces and abstract classes. Interfaces allows you to abstract the implementation completely while abstract classes allow partial abstraction as well.**

Data abstraction and Control abstraction are types of abstraction.

For example, a **HashMap** stores key-value pairs. It provides you two methods **get() and put()** methods to store and retrieve key-value pairs from map. It is, in fact, the only information you will need if you want to use the map in your application. How it works inside, you are not required to know it to use it. This is very much example of abstraction in Java.

Take a more real-life example of abstraction which can be a TV remote. You know that when you press any button in remote, some function is applied on television e.g. change the channel, change the volume level etc. You are not required to know how internally remote works, to use it properly. It is an abstraction example.

**Abstraction in Java:**

* Hides the underlying complexity of data
* Helps avoid repetitive code
* Presents only the signature of internal functionality
* Gives flexibility to programmers to change the implementation of the abstract behaviour
* Partial abstraction (0-100%) can be achieved with abstract classes
* Total abstraction (100%) can be achieved with interfaces

**Encapsulation** – Encapsulation is the technique used to implement Abstraction in object-oriented programming. Encapsulation allows us to protect the data stored in a class from system-wide access. It **protects/hides data implementation** by wrapping the implementation (code) and the data it manipulates (variables) into a single unit i.e the same class.

A Java class, where all **instance variables are private and only public setter and getter methods** within the class can manipulate those variables, is an example of an encapsulated class. **Java Beans** are examples of fully encapsulated classes.

**Encapsulation in Java:**

* Restricts direct access to data members (fields) of a class.
* Fields are set to private
* Each field has a getter and setter method
* Getter methods return the field
* Setter methods let us change the value of the field

**Polymorphism** – It means ability to perform a certain action in different ways i.e. process objects differently based on their data type. In Java, Polymorphism could be **static** and **dynamic** both. **Method Overloading** is static polymorphism while, **Method overriding** is dynamic polymorphism.

* **Overloading** in simple words means more than one method having the same method name that behaves differently based on the arguments passed while calling the method. This called static because, which method to be invoked is decided at the time of compilation
* **Overriding** means a derived class is implementing a method of its super class. The call to overriden method is resolved at runtime, thus called runtime polymorphism.
* **This can be implemented by designing a generic interface**, which provides generic methods for a certain class of action and there can be multiple classes, which provides the implementation of these generic methods.

**Inheritance** – It is a mechanism to create a child class that inherits the fields and methods(properties) of the parent class. The child class can override the values and methods of the parent class, however it’s not necessary. It can also add new data and functionality to its parent. Parent classes are also called super classes or base classes, while child classes are known as subclasses or derived classes as well. Java uses the **extends** keyword to implement the principle of inheritance in code.

**Inheritance in Java:**

* A class (child class) can extend another class (parent class) by inheriting its features.
* Implements the DRY (Don’t Repeat Yourself) programming principle.
* Improves code reusability.
* Multilevel inheritance is allowed in Java (a child class can have its own child class as well).
* Multiple inheritances are not allowed in Java (a class can’t extend more than one class).

**Association** – Besides the four main principles of OOP, Java also works with three further concepts (association, aggregation, composition) you can make use of when designing your programs. Aggregation is a special form of association, while composition is a special form of aggregation.

**Association simply means defining a relationship between two unrelated classes.** For example, when you declare two fields of different types (e.g. Car and Bicycle) within the same class and make them interact with each other, you have performed association.

**Association in Java:**

* Two separate classes are associated through their objects.
* The two classes are unrelated, each can exist without the other one.
* Can be a one-to-one, one-to-many, many-to-one, or many-to-many relationship.

**Aggregation** – Aggregation is a narrower kind of association**. It occurs when there’s a one-way (HAS-A) relationship between the two classes you associate through their objects**. For example, every Passenger has a Car but a Car doesn’t necessarily have a Passenger. When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to. Then, when you instantiate a new Passenger object, you can access the data stored in the related Car as well.

Aggregation in Java:

* **One-directional association.**
* **Represents a HAS-A relationship between two classes**.
* Only one class is dependent on the other.

**Composition** – It is a stricter form of aggregation. **It occurs when the two classes you associate are mutually dependent on each other and can’t exist without each other**. For example, take a Car and an Engine class. A Car cannot run without an Engine, while an Engine also can’t function without being built into a Car. This kind of relationship between objects is also called a **PART-OF relationship**.

Composition in Java:

* A restricted form of aggregation
* Represents a PART-OF relationship between two classes
* Both classes are dependent on each other
* If one class ceases to exist, the other can’t survive alone

Refer - <https://www.baeldung.com/java-inheritance-composition>

1. What are access modifiers?

Only classes, interfaces and enums can exist at top level, everything else must be included within one of these.

**public**: the object is visible to all classes **everywhere**, whether they are in the same package or have imported the package containing the public class.

**package-private**: the object is only available **within its own package**. Package-private is specified by not specifying, i.e. it is default. There is no package-private keyword.

Member Level

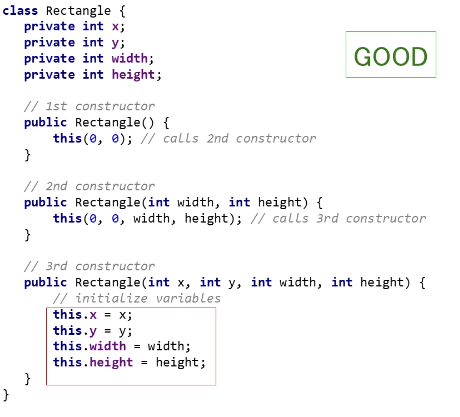
**protected**: the object is available **within its own package** (like default) but also in **subclasses** even if they are **in another package**.

**private**: members will be available only within the class and it’s not accessible outside the class.

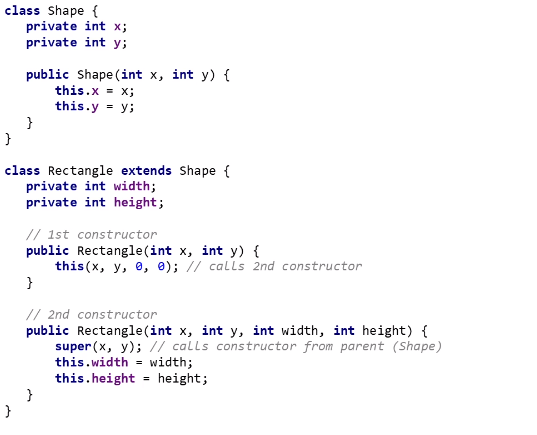
1. Difference between super () and this ()?

|  |  |
| --- | --- |
| super () | this () |
| It is used to call the Base class (Parent class) constructor. | It is used to call the current class constructor. |
| super () should be first statement inside any constructor. | this () should be first statement inside any constructor. |
| It can be used only inside constructor and nowhere else. | It can be used only inside constructor and nowhere else. |
| It can’t be used in static reference. | It can’t be used in static reference. |
| super () can be used to call the parent class constructor. | It can be used in constructor chaining. |
| super () and this () can’t be used together inside the constructor. Otherwise, there will be compilation error. | super () and this () can’t be used together inside the constructor. Otherwise, there will be compilation error. |
| super keyword is used to access the parent class members. | this keyword is used to access the current class members. |

This constructor chaining.



This and super example



1. Difference between method **Overloading and Overriding**?

|  |  |
| --- | --- |
| **Overloading** | **Overriding** |
| It means providing two or more separate methods in a class with same name but different parameters. | It means defining a method in a child class that already exists in the parent class with the same signature (same name and arguments) |
| It reduces duplicate code and we don’t have to remember the multiple method names. | @Override annotation is used to enforce the overriding rules. |
| It is also considered as compile time polymorphism. | It is also considered as runtime polymorphism. |
| Static and Instance method can be overloaded. | Static, final and private methods can’t be overridden only the instance methods. |
| A method can also be treated as overloaded in the subclass of that class. | A method is usually overridden in subclass. |
| Overloaded methods may have different return types. | Return type can be a subclass of the return type in parent class. |
| Overloaded methods may have different access modifiers. | It can’t have the lower access modifiers. Eg: if the parent method is “protected” then using “private” in the child is not allowed but using “public” would be allowed. |
| Throw different checked or unchecked exceptions. | Must not throw a new or broader checked exception. |

Difference between **static** and **instance** variable/method?

**Static** variable and method belong to the Class not to the object which means all the objects shares the same copy of variable and method. So, the changes made by one object to the variable will be visible to other objects.

**Note**: Non-static variable or methods can not be referenced from a static context.

**Instance** variable and method belong to object which means each object will have their own copy of instance variable and methods.

What is **final** variable, method and class?

**Classes** marked as final can’t be extended. Subclasses cannot be created.

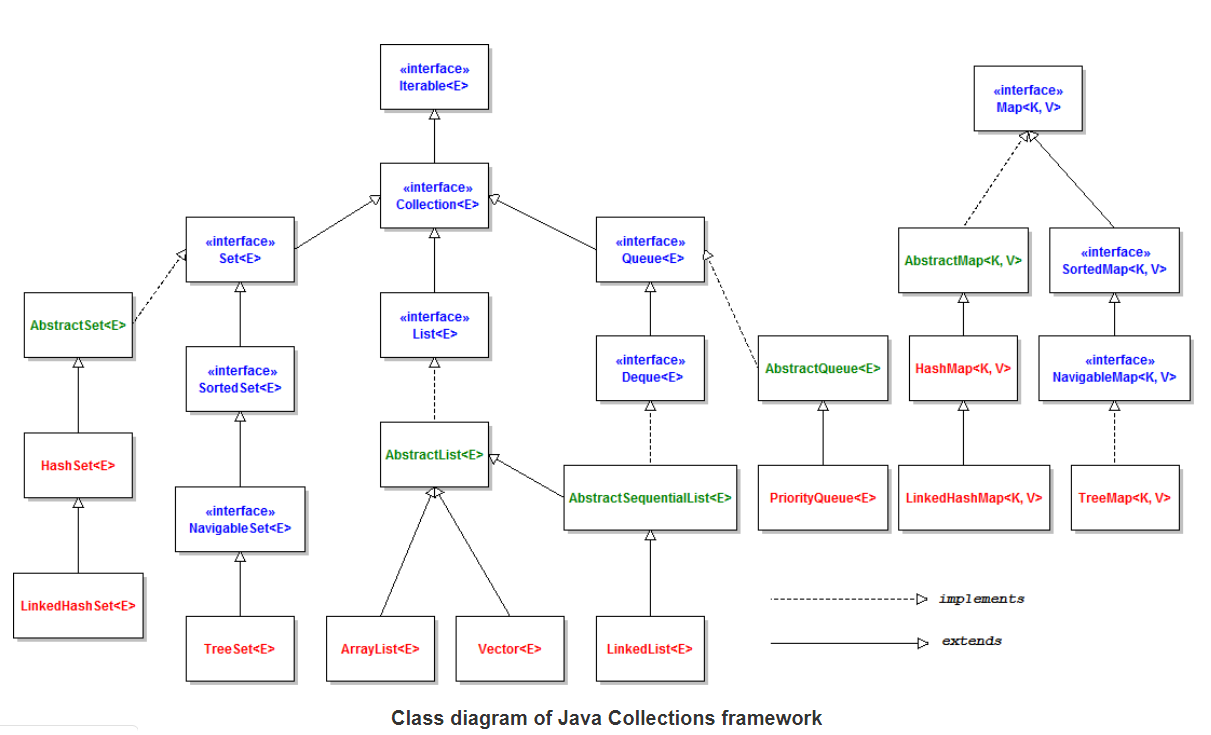
**Methods** marked as final cannot be overridden. Sometimes we don’t need to prohibit a class extension entirely, but only prevent overriding of some methods. A good example of this is the Thread class. It’s legal to extend it and thus create a custom thread class. But its isAlive() methods is final.

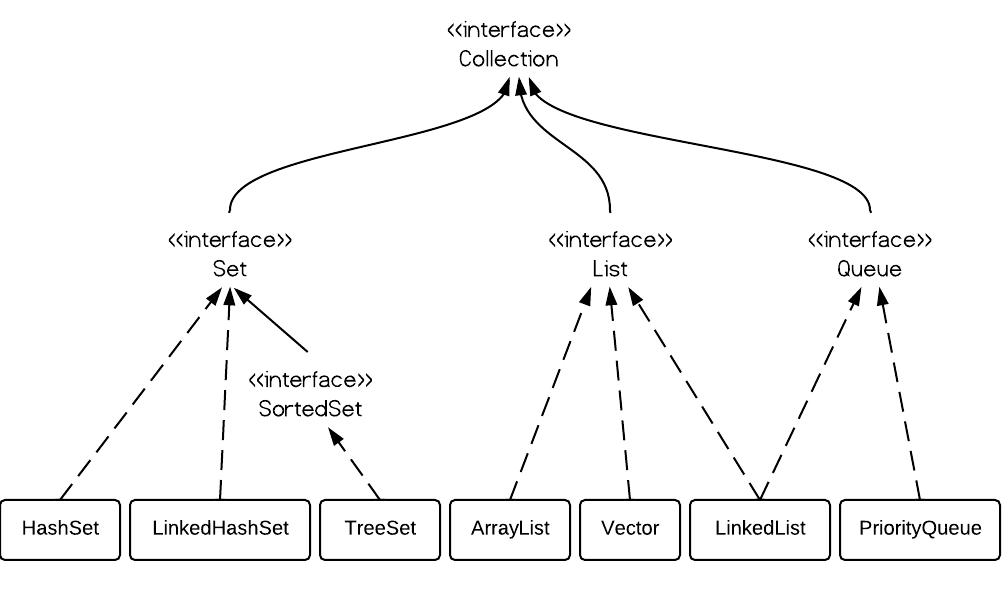
**Variables** marked as final can't be reassigned. Once a final variable is initialized, it can’t be altered.

What is Reference Type and Value Type?

Autoboxing and Unboxing?

**Collections**





1. Difference between **Array** and **ArrayList**?

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Array is a **fixed length** data structure. Length of the array can’t be changed once the array is created. | ArrayList a **variable length** Collection class. |
| **Resizing** is **not possible** in Array. | ArrayList **re-sizes itself** when gets full depending upon the **capacity and load factor**. Automatic resize of ArrayList will slow down the performance as it will use temporary array to copy elements from the old array to new array. |
| Array provides **length variable** which denotes length of Array | ArrayList provides **size ()** method to calculate size of ArrayList |
| **Generics can’t be used** with Array, as Array knows about what kind of type it can hold and throws **ArrayStoreException,** if you try to store type which is not convertible into type of Array. | **ArrayList allows you to use Generics** to ensure type-safety. |
| Array can **contain both primitive data types as well as objects.** | ArrayList **cannot** contains **primitive data types** (like int, float, double) it can only **contain Objects** |
| You can simply **use assignment operator to store element** into Array | Java provides **add()** method to insert element into ArrayList |
| Array can be multi-dimensional | ArrayList is always single dimensional. |

1. Difference between **Vector** and **ArrayList**?

**ArrayList and Vector**, both implements **java.util.List** interface and provide capability to store and get objects within using simple API methods.

|  |  |
| --- | --- |
| **Vector** | **ArrayList** |
| Vector is a **synchronized** collection. | ArrayList is **not a synchronized** collection. |
| **Vector can be used in concurrent application** without any additional synchronization control implemented by developer using **synchronized** keyword. Public methods inside vector are defined **synchronized** which make all operations in vector safe for concurrency needs. | To use Arraylist in concurrent application, we must explicitly control the thread access to instance to make application work as intended. If we want to get synchronized version of Arraylist, then we can use **Collections.synchronizedList()** method or **CopyOnWriteArrayList** |
| By default, when vector gets filled, it **increases capacity by 100%**. It means vector size grows to double of previous capacity. Default capacity can be overridden using constructor **public Vector(int initialCapacity, int capacityIncrement)** | By default, **capacity grows by 50%** of existing capacity. In ArrayList, we can define the initial capacity but not the capacity increment factor. |
| Default capacity of vector is 10. |  |
| Vector is **synchronized**, so it has some **overhead** in thread management. | **ArrayList** is **non-synchronized** so there is no time lapse in thread safety and therefore it’s **faster**. |
| It provides **iterator** as well as **enumeration**. **Iterators are fail-fast, but enumerations are not**. If we modify the vector during iteration over enumeration, **it does not fail**. | It provides **iterator**, which are **fail-fast**. As soon as we modify the Arraylist structure (add or remove elements), the iterator will throw **ConcurrentModificationException** error. |

Refer <https://howtodoinjava.com/java/collections/arraylist/arraylist-vs-vector/>

1. Difference between **LinkedList** and **ArrayList**?

**ArrayList and LinkedList**, both implements **java.util.List** interface and provide capability to store and get objects as in **ordered collections** using simple API methods.

|  |  |
| --- | --- |
| **LinkedList** | **ArrayList** |
| LinkedList internally uses a **doubly linked list** to store the elements. | ArrayList internally uses a **dynamic array** to store the elements. |
| **Appending an element** in LinkedList is **O(1)** operation, as it doesn’t require any navigation. | **Adding element** in ArrayList is O(1) operation if it doesn’t require **resize of Array**. If array is **resized** then it becomes **O(log(n)).** |
| LinkedList **remove** operation gives **O(1)** performance because it just need to reset the pointers of previous and next nodes. No copy or movement is required. | When we **remove an element** from ArrayList, it moves/shift all elements on right. It makes it close to O(n) in worst case (remove first element) and O(1) in best case (remove last element). |
| Iteration is the **O(n)** operation for both LinkedList and ArrayList where n is a number of an element. | Iteration is the **O(n)** operation for both LinkedList and ArrayList where n is a number of an element. |
| LinkedList also provide **get(int index)** method BUT it first traverses all nodes to reach the correct node. It makes the performance variable. In best case it is O(1) and in worst case it is O(n). | ArrayList provides get(int index) method which directly find the element at given index location. It is of order **O(1).** |
| LinkedList is **better for addition and removal**. | ArrayList is **better for accessing the data**. |
| It has **more memory overhead** because each node holds data and address of next and previous node. | It has **less memory overhead** because each index holds the actual object. |
| LinkedList class can act as a list and queue both because **it implements List and Deque interfaces**. | An ArrayList class can act as a list only because **it implements List only**. |

1. Difference between ArrayList and HashSet?

|  |  |
| --- | --- |
| **ArrayList** | **HashSet** |
| ArrayList is the **implementation** of the **List interface.** | HashSet on the other hand is the **implementation** of a **Set interface.** |
| ArrayList is **backed** by an **Array** | HashSet is **backed** by an **HashMap** |
| ArrayList **maintains the insertion order** i.e order of the object in which they are inserted. | HashSet is an **unordered collection** and doesn't maintain any order. |
| ArrayList **allows duplicate values** in its collection. | On other hand **duplicate elements are not allowed** in Hashset. |
| **Any number of null value can be inserted** in ArrayList without any restriction. | On other hand Hashset **allows only one null value** in its collection,after which no null value is allowed to be added. |
| ArrayList uses index for its performance i.e its index based one can retrieve object by calling get(index) or remove objects by calling remove(index) | HashSet is completely based on object also it doesn't provide get() method. |

1. Difference between HashSet, LinkedHashSet and TreeSet?

|  |  |  |
| --- | --- | --- |
| **HashSet** | **LinkedHashSet** | **TreeSet** |
| HashSet internally uses HashMap for storing objects | LinkedHashSet uses LinkedHashMap internally to store objects | TreeSet uses TreeMap internally to store objects which is implemented by Red-Black Tree. |
| HashSet does not maintain insertion order | LinkedHashSet maintains insertion order of objects | While TreeSet orders the elements according to supplied Comparator. By Default, It’s objects will be placed in their natural ascending order. |
| HashSet gives O(1) complicity for insertion, removing and retrieving objects | LinkedHashSet gives insertion, removing and retrieving operations performance in order O(1). | TreeSet gives performance of order O(log(n)) for insertion, removing and retrieving operations. |
| HashSet performance is better than LinkedHashSet and TreeSet. | The performance of LinkedHashSet is slow to TreeSet. The performance LinkedHashSet is almost similar to HashSet but slower because, LinkedHashSet maintains LinkedList internally to maintain the insertion order of elements | TreeSet performance is better to LinkedHashSet excluding insertion and removal operations because, it has to sort the elements after each insertion and removal operations. |
| HashSet uses equals() and hashCode() methods to compare the objects | LinkedHashSet uses equals() and hashCode() methods to compare it’s objects | TreeSet uses compare() and compareTo() methods to compare the objects |
| HashSet allows only one null objects | LinkedHashSet allows only one null objects. | TreeSet does not allow any null objects. If you insert null objects into TreeSet, it throws NullPointerException |

1. Difference between HashSet and HashMap?

|  |  |
| --- | --- |
| **HashSet** | **HashMap** |
| HashSet is an implementation of Set interface. HashSet internally uses a HashMap object to store objects. | HashMap is an implementation of Map interface |
| HashSet internally uses HashMap to add elements. In HashSet, the argument passed in **add(Object)** method **serves as key** of HashMap and value is a final object **PRESENT** which is a **dummy** created by jdk.  public boolean add(E e) {         return map.put(e, PRESENT)==null; } | There is no concept of dummy value in HashMap |
| HashSet doesn’t allow duplicate values. | It does not allow duplicate keys, but duplicate values are allowed. |
| HashSet requires only one object add(Object o) . | HashMap requires two objects put(K key, V Value) to add an element to HashMap object. |
| HashSet allows only one null object. | It can contain a single null key and multiple null values. |
| HashSet uses the add() method to add elements in the HashSet. | HashMap uses the put() method to add the elements in the HashMap. |
| HashSet is slower than HashMap because the member object is used for calculating hashcode value, which can be same for two objects. | HashMap is faster than HashSet because values are associated with a unique key. |
| HashSet **internally uses HashMap** object to add or store the objects. | HashMap **internally uses** **hashing** to add or store objects. |

1. Difference between HashMap and HashTable?

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| HashMap is non synchronized. It is not-thread safe and can’t be shared between many threads without proper synchronization code. | Hashtable is synchronized. It is thread-safe and can be shared with many threads. |
| HashMap allows one null key and multiple null values. | Hashtable doesn’t allow any null key or value. |
| HashMap object values are iterated by using iterator. | Hashtable is the only class other than Vector which uses enumerator to iterate the values of Hashtable object. |
| The iterator in HashMap is fail-fast iterator | Enumerator for Hashtable is not fail-fast. |
| HashMap is much faster and uses less memory than Hashtable as former is unsynchronized | HashTable is slower than HashMap |
| HashMap is the subclass of the **AbstractMap** class. Although Hashtable and HashMap has different superclasses but they both are implementations of the "Map” abstract data type | Hashtable is a subclass of **Dictionary** class which is now obsolete in Jdk 1.7 ,so ,it is not used anymore. It is better off externally synchronizing a HashMap or using a ConcurrentMap implementation (e.g ConcurrentHashMap). Or use Collections.synchronizedMap() |

1. Difference between HashMap, LinkedHashMap and TreeMap?

|  |  |  |
| --- | --- | --- |
| **HashMap** | **LinkedHashMap** | **TreeMap** |
| It is implemented by an array of LinkedList. | It is implemented by an array of doubly-LinkedList | TreeMap is implemented by a Red-Black Tree. |
| HashMap does not maintain insertion order of Keys | LinkedHashMap maintains insertion order of Keys | While TreeMap orders the Keys according to supplied Comparator. By Default, It’s Keys will be placed in their natural ascending order. |
| HashMap gives O(1) complexity for insertion, removing and retrieving objects | LinkedHashMap gives insertion, removing and retrieving operations performance in order O(1). | TreeMap gives performance of order O(log(n)) for insertion, removing and retrieving operations. |
| HashMap performance is better than LinkedHashMap and TreeMap. | The performance of LinkedHashMap is slow to TreeMap. The performance LinkedHashMap is almost similar to HashMap but slower because, LinkedHashMap maintains LinkedList internally to maintain the insertion order of elements | TreeMap performance is better to LinkedHashMap excluding insertion and removal operations because, it has to sort the elements after each insertion and removal operations. |
| HashMap uses equals() and hashCode() methods to compare the keys | LinkedHashMap uses equals() and hashCode() methods to compare it’s keys | TreeMap uses compare() and compareTo() methods to compare the keys |
| HashMap allows only one null objects | LinkedHashMap allows only one null objects. | TreeMap does not allow any null objects. If you insert null key into TreeMap, it throws NullPointerException |
| Implements Map Interface | Implements Map Interface | Implements Map, SortedMap and NavigableMap Interface |

1. Difference between **HashTable** and **Collections.synchronizedmap(HashMap)**?

Both are synchronized version of collection. Both have synchronized methods inside class. Both are blocking in nature i.e. multiple threads will need to wait for getting the lock on instance before putting/getting anything out of it.

Only thing which separates them is the fact **HashTable is legacy class** promoted into collection framework. It got its own extra features like enumerators.

1. Difference between **HashMap** and **ConcurrentHashMap**?

|  |  |
| --- | --- |
| **HashMap** | **ConcurrentHashMap** |
| HashMap is **not thread-safe** and can’t be used in a multithreaded concurrent environment. | ConcurrentHashMap is **thread-safe** and can be used in a **concurrent environment** without external synchronization. |
| HashMap can be synchronized by wrapping it on **Collections.synchornizedMap(HashMap)** which will return a collection which is almost equivalent to **Hashtable**, where every operation on Map need to lock the entire Map object. | In ConcurrentHashMap, thread-safety is achieved by dividing whole Map into different partitions/segments based upon Concurrency level and only locking particular portion instead of locking the whole Map. |
| HashMap allows key and value to be null. | ConcurrentHashMap does not allow null key/value. It will throw NullPointerException. |
| Performance wise HashMap is better as there is no synchronization. | In case HashMap has to be used in a multi-threaded environment and there is a need to use Collections.SynchronizedMap() method then ConcurrentHashMap() is a better choice as ConcurrentHashMap still gives a chance to more than one thread to access map thus improving performance. |
| **Iterator** provided by HashMap is **fail-fast** as it throws a ConcurrentModificationException if the underlying collection is structurally modified at any time after the iterator is created | **Iterator** provided by ConcurrentHashMap is **fail-safe** which means it will not throw ConcurrentModificationException if the underlying structure is changed during iteration. |
|  | In ConcurrentHashMap performance is further improved by providing read access concurrently without any blocking. Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). |

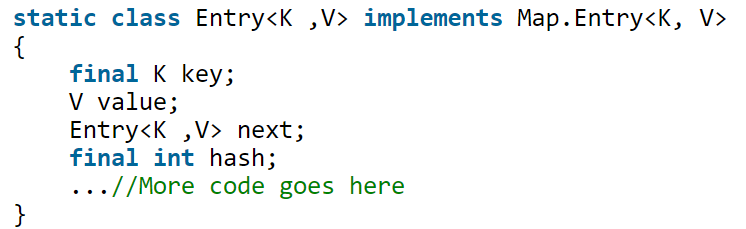
Difference between Iterator and ListIterator?

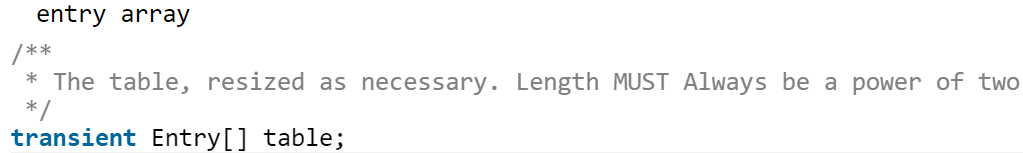
1. Difference between Comparable and Comparator?

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| Comparable interface can be used to provide single way of sorting | Comparator interface is used to provide different ways of sorting. |
| 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. |
| Comparable interface is in java.lang package | Comparator interface is present in java.util package. |
| 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. |
| Comparable provides compareTo() method to sort elements. | Comparator provides compare() method to sort elements. |

1. How HashMap works in java?

HashMap works on the **principle of hashing**, we have put(key, value) and get(key) method for storing and retrieving Objects from HashMap. When we pass Key and Value object to put() method in Java HashMap, HashMap implementation calls hashCode() method on Key object and **applies returned hashcode** into its **own hashing function** to find a bucket location (which is actually an index of the **internal array**, known as the **table**) for storing **Map.Entry** object which has both key and value.





Since the **internal array** of HashMap is **of fixed size**, and if we keep storing objects, at some point of time **hash function** will return **same bucket** location for **two different keys**, this is called **collision in HashMap**. In this case, a **linked list** is formed at that bucket location and a new entry is stored as next node.

In order to retrieve the object using the get(key) method, the hashcode of the key is calculated again and hashmap **applies returned hashcode** into its **own hashing function** to find a bucket location. If there is only one object, then it is returned and that's our value object which we have stored earlier. If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it returns true, Map returns the corresponding value.

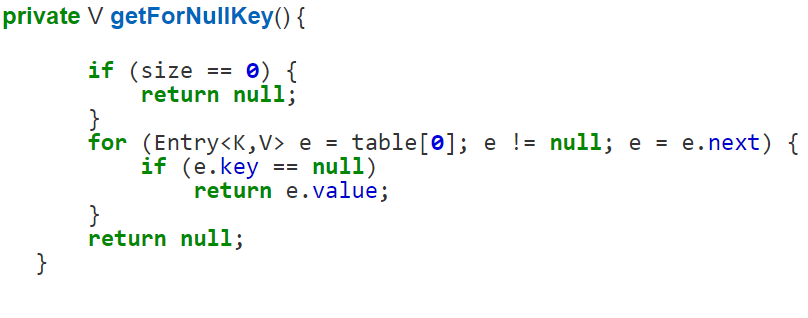
1. What happens in HashMap if the size of the HashMap exceeds a given threshold defined by load factor?

If the size of the **Map exceeds a given threshold defined by load-factor** e.g. if the load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like ArrayList, Java HashMap **re-size itself** by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called **rehashing** because it also applies the hash function to find new bucket location.

However, there is potential **race condition** exists while **resizing HashMap** in Java, if two thread at the same time found that now HashMap needs resizing and they both try to resizing. In the process of resizing of HashMap in Java, **the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket** because Java HashMap doesn’t append the new element at tail instead it appends new element at the head to avoid tail traversing. If race condition happens then you will **end up with an infinite loop**.

1. How **null key** is handled in HashMap? Since equals () and hashCode() are used to store and retrieve values, how does it work in case of the null key?

The null key is handled specially in HashMap, there are two separate methods for that **putForNullKey(V value) and getForNullKey().** Later is offloaded version of get() to look up null keys. **Null keys always map to index 0**. In short, **equals() and hashcode() method are not used in case of null keys** in HashMap.



1. Why immutable objects are considered as better option for Key?

**Immutable object** with proper **equals() and hashcode() implementation** would act as perfect Java HashMap keys and **improve the performance** of Java HashMap **by reducing collision**. **Immutability** also allows **caching their hashcode** of different keys which makes overall **retrieval process very fast** and suggest that **String** and various **wrapper classes** e.g. Integer very good keys in Java HashMap.

1. What are the HashMap changes in JDK1.7 and JDK 1.8?

**In JDK 1.7**

There is some performance improvement done on HashMap and ArrayList from JDK 1.7, which reduce memory consumption. **Due to this empty Map are lazily initialized and will cost you less memory**. Earlier, when you created HashMap e.g. new HashMap () it automatically created an array of default length e.g. 16. After some research, Java team found that most of this Map are temporary and never use that many elements, and only end up wasting memory.

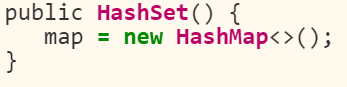
**In JDK 1.8**

HashMap has introduced an improved strategy to deal with high collision rate. Since a **poor hash function** e.g. which always return location of same bucket, can **turn a HashMap into Linked list**, i.e. converting get() method to perform in O(n) instead of O(1) and someone can take advantage of this fact, Java now internally **replace linked list to a binary** **tree** once certain threshold is breached. This **ensures performance or order O(log(n))** even in the worst case where a hash function is not distributing keys properly.

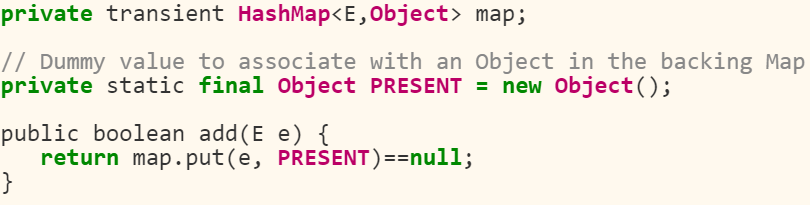
1. How HashSet works internally in java?

**HashSet** uses **HashMap** internally to store its objects. Whenever you create a HashSet object, one HashMap object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add **into HashSet are stored as keys of this HashMap object**. The **value** associated with those keys will be a **constant**.

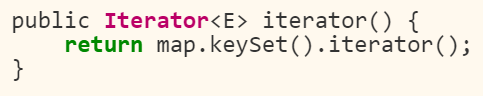
Every constructor of HashSet class internally creates one HashMap object. When you create an object of HashSet in Java, it internally create instance of backup Map with default initial capacity 16 and default load factor 0.75 as shown below :



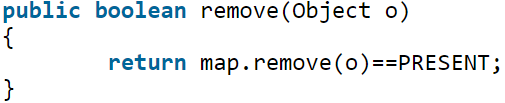
Whenever you insert an element into HashSet using add() method, it actually creates an entry in the internally backing HashMap object with element you have specified as it’s key and constant called “PRESENT” as it’s value. This “**PRESENT**” is defined in the HashSet class as below.



iterator() method from java.util.HashSet class returns iterator for backup Map returned by map.keySet().iterator() method.



remove() method also works in the same manner.



1. Why ConcurrentHashMap is needed when already had Hashtable?

Prior to Java 1.5 if you need a Map implementation, which can be safely used in a concurrent and multi-threaded Java program, then, you only have Hashtable or synchronized Map because HashMap is not thread-safe. With ConcurrentHashMap, now you have a better choice; because not only it can be safely used in the concurrent multi-threaded environment but also provides better performance over Hashtable and synchronizedMap.

Hashtable provides concurrent access to the Map.Entries objects by locking the entire map to perform any sort of operation (update,delete,read,create). Suppose we have a web application , the overhead created by Hashtable (locking the entire map) can be ignored under normal load. But under heavy load , the overhead of locking the entire map may prove fatal and may lead to delay response time and overtaxing of the server.

1. How ConcurrentHashMap works internally?

The main purpose of ConcurrentHashMap is to provide the same functionality as of Hashtable but with a performance comparable to HashMap. The constructor of ConcurrentHashMap looks like this :

**public ConcurrentHashMap (int initialCapacity, float loadFactor, int concurrencyLevel)**

**initial capacity parameter and concurrency level** parameters of ConcurrentHashMap constructor (or Object) are set to 16 by default.

ConcurrentHashMap divides into different parts/segments based on concurrency level and locking only a portion of Map during update operations. Since default concurrency level is 16, and accordingly Map is divided into 16 segments and each segment is governed with a different lock. Thus, instead of a map wide lock, ConcurrentHashMap maintains a list of 16 locks by default, each of which is used to lock on a single bucket of the Map. This indicates that 16 threads can modify the collection at the same time , given ,each thread works on different bucket. So unlike Hashtable, we perform any sort of operation (update, delete ,read ,create) without locking on entire map in ConcurrentHashMap.

**Read/Retrieval operations generally do not block**, so may overlap with update operations. This makes ConcurrentHashMap high performance despite keeping thread-safety intact. Though, it comes with a caveat. **Since update operations** like put(), remove(), putAll() or clear() **is not synchronized**, **concurrent read may not reflect most recent change on Map**. In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries.

Another important point to remember is iteration over CHM, **Iterator returned by keySet** of **ConcurrentHashMap** are **weakly consistent** and they **only reflect state** of ConcurrentHashMap and **certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's** keySet area also **fail-safe** and **doesn’t throw ConcurrentModificationExceptoin**.

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. **Since concurrency level is used for internal sizing and indicate number of concurrent update without contention**, so, if you just have **few writers** or thread to update Map **keeping it low is much better**. ConcurrentHashMap also uses **ReentrantLock** to internally lock its segments.

The segment in CHM is nothing but “a specialized hash table”.

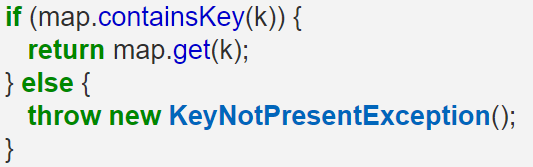


1. When ConcurrentHashMap should be used?

**ConcurrentHashMap** is **best suited** when you have **multiple readers and few writers**. If writers outnumber reader, or **writer is equal to reader**, than **performance** of ConcurrentHashMap effectively **reduces to synchronized map or Hashtable**. Performance of CHM drops, because you **got to lock all portion of Map**, and effectively each writer will wait for another writer, operating on that portion of Map. **ConcurrentHashMap is a good choice for caches**, **which can be initialized during application start up and later accessed my many request processing threads**. As javadoc states, CHM is also a good replacement of Hashtable and should be used whenever possible, keeping in mind, that CHM provides slightly weaker form of synchronization than Hashtable.

1. Why ConcurrentHashMap does not allow null keys and null values ?

**The main reason** that nulls aren't allowed in ConcurrentMaps (ConcurrentHashMaps, ConcurrentSkipListMaps) is that **ambiguities** that may be just barely tolerable in non-concurrent maps can't be accommodated. **The main one is that if map.get(key) returns null, you can't detect whether the key explicitly maps to null vs the key isn't mapped**. **In a non-concurrent map, you can check this via map.contains(key), but in a concurrent one, the map might have changed between calls.**



It might be possible that key k might be deleted in between the get(k) and containsKey(k) calls. As a result , the code will return null as opposed to KeyNotPresentException (Expected Result if key is not present).

1. What are top concurrent collections?

**ConcurrentHashMap** - It provides a concurrent alternative of Hashtable or Synchronized Map classes with aim to support higher level of concurrency by implementing fined grained locking. Multiple reader can access the Map concurrently while a portion of Map gets locked for write operation depends upon concurrency level of Map. ConcurrentHashMap provides better scalability than there synchronized counterpart. Iterator of ConcurrentHashMap are fail-safe iterators which doesn't throw ConcurrencModificationException thus eliminates another requirement of locking during iteration which result in further scalability and performance.

**BlockingQueue** - It makes it easy to implement producer-consumer design pattern by providing inbuilt blocking support for put() and take() method. put() method will block if Queue is full while take() method will block if Queue is empty. Java 5 API provides two concrete implementations of BlockingQueue in form of **ArrayBlockingQueue** and **LinkedBlockingQueue**, both implement FIFO ordering of element. ArrayBlockingQueue is backed by Array and its bounded in nature while LinkedBlockingQueue is optionally bounded.

**CopyOnWriteArrayList and CopyOnWriteArraySet** – CopyOnWriteArrayList provides better concurrency than synchronized List by allowing multiple concurrent reader and replacing the whole list on write operation. Yes, write operation is costly on CopyOnWriteArrayList but it performs better when there are multiple reader and requirement of iteration is more than writing. Since CopyOnWriteArrayList Iterator also don't throw ConcurrentModificationException it eliminates need to lock the collection during iteration.

**Deque and BlockingDeque** - Deque interface is added in Java 6 and it extends Queue interface to support insertion and removal from both end of Queue referred as head and tail. Java6 also provides concurrent implementation of Deque like **ArrayDeque** and **LinkedBlockingDeque**. Deque Can be used efficiently to increase parallelism in program by allowing set of worker thread to help each other by taking some of work load from other thread by utilizing Deque double end consumption property. So if all Thread has their own set of task Queue and they are consuming from head; helper thread can also share some work load via consumption from tail.

**ConcurrentSkipListMap and ConcurrentSkipListSet** - ConcurrentSkipListMap and ConcurrentSkipListSet provide concurrent alternative for synchronized version of SortedMap and SortedSet. For example instead of using TreeMap or TreeSet wrapped inside synchronized Collection, You can consider using ConcurrentSkipListMap or ConcurrentSkipListSet from java.util.concurrent package. They also implement NavigableMap and NavigableSet to add additional navigation method

How do you create **Immutable** Class?

Immutable objects are instances whose state doesn’t change after it has been initialized. An immutable class is **good for caching purpose** because you don’t need to worry about the value changes. Other benefit of immutable class is that it is inherently **thread-safe**, so you don’t need to worry about thread safety in case of multi-threaded environment.

To create an immutable class in java, you have to do following steps.

* Declare the **class as final** so it can’t be extended.
* Make **all fields private** so that direct access is not allowed.
* **Don’t provide setter methods** for variables
* **Make all mutable fields final** so that it’s value can be assigned only once.
* **Initialize all the fields via a constructor performing deep copy**.
* **Perform cloning of objects in the getter methods to return a copy** rather than returning the actual object reference.

Difference between Abstract Classes and Interface?

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| Abstract Class can have both abstract and concrete methods. | Interface provides absolute abstraction and can’t have any method implementations.  Since Java 8 interfaces can contain default and static methods. “default” is mostly used for backward compatibility. Since Java 9 interface can also contain “private” methods (commonly used when two default methods in an interface share common code.) |
| Methods can be implemented as well as remain abstract in the Abstract class. The subclass which extends the abstract is responsible to provide the implementation for the abstract methods. | An interface **is just the declaration of methods,** **it’s not the implementation.** These methods are defined by the classes that implement the interface. It’s a contract between the class and outside world which is enforced by compiler. |
| Abstract class also define contract, but it can provide other methods implementations for subclasses to use. | Interfaces are used **to define contract for the subclasses** |
| The subclass of abstract class must provide the implementation for all of the abstract methods otherwise subclass must be declared abstract. | Class needs to use **implements** keyword to implement the methods of interface. |
| Abstract class can have constructors, but it can’t be instantiated. | Interface doesn’t contain constructor and it can’t be instantiated. |
| Abstract classes **methods** can have access modifiers **as public, private, protected, static** | By default, **all the methods** in the interface are **public abstract**. |
| Abstract class can have **final, non-final, static and non-static variables**. | Interface **has only static and final variables**. |
| It can **extend only one parent class, but it can implement multiple interfaces.** | An interface **can extend another interface**. |
| Abstract class **provides the partial abstraction**. | Interfaces **are more flexible** and can **deal with more stress on your design than implementation**. Interface primary purpose is abstraction, decoupling “what” from “how” |
| Abstract class should be used when you have requirement for base class to **provide the default implementation of certain methods but other methods should be open to being overridden by child classes.** | Interface can be used **when the same behavior needs to be added in unrelated classes** such as Comparable interface used by many classes. |
| The purpose of abstract class is to **provide a common definition of a base class that multiple derived classes can share.** | It can also be used **when you want to specify/declare the behavior but not concerned about who implement it**.  It can also be used to separate different behavior. |
| **We can run an abstract class if it has main() method.** | **We can’t run an interface because they can’t have main method implementation.** |

Which is better design interface or Abstract Class?

Whether to choose between Interface or abstract class for providing a contract for subclasses is a design decision and depends on many factors. Let’s see when Interfaces are the best choice and when can we use abstract classes.

1. Java **doesn’t support multiple class level inheritance**, so every class can extend only one superclass. But a class **can implement multiple interfaces**. So, most of the times Interfaces are a good choice for providing the base for class hierarchy and contract. Also **coding in terms of interfaces is one of the best practices** for coding in java.
2. If **there are a lot of methods in the contract**, **then abstract class is more useful** because we can provide a default implementation for some of the methods that are common for all the subclasses. Also if subclasses don’t need to implement a particular method, they can avoid providing the implementation but in case of interface, the subclass will have to provide the implementation for all the methods even though it’s of no use and implementation is just empty block.
3. **If our base contract keeps on changing then interfaces can cause issues because we can’t declare additional methods to the interface without changing all the implementation classes**, with the abstract class we can provide the default implementation and only change the implementation classes that are actually going to use the new methods.

**Use Abstract Class and Interface both**

**Using interfaces and abstract classes together is the best approach to design a system**. For example, in JDK **java.util.List** is an interface that contains a lot of methods, so there is an abstract class **java.util.AbstractList** that provides a skeletal implementation for all the methods of List interface so that any subclass can extend this class and implement only required methods.

**We should always start with an interface** as the base and define methods that every subclass should implement and then if there are some methods that only certain subclass should implement, we can extend the base interface and create a new interface with those methods.

The subclasses will have the option to choose between the base interface or the child interface to implement according to its requirements.

If the number of methods grows a lot, it’s not a bad idea to provide a skeletal abstract class implementing the child interface and providing flexibility to the subclasses to choose between interface and an abstract class.

**Generics**

What are **Generics**? Advantages of Generics?

In a nutshell, **generics enable types** (classes and interfaces) **to be parameters when defining classes, interfaces and methods**. Much like the more familiar formal parameters used in method declarations, type parameters **provide a way for you to re-use the same code with different inputs**. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

**Advantages of Generics**

* **Stronger type checks at compile time**. A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. **Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.**
* **Elimination of casts**

List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* **Enabling programmers to implement generic algorithms**. By using generics, programmers can implement generic algorithms **that work on collections of different types**, **can be customized, and are type safe and easier to read**.

**Bounded Type Parameters**

**Type parameters can be bounded. Bounded means “restricted “, we can restrict types that can be accepted by a method.** For example, we can specify that a method accepts a type and all its subclasses (upper bound) or a type all its superclasses (lower bound).

To declare an **upper bounded type,** we use the **keyword extends** after the type followed by the upper bound that we want to use.



The keyword **extends** is used here to mean that the type **T extends the upper bound in case of a class** or **implements an upper bound in case of an interface**.

**Multiple Bounds**

A type can also have multiple upper bounds as below. **If one of the types that are extended by T is a class (i.e Number), it must be put first in the list of bounds**. **Otherwise, it will cause a compile-time error.**



**Wildcards with Generics**

Wildcards are represented by the question mark in Java “?” and **they are used to refer to an unknown type.** Wildcards are particularly useful when using generics and can be used as a parameter type.

**Common Misunderstanding with Generics**

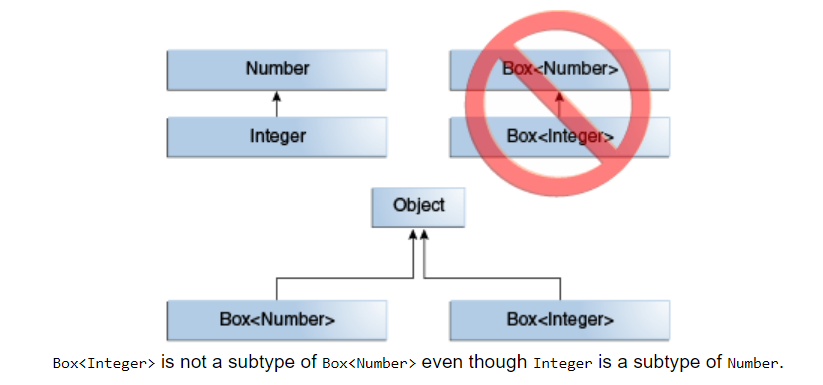
**Note**: - It is known that **Object** is **the supertype of all Java classes**, however, **a collection of Object is not the supertype of any collection**. For example, a List<Object> is not the supertype of List<String> and assigning a variable of type List<Object> to a variable of type List<String> will cause a compiler error.

Now consider the following method:

public void boxTest(Box<Number> n) { /\* ... \*/ }

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>. But what does that mean? Are you allowed to pass in Box<Integer> or Box<Double>, as you might expect? The answer is "no", because Box<Integer> and Box<Double> are not subtypes of Box<Number>.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



**Generics Upper Bounded Wildcard**

**You can use an upper bounded wildcard 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 achieve this by using an upper bounded wildcard.

**List<? extends Number>**

**Unbounded Wildcard**

The unbounded wildcard type is specified using the wildcard character (?), for example, **List<?>.** This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach:

* If you are writing a method that can be implemented using functionality provided in the Object class.
* When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

**Lower Bound Wildcards**

**Lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type**.

A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: **<? super A>**. You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

**Subtyping using wildcards**

Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number> and, in fact, these two types are not related. The common parent of List<Number> and List<Integer> is List<?>.

In order to create a relationship between these classes so that the code can access Number's methods through List<Integer>'s elements, use an upper bounded wildcard:

List<? extends Integer> intList = new ArrayList<>();

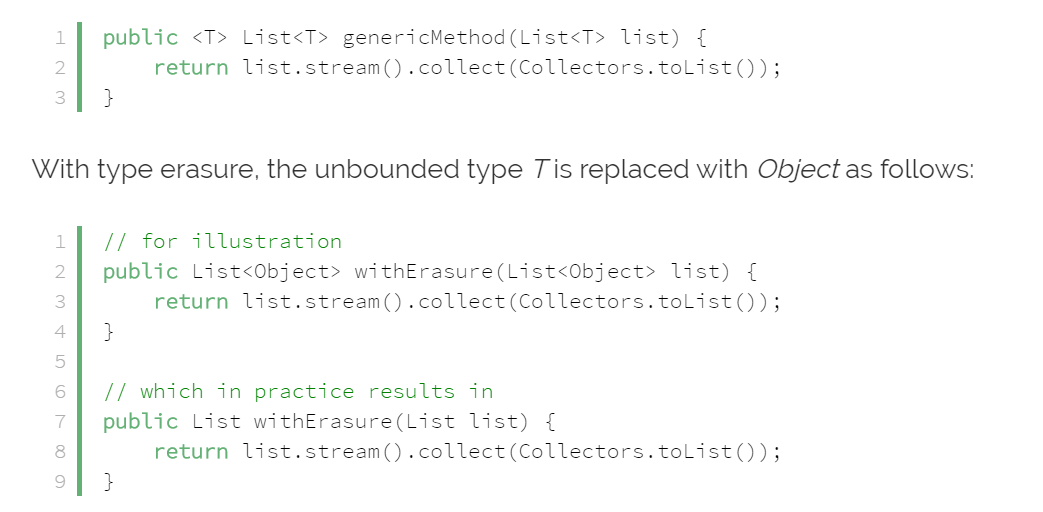
List<? extends Number> numList = intList; **// OK. List<? extends Integer> is a subtype of List<? extends Number>**

**Type Erasure**

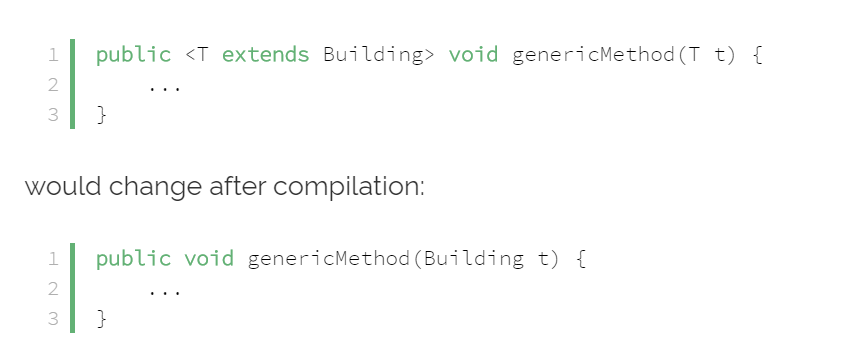
Generics were added to Java to ensure type safety and to ensure that generics wouldn't cause overhead at runtime, **the compiler applies a process called type erasure on generics at compile time**.

Type erasure removes all type parameters and **replaces it with their bounds or with Object if the type parameter is unbounded**. Thus, the bytecode after compilation contains only normal classes, interfaces and methods thus ensuring that no new types are produced. **Proper casting is applied as well to the Object type at compile time**.

This is an example of type erasure:



If the type is bounded, then the type will be replaced by the bound at compile time:



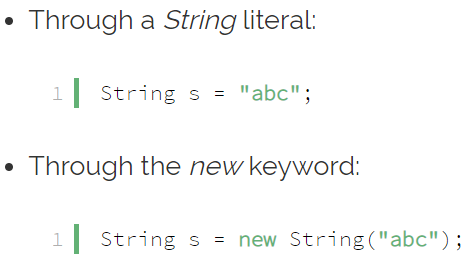
**String**

1. What Is a String in Java?

In Java, a String is represented internally by an array of byte values (or char values before JDK 9). In versions up to and including Java 8, a String was composed of an immutable array of Unicode characters. However, most characters require only 8 bits (1 byte) to represent them instead of 16 bits (char size).

To improve memory consumption and performance, Java 9 introduced compact Strings. This means that if a String contains only 1-byte characters, it will be represented using Latin-1 encoding. If a String contains at least 1 multi-byte character, it will be represented as 2 bytes per character using UTF-16 encoding.

1. How Can We Create a String Object in Java?



1. Is String a Primitive or a Derived Type?

**A String is a derived type** **since it has state and behavior**. For example, it has methods like substring(), indexOf(), and equals(), which primitives cannot have. But, since we all use it so often, it has some special characteristics that make it feel like a primitive:

* While **strings are not stored on the call stack like primitives** are, they are stored in a special memory region called **the string pool**
* Like primitives, we can use the + operator on strings
* And again, like primitives, we can create an instance of a String without the new keyword

1. What are the benefits of Strings being Immutable?

* The string pool is only possible if the strings, once created, are never changed, as they are supposed to be reused
* The code can safely pass a string to another method, knowing that it can't be altered by that method
* Immutably automatically makes this class thread-safe
* Since this class is thread-safe, there is no need to synchronize common data, which in turn improves performance
* Since they are guaranteed to not change, their hashcode can be easily cached.
* Strings are immutable to improve performance and security.

1. How is a String stored in memory?

According to the JVM Specification, **String literals are stored in a runtime constant pool**, which is allocated from the **JVM's method area**. **Although the method area is logically part of the heap memory**, the **specification does not dictate the location, memory size, or garbage collection policies**. It can be implementation-specific.

This runtime constant pool for a class or interface is constructed when the class or interface is created by the JVM.

1. What is the String Constant Pool?

The string pool, also known as the String constant pool or the String intern pool, is a special memory region where the JVM stores String instances. It optimizes application performance by reducing how often and how many strings are allocated:

* The JVM stores only one copy of a particular String in the pool
* When creating a new String, the JVM searches in the pool for a String having the same value
* If found, the JVM returns the reference to that String without allocating any additional memory
* If not found, then the JVM adds it to the pool (interns it) and returns its reference

1. Is String Thread-Safe? How?

Strings are indeed completely thread-safe because they are immutable. Any class which is immutable automatically qualifies for thread-safety because its immutability guarantees that its instances won't be changed across multiple threads. For example, if a thread changes a string's value, a new String gets created instead of modifying the existing one.

1. Difference between String, StringBuffer and StringBuilder?

**Strings are immutable. This means that if we try to change or alter its values, then Java creates an absolutely new String**. For example, if we add to a string str1 after it has been created:



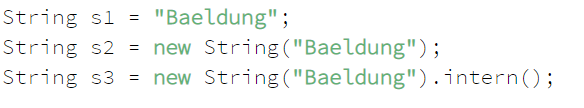
Then the JVM, instead of modifying str1, creates an entirely new String. However, for most of the simple cases, the **compiler internally uses StringBuilder** and optimizes the above code. But, **for more complex code like loops**, it will create an **entirely new String**, **deteriorating performance**. This is where StringBuilder and StringBuffer are useful.

Both **StringBuilder and StringBuffer** in Java create objects that **hold a mutable sequence of characters**. **StringBuffer is synchronized** and therefore thread-safe whereas **StringBuilder is not**. Since the extra synchronization in StringBuffer is typically unnecessary, we can often get a performance boost by selecting StringBuilder.

1. What does String’s **Intern**() method do?

**The method intern() creates an exact copy of a String object in the heap and stores it in the String constant pool, which the JVM maintains**.

**Java automatically interns all strings created using string literals**, but if we create a String using the new operator, for example, String str = new String(“abc”), then Java adds it to the heap, just like any other object. We can call the intern() method to tell the JVM to add it to the string pool if it doesn't already exist there, and return a reference of that interned string:



1. Why is it safer to store Passwords in a Char[] Array rather than a String?

**Since strings are immutable**, **they don't allow modification**. **This behavior keeps us from overwriting, modifying, or zeroing out its contents**, making Strings unsuitable for storing sensitive information. We have to **rely on the garbage collector to remove a string's contents**. Moreover, **in Java versions 6 and below, strings were stored in PermGen**, meaning that once a String was created, it was never garbage collected.

By using a char[] array, we have complete control over that information. We can modify it or wipe it completely without even relying on the garbage collector. Using char[] over String doesn't completely secure the information; it's just an extra measure that reduces an opportunity for the malicious user to gain access to sensitive information.

1. What is the underlying **Character Encoding** for Strings?

According to String's Javadocs for versions up to and including Java 8, **Strings are stored in the UTF-16 format internally**. The **char** data type and java.lang.Character objects are also based on the original Unicode specification, which **defined characters as fixed-width 16-bit entities**.

**Starting with JDK 9**, Strings that contain only **1-byte characters use Latin-1 encoding**, while Strings with at least 1 **multi-byte character use UTF-16 encoding**.

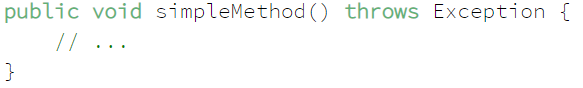
**Exceptions**

1. What is an exception?

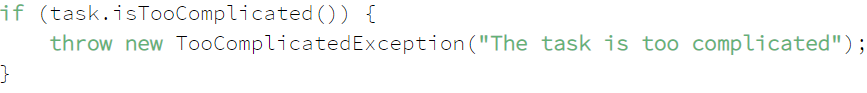
An exception is an abnormal event that occurs during the execution of a program and disrupts the normal flow of the program's instructions.

1. What Is the Purpose of the Throw and Throws keywords?

The **throws** keyword is used to specify that a method may raise an exception during its execution. It enforces explicit exception handling when calling a method:



The **throw** keyword allows us to throw an exception object to interrupt the normal flow of the program. This is most commonly used when a program fails to satisfy a given condition:



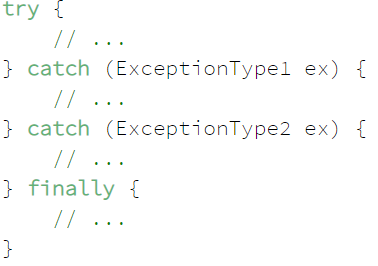
1. How Can You Handle an Exception?

By using a **try-catch-finally** statement:

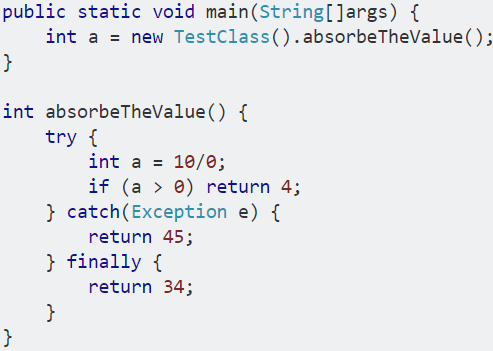
The block of code in which an exception may occur is enclosed in a **try** block. This block is also called “protected” or “guarded” code.

If an exception occurs, the **catch** block that matches the exception being thrown is executed, if not, all catch blocks are ignored.

The **finally** block is always executed after the try block exits, whether an exception was thrown or not inside it.



1. How the return works with try-catch-finally?



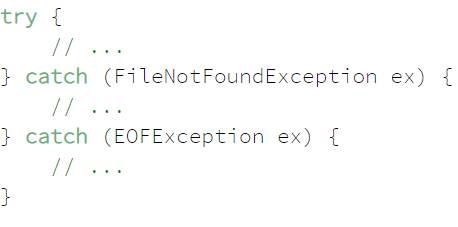
If the return in the try block is reached, it transfers control to the finally block, and the function eventually returns normally (not a throw).

If an exception occurs, but then the code reaches a return from the catch block, control is transferred to the finally block and the function eventually returns normally (not a throw).

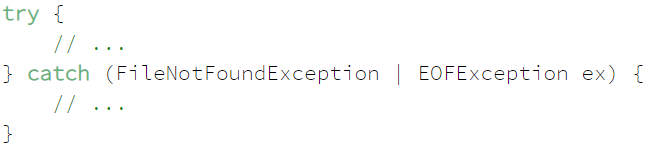
In your example, you have a return in the finally, and so regardless of what happens, the function will return 34, **because finally has the final (if you will) word**.

1. How Can You Catch Multiple Exceptions?

**One way is implementing multiple catch blocks**. Note that, if the exceptions have an inheritance relationship**; the child type must come first and the parent type later**. If we fail to do this, it will result in a compilation error.



**The second is to use a multi-catch block**. This feature first introduced in Java 7; reduces code duplication and makes it easier to maintain.



1. What is the difference Between a Checked and an Unchecked Exception?

A **checked exception** must be **handled within a try-catch block or declared** in a throws clause; whereas an **unchecked exception** is **not required to be handled nor declared**. Checked and unchecked exceptions are also known as compile-time and runtime exceptions respectively. **All exceptions are checked exceptions, except those indicated by Error, RuntimeException, and their subclasses.**

1. What is the difference between an **Exception** and **Error**?

An exception is an **event that represents a condition from which is possible to recover**, whereas error represents **an external situation usually impossible to recover from**.

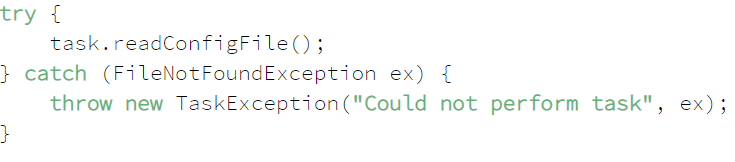
All errors thrown by the JVM are instances of Error or one of its subclasses, the more common ones include but are not limited to:

* **OutOfMemoryError** – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available
* **StackOverflowError** – occurs when the stack space for a thread has run out, typically because an application recurses too deeply
* **ExceptionInInitializerError** – signals that an unexpected exception occurred during the evaluation of a static initializer
* **NoClassDefFoundError** – is thrown when the classloader tries to load the definition of a class and couldn't find it, usually because the required class files were not found in the classpath
* **UnsupportedClassVersionError** – occurs when the JVM attempts to read a class file and determines that the version in the file is not supported, normally because the file was generated with a newer version of Java

Although an error can be handled with a try statement, this is not a recommended practice since there is no guarantee that the program will be able to do anything reliably after the error was thrown.

1. What is Exception Chaining?

Occurs **when an exception is thrown in response to another exception**. This allows us to discover the complete history of our raised problem:



1. Why would you want to subclass an Exception?

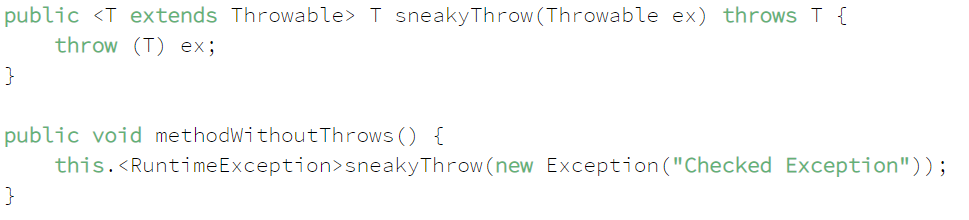
If **the exception type isn't represented by those that already exist in the Java platform, or if you** **need to provide more information to client** **code to treat it in a more precise manner**, then you should create a custom exception.

Deciding whether **a custom exception should be checked or unchecked depends entirely on the business case**. However, as a rule of thumb; if the code using your exception can be expected to recover from it, then create a checked exception otherwise make it unchecked.

Also, you should **inherit from the most specific Exception subclass that closely relates to the one you want to throw**. **If there is no such class, then choose Exception as the parent**.

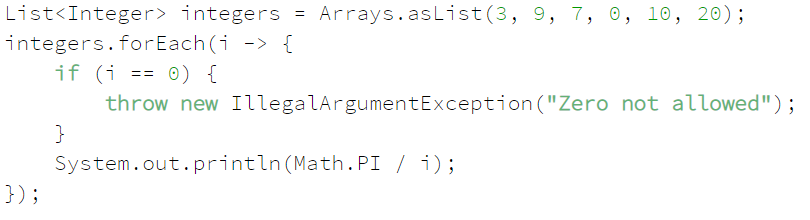
1. Is there any way of throwing a Checked Exception from a method that does not have a Throws Clause?

Yes. We can take advantage of the type erasure performed by the compiler and make it think we are throwing an unchecked exception, when, in fact; we're throwing a checked exception:

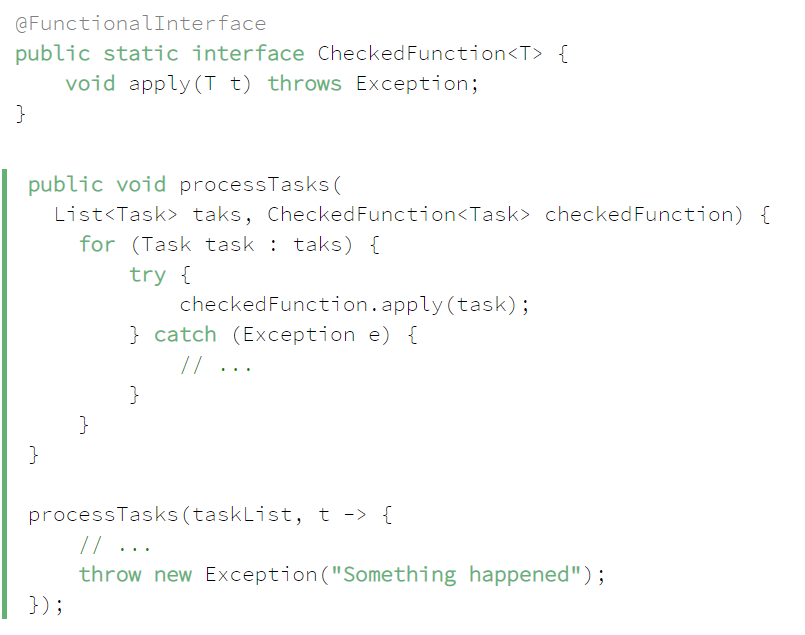


1. Can you throw any exception inside a Lambda Expression's body?

When using a **standard functional interface already provided by Java**, you can **only throw unchecked exceptions because standard functional interfaces do not have a “throws” clause** in method signatures:



However, if you are **using a custom functional interface**, **throwing checked exceptions is possible**:



**Concurrency in Java**

What is Race Condition? Critical Section? Thread Safe?

**What is the difference between Process and Thread?**

**Process** – **A process is a unit of execution that has its own memory**. For example, each instance of JVM runs as a process (true for most of them). When we run **a java console application**, we’re kicking off a process. Term Process and Application is used interchangeably. If one java application is running and we run another one, each application has its own memory space of heap. The heap is not shared between them.

**Thread** – **A thread is a unit of execution within a process**. Each process can have multiple threads. In java, every process (or application) has at least one thread, the **main** thread. Creating a thread doesn’t require as many resources as creating a process. **Every thread created by a process shares the process’s memory and files**. In addition to the process’s memory, or heap, each thread has what’s called a **thread stack**, which is the memory that only that thread can access.

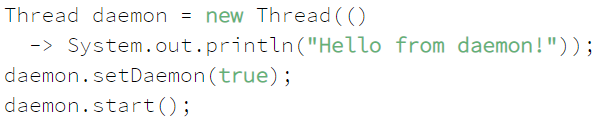
**What is the meaning of Concurrency?**

**Concurrency** – It means an application doing more than one thing at a time. Now, that doesn’t necessarily mean that the application is doing more than one thing at the same time. **It means that progress can be made on more than one task.** Let’s say that an application wants to download data and draw a shape on the screen.

**What Is a Daemon Thread, What Are Its Use Cases? How Can You Create a Daemon Thread?**

**A daemon thread is a thread that does not prevent JVM from exiting**. **When all non-daemon threads are terminated, the JVM simply abandons all remaining daemon threads**. Daemon threads are usually used to carry out some supportive or service tasks for other threads, but you should take into account that they may be abandoned at any time.

To start a thread as a daemon, you should use the setDaemon() method before calling start():



Curiously, if you run this as a part of the main () method, the message might not get printed. This could happen if the main () thread would terminate before the daemon would get to the point of printing the message. You generally should not do any I/O in daemon threads, as they won't even be able to execute their finally blocks and close the resources if abandoned.

**How can we create a Thread in Java?**

There are two ways to create Thread in Java – **first by implementing Runnable interface** and then creating a Thread object from it and **second is to extend the Thread Class**.

**What are different states in lifecycle of Thread?**

The state of a Thread can be checked using the **Thread.getState()** method. Different states of a Thread are described in the **Thread.State enum**. They are:

**NEW** — a new Thread instance that was not yet started via Thread.start()

**RUNNABLE** — a running thread. It is called runnable because at any given time it could be either running or waiting for the next quantum of time from the thread scheduler. A NEW thread enters the RUNNABLE state when you call Thread.start() on it

**BLOCKED** — a running thread becomes blocked if it needs to enter a synchronized section but cannot do that due to another thread holding the monitor of this section

**WAITING** — a thread enters this state if it waits for another thread to perform a particular action. For instance, a thread enters this state upon calling the Object.wait() method on a monitor it holds, or the Thread.join() method on another thread

**TIMED\_WAITING** — same as the above, but a thread enters this state after calling timed versions of Thread.sleep(), Object.wait(), Thread.join() and some other methods

**TERMINATED** — a thread has completed the execution of its Runnable.run() method and terminated

**Can we call run() method of a Thread class?**

If you call the run() method instead of start() method then no new thread will be created and run() method will in the thread which called the run() method.

**Can we call the start () method more than one time?**

If you call start() method twice on a thread then you will get **IllegalThreadStateException**.

**What is sleep() method?**

sleep() method can be **used to pause the execution of current thread** for specified time in milliseconds. sleep() method can be called to give another thread chance to run or maybe we want the thread to wait for another thread to do something.

**What is join() method?**

Join method **allows one thread to wait until another thread completes its execution**. When we invoke the join() method on a thread, the **calling thread goes into a waiting state**. It remains in a waiting state until the referenced thread terminates.

The join() method may also return if the referenced thread was interrupted. In this case, the method throws an **InterruptedException**. Finally, if the referenced thread was already **terminated or hasn't been started**, the call to **join() method returns immediately**.

**What do you understand about Thread Priority?**

Every thread has a priority, usually higher priority thread gets precedence in execution, but it depends on **Thread Scheduler** implementation that is OS dependent. We can specify the priority of thread, **but it doesn’t guarantee that higher priority thread will get executed before lower priority thread**. Thread priority is an int whose value varies from 1 to 10 where 1 is the lowest priority thread and 10 is the highest priority thread.

**How does thread communicate with each other?**

When threads share resources, communication between Threads is important to coordinate their efforts. Object class **wait (), notify() and notifyAll()** methods allows threads to communicate about the lock status of a resource.

**Why thread communication methods wait(), notify() and notifyAll() are in Object class?**

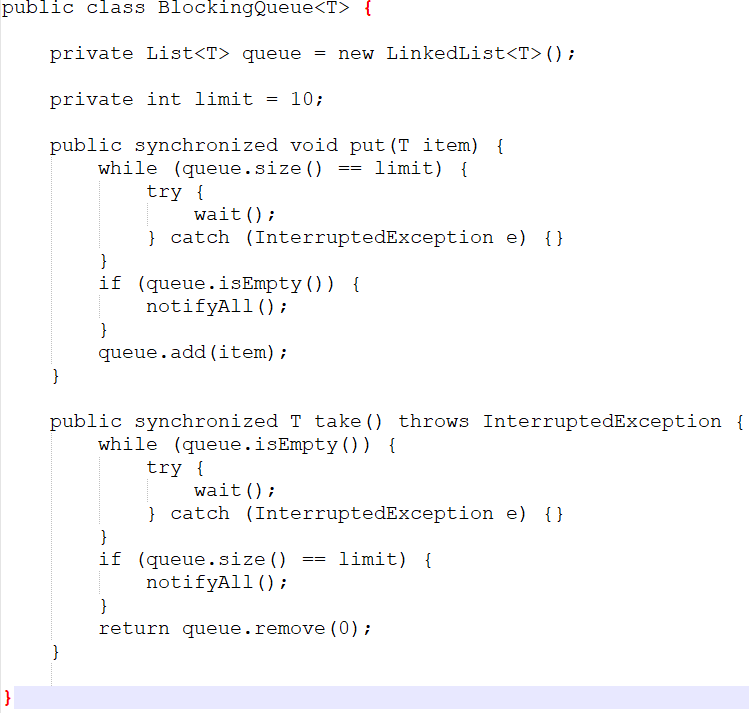
In Java every **Object has a monitor/lock (intrinsic)**. Wait () method is used to wait for the Object monitor and notify () method to notify other threads that Object monitor is free now. **There is no monitor on threads in java** and **synchronization can be used with any Object**, that’s why it’s part of Object class so that every class in java has these essential methods for inter thread communication.

**What Is the Purpose of the Wait, Notify and NotifyAll Methods of the Object Class?**

A thread that owns the object's monitor (for instance, a thread that has entered a synchronized section guarded by the object) may call object.wait() to temporarily release the monitor and give other threads a chance to acquire the monitor. This may be done, for instance, to wait for a certain condition.

When another thread that acquired the monitor fulfills the condition, it may call object.notify() or object.notifyAll() and release the monitor. The notify method awakes a single thread in the waiting state, and the notifyAll method awakes all threads that wait for this monitor, and they all compete for re-acquiring the lock.

The following **BlockingQueue implementation** shows how multiple threads work together via the wait-notify pattern. If we put an element into an empty queue, all threads that were waiting in the take method wake up and try to receive the value. If we put an element into a full queue, the put method waits for the call to the get method. The get method removes an element and notifies the threads waiting in the put method that the queue has an empty place for a new item.



**What Is the Thread’s Interrupt Flag? How Can You Set and Check It? How Does It Relate to the InterruptedException?**

**Interrupt () Method**

There are two ways for a thread to know that it has been interrupted.

- Catch the InterruptedException

- Call the interrupt () method on thread instance

**The interrupt flag, or interrupt status, is an internal Thread flag that is set when the thread is interrupted. To set it, simply call thread.interrupt() on the thread object.**

If a thread is currently inside one of the methods that throw InterruptedException (wait, join, sleep etc.), then this method immediately throws InterruptedException. The thread is free to process this exception according to its own logic.

If a thread is not inside such method and thread.interrupt() is called, nothing special happens. It is thread's responsibility to periodically check the interrupt status using static Thread.interrupted() or instance isInterrupted() method. The difference between these methods is that the static Thread.interrupt() clears the interrupt flag, while isInterrupted() does not.

**Difference between sleep() and wait()?**

The major difference is that wait() releases the lock or monitor while sleep() doesn't releases the lock or monitor while waiting. wait() is used for inter-thread communication while sleep() is used to introduce pause on execution.

**What is Thread Scheduler and Time Slicing?**

**Thread Scheduler** is the Operating System service that **allocates the CPU time** to the **available runnable threads**. Once we create and start a thread, its execution depends on the implementation of Thread Scheduler.

**Time Slicing** is the process to **divide the available CPU time** to the **available runnable threads**. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can’t be controlled by java, so it’s always better to control it from application itself.

**What is context-switching in multi-threading?**

Context Switching is the process of **storing and restoring of CPU state** so that **Thread execution can be resumed from the same point at a later point of time**. Context Switching is the essential **feature for multitasking operating system** and support for multi-threaded environment.

---

**Local variables** are stored in Thread Stack which means each thread will have the own copy of local variable, however **instance variables** are stored in Object’s heap which is shared by all the threads.

**Synchronization**

We can synchronize only method and statement. When a method is synchronized only one thread can access the method at a time. All other threads will be suspended and wait for the thread to complete the execution of the method.

**Intrinsic Lock**

**Every java object has the** **intrinsic lock**. When synchronize is used, thread acquire the object’s intrinsic lock and only one thread can acquire the lock at a time. Hence all other thread will be suspended and will wait for the object’s lock. Once the synchronized block execution is completed by the thread, the lock is released. Now again only one of waiting thread gets the lock to execute synchronized block. Synchronize code at absolute minimum.

**Local Variables can’t be used synchronized to get the locks.**

**We can synchronize static methods using static objects. The lock which is used owned by the class.**

**Synchronization is Reentrant.** If a thread acquires an object’s lock and within the synchronized code it calls the method which is using the same object to synchronized the code then thread can keep executing. i.e. Thread can get the lock which is already owned by the Thread.

**Disadvantages of Synchronization**

- Threads that are blocked waiting to execute synchronized code can’t be interrupted. Once they are blocked, they are stuck their until they get the lock.

- Synchronized block must be within the same method

- We can’t test if an object intrinsic lock is available.

- Also, if the lock is not available, we can’t timeout after we have waited for the lock.

- if multiple threads are waiting for the lock, it’s not first come first serve. It can lead to starvation.

**Locks**

**What is Lock interface in Java Concurrency API? What are its benefits over synchronization?**

Lock interface (**java.util.concurrent.locks**) provides more extensive locking operations than can be obtained using synchronized methods and statements. The advantages of a lock are

* it’s possible to make them fair
* it’s possible to make a thread responsive to interruption while waiting on a Lock object.
* it’s possible to try to acquire the lock, but return immediately or after a timeout if the lock can’t be acquired
* it’s possible to acquire and release locks in different scopes, and in different orders

What are the advantages of java.util.concurrent.lock?

- Lock object can be used to **check if the lock is available** or not using the **tryLock ()** method.

ReentrantLock reentrantLock = new ReentrantLock();

if(reentrantLock.tryLock()){

try{

// critical section

} finally {

reentrantLock.unlock();

}

}

- Using the lock object **we can find out how many threads are already waiting for the lock**. **getQueueLength()**

What is **ReentrantLock**?

A **Reentrant Lock** with the **same** basic behavior and semantics **as the implicit monitor lock** accessed using synchronized methods and statements, **but with extended capabilities**. A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. **A thread invoking lock method will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.** This can be checked using methods **isHeldByCurrentThread(),** and **getHoldCount().**

**The ReentrantLock constructor for this class accepts an optional fairness parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order.** Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting but have smaller variances in times to obtain locks and guarantee lack of starvation.

When using the lock object, we’re responsible for **releasing the lock** by calling the **unlock ()** method, it doesn’t happen automatically. As opposed to intrinsic locks, when the threads exit the synchronized block of code it releases the lock automatically. This is one of **difference between using the intrinsic locks and lock object from java.util.concurrent.locks**

Recommended way of using the **lock object is with** **try and finally clauses**. Critical section of the code should be placed inside the try and lock should be released in finally.

ReentrantLock reentrantLock = new ReentrantLock();

reentrantLock.lock();

try{

// critical section

} finally {

reentrantLock.unlock();

}

What is FairLock?

It is based on first come first serve. Reentrant Lock allows to create the Fair Lock.

Only fairness in acquiring the lock is guaranteed not fairness in Thread scheduling which means a particular thread can still take more time to execute**. It will be on first come first serve and the performance will be impacted to ensure the fairness.**

**Problems with multiple threads or concurrent application**

**What is Deadlock? How to analyze and avoid deadlock situation?**

**DeadLock** – **If two or more threads are blocked on locks and every thread holding the lock that another thread wants.**

Example 1: Thread1 holds the lock1 waiting to acquire lock2, thread2 holding the lock2 and waiting to acquire lock1.

Example 2: Let’s suppose there are two classes that contain synchronized methods, and each class calls a method in other class.

**How can we prevent deadlock**?

- If you can write an application which acquires the lock on single object then deadlock will not happen. This solution is not very practical.

- Multiple **Threads must acquire the locks in the same order** that will stop deadlock occurrence.

- Another solution is to **use lock object with timeout** which will give up the on the lock and stop from deadlock.

- **Avoid Nested Locks, Lock Only What is Required and Avoid waiting indefinitely** are common ways to avoid deadlock situation

To **analyze a deadlock**, we need to look at **the java thread dump** of the application, we need to **look out for the threads with state as BLOCKED** and then the **resources it’s waiting to lock**, every **resource has a unique ID** using which we can find which thread is already holding the lock on the object.

**Thread Starvation**

Thread starvation occurs when a thread never gets the chance to acquire the lock and run because other threads keep taking the lock. It mainly happens due to setting the ThreadPriority which is only suggestion to the operating system, the order of execution can’t be controlled.

**LiveLock**

It is similar to deadlock but instead of threads being blocked, they constantly active and waiting for other threads to complete. Since all the threads waiting for each other to complete hence none of them progress.

**Slipped Condition**

It can occur when a thread is suspended between reading a condition and acting on it. It is a particular type of race condition.

**Atomic Actions**

Which actions are atomic in Java?

1. **Reading and writing reference variables**. For example, myObject1 = myObject2 is atomic.
2. **Reading and writing primitive variables**, except those of type long and double. The JVM may require two operations to read and write long and doubles, and a thread can be suspended between each operation.
3. **Reading and writing** all **variables** declared **Volatile**

We may think that since atomic actions are not impacted by thread interference, there is no need to synchronize them but that’s not true. Because of the way java manages memory, **it’s possible to get the memory inconsistency errors when multiple threads can read and write the same variable.**

**Each thread has a CPU cache, which can contain copies of values that are in main memory**. When running an application, each thread may be running on a different CPU, and each CPU has its own cache. It’s possible for the values in caches to become out of sync with each other, and the value in main memory. – a **memory inconsistency error**.

**What is Volatile Variable?**

This is where volatile variables come in. When we use a non-volatile variable the JVM doesn’t guarantee when it writes an updated value back to main memory. But when we use a volatile variable, the JVM writes the value back to the main memory immediately after a thread updates the value in its CPU cache. It also guarantees that every time a variable reads from a volatile variable, it will get the latest value.

**public volatile int counter;**

If more than one thread can update the value of volatile variable, then we can still get the **memory inconsistency error**. Therefore, we may still need to synchronize the code.

For example, in case of volatile counter (counter++) or long/double value assignment

1. Read the value of counter from memory.
2. Add 1 to counter
3. Write the new value of counter back to memory

A thread can be suspended between any of these steps which can eventually lead to memory inconsistency error.

**java.util.concurrent.atomic** provides classes that we can ensure that reading and writing variable is atomic.

**What is atomic operation? What are atomic classes in Java Concurrency API?**

**Atomic operations are performed in a single unit of task without interference** from other operations. Atomic operations are **necessity in multi-threaded environment to avoid data inconsistency**.

int++ is not an atomic operation. So, by the time one thread read its value and increment it by one, another thread has read the older value leading to the wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 **java.util.concurrent.atomic** provides wrapper classes for int and long that can be used to achieve this atomically without the usage of Synchronization.

**What is ThreadLocal?**

**Java ThreadLocal is used to create thread-local variables**. We know that all threads of an Object share it’s variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.

Every thread has its own ThreadLocal variable and they can use it gets () and set() methods to get the default value or change it’s value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate the state with a thread.

**What is Thread Group? Why it’s advised not to use it?**

ThreadGroup is a class which was intended to provide information about a thread group. ThreadGroup API is weak and it doesn’t have any functionality that is not provided by Thread. It has two main features – to get the list of active threads in a thread group and to set the uncaught exception handler for the thread. But Java 1.5 has added setUncaughtExceptionHandler(UncaughtExceptionHandler eh) method using which we can add uncaught exception handler to the thread. So ThreadGroup is obsolete and hence not advised to use anymore.

**What is Java Thread Dump, how can we get Java Thread dump of a Program?**

A thread dump is a list of all the threads active in the JVM, thread dumps are very helpful in analyzing bottlenecks in the application and analyzing deadlock situations. There are many ways using which we can generate Thread dump – **Using Profiler, Kill -3 command, jstack tool**, etc. I prefer jstack tool to generate thread dump of a program because it’s easy to use and comes with JDK installation. Since it’s a terminal-based tool, we can create a script to generate thread dump at regular intervals to analyze it later on.

**What is Thread Pool? How can we create Thread Pool in Java?**

A thread pool manages the **collection of Runnable threads (worker threads)** and **worker threads execute Runnable (task) from the queue**. **java.util.concurrent.Executors** provide implementation of **java.util.concurrent.Executor** interface to create the thread pool in java.

Executors class provide utility methods for **Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes.**

**What is Executors Framework?**

In Java 5, Executor framework was introduced with the **java.util.concurrent.Executor** interface. The Executor framework is a framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies. **Creating a lot many threads with no bounds to the maximum threshold can cause the application to run out of heap memory. So, creating a ThreadPool is a better solution as a finite number of threads can be pooled and reused**. Executor framework facilitate the process of creating Thread pools in java.

**What is Executors Class?**

Executors class provide utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes. Executors class can be used to easily create Thread Pool in java, also this is the only class supporting execution of Callable implementations.

**What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?**

**java.util.concurrent.BlockingQueue** is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

BlockingQueue **doesn’t accept null values** and **throw NullPointerException** if you try to store null value in the queue. BlockingQueue implementations are **thread-safe**. All queuing **methods are atomic in nature** and use **internal locks or other forms of concurrency control**. BlockingQueue interface is part of the Java collections framework and it’s primarily **used for implementing the producer-consumer problem**.

**What is Callable and Future?**

Java 5 introduced **java.util.concurrent.Callable interface** in concurrency package that is similar to Runnable interface **but it can return any Object and able to throw Exception**.

The Callable interface **uses Generics to define the return type of Object**. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. **Callable tasks return java.util.concurrent.Future object.** Using Future, we can **find out the status of the Callable task and get the returned Object**. It provides the get() method that can wait for the Callable to finish and then return the result.

**What is FutureTask Class?**

**FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing.** Most of the time we don’t need to use FutureTask class but it comes real handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements.

**Java Memory Management**

1. What is JRE?

When you download Java, you get the Java Runtime Environment (JRE). The JRE consists of the Java Virtual Machine (JVM), Java platform core classes, and supporting Java platform libraries. All three are required to run Java applications on your computer.

1. What is JDK?

The Java Development Kit (JDK) is a collection of tools for developing Java applications. With the JDK, you can compile programs written in the Java Programming language and run them in a JVM. In addition, the JDK provides tools for packaging and distributing your applications.

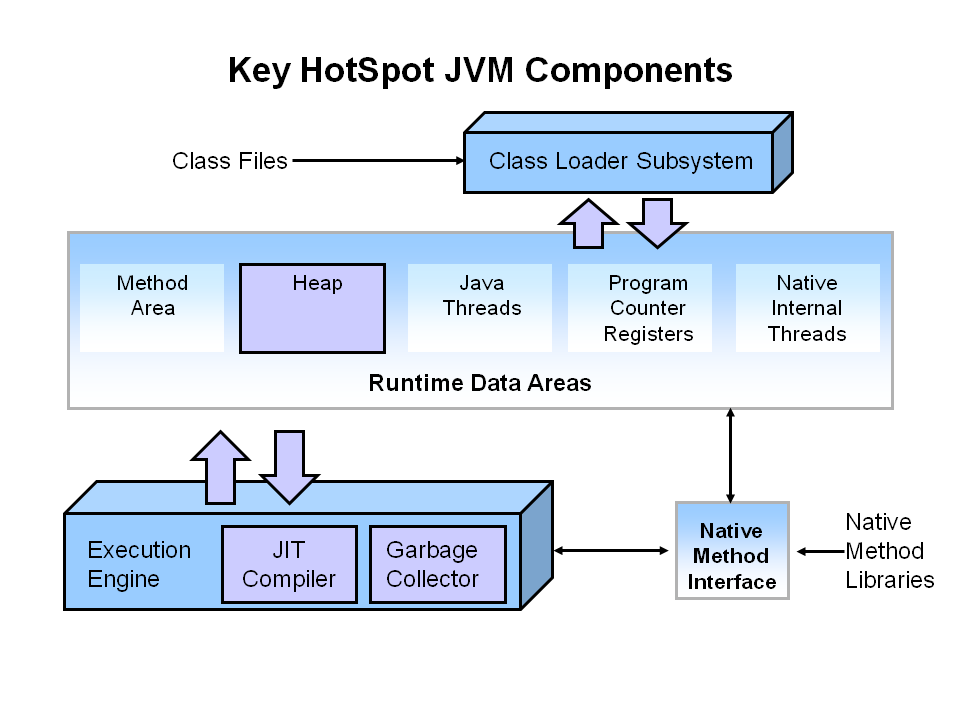
The JDK and the JRE share the Java Application Programming Interfaces (Java API). The Java API is a collection of prepackaged libraries developers use to create Java applications. The Java API makes development easier by providing the tools to complete many common programming tasks including string manipulation, date/time processing, networking, and implementing data structures (e.g., lists, maps, stacks, and queues).

1. What is JVM?

The **Java Virtual Machine (JVM) is an abstract computing machine**. The JVM is a program that looks like a machine to the programs written to execute in it. This way, Java programs are written to the same set of interfaces and libraries. Each JVM implementation for a specific operating system, translates the Java programming instructions into instructions and commands that run on the local operating system. This way, Java programs achieve platform independence.

1. What are the key components of JVM architecture that relate to performance?

The key components of the JVM that relate to performance are highlighted in the following image.



There are three components of the JVM that are focused on when tuning performance. The **heap** is where your object data is stored. This area is then managed by the **garbage collector** selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation. The **JIT compiler** also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.

1. What are Stack and Heap? What is stored in each of these memory structures, and how are they interrelated?

The **stack** is a part of memory that contains information about **nested method calls down to the current position in the program**. It also contains **all** **local variables and references to objects on the heap** defined in currently executing methods.

This structure allows the runtime to return from the method knowing the address hence it was called, and also clear all local variables after exiting the method. Every thread has its own stack.

The **heap** is a large bulk of memory intended for allocation of objects. When you create an object with the new keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

1. What does the “**Memory is managed in java**” mean?

In Java, a developer does not need to explicitly allocate and deallocate memory – the JVM and more specifically the Garbage Collector – has the duty of handling memory allocation so that the developer doesn't have to.

This is contrary to what happens in languages like C where a programmer has direct access to memory and literally references memory cells in his code, creating a lot of room for memory leaks.

1. What is **Garbage Collection** and what are its advantages and disadvantages?

Garbage collection is the **process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object**, or a referenced object, means that some part of your program still maintains a pointer to that object. **An unused object**, or unreferenced object, is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed.

**Advantages**

The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deallocation from us so that we can focus on solving the problem at hand.

**Disadvantages**

Whenever the garbage collector runs, it has an **effect on the application's performance**. This is because all other threads in the application have to be stopped to allow the garbage collector thread to effectively do its work. Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using different GC algorithms.

1. When does an object become eligible for Garbage Collection?

An object becomes eligible for Garbage collection if it cannot be reached through any reference from the stack or by any static references.

* **If the object is not reachable by any live reference and all of its references are null.**
* **Cyclic dependencies without any live external reference are also eligible for GC.**

So, if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.

* **Another obvious case is when a parent object is set to null.**

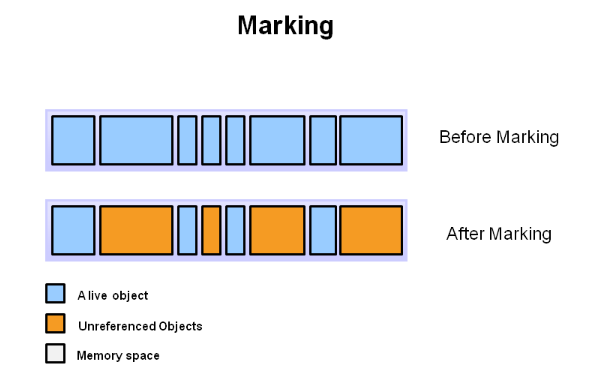
When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

1. How does Garbage Collection works?

GC works in two simple steps known as **Mark and Sweep**

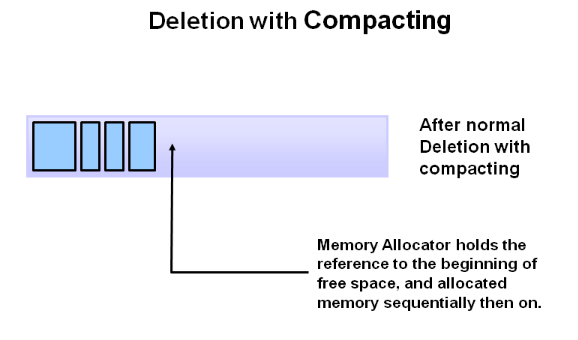
**Marking**

This is where the garbage collector identifies which object are in use and which are not. All objects are scanned in the marking phase to make this determination. **This can be a very time-consuming process if all objects in a system must be scanned**.



**Sweeping**

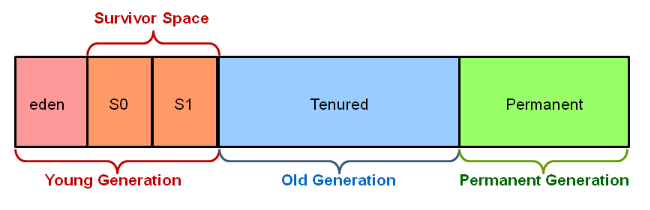
This step removes objects identified during the “marking” phase to free space. It can either do the Normal Deletion or Deletion with Compacting (By moving referenced object together, this makes new memory allocation much easier and faster.)



1. Why is Generational Garbage Collection?

As stated earlier, **having to mark and compact all the objects in a JVM is inefficient**. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that **most objects are short lived**.

Therefore, the heap is broken up into smaller parts or generations. The heap parts are: **Young Generation, Old or Tenured Generation, and Permanent Generation**



**Young Generation -** The Young Generation is where **all new objects are allocated and aged**. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

**Stop the World Event** - All **minor garbage collections** are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

**Old Generation** - The Old Generation is used to **store long surviving objects**. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

Major garbage collection are **also Stop the World events**. Often a major collection is much slower because it involves all live objects. So, for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a **major garbage collection is affected by the kind of garbage collector** that is used for the old generation space.

**Permanent Generation** - The Permanent generation contains **metadata** required by the JVM **to describe the classes and methods** used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed, and space may be needed for other classes. The **permanent generation is included in a full garbage collection**.

1. How does Generational Garbage Collection works?
2. First, any **new objects** are allocated to the **eden space**. Both survivor spaces start out empty.
3. When the **eden space** fills up, a **minor garbage collection** is triggered.
4. **Referenced objects** are moved to the **first survivor space**. Unreferenced objects are deleted when the eden space is cleared.
5. At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the **second survivor space (S1).** In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.
6. At the next minor GC, the same process repeats. However, this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.
7. After a minor GC, when **aged objects** reach a certain age **threshold** (8 in this example) they are promoted from **young generation to old generation**.
8. As minor GCs continue to occur objects will continue to be promoted to the old generation space.
9. So that pretty much covers the entire process with the young generation. Eventually, a **major GC** will be performed **on the old generation** which cleans up and compacts that space.
10. What are the different Java Memory switches?

**Switch** **Description**

-Xms Sets the initial heap size for when the JVM starts.

-Xmx Sets the maximum heap size.

-Xmn Sets the size of the Young Generation.

-XX:PermSize Sets the starting size of the Permanent Generation.

-XX:MaxPermSize Sets the maximum size of the Permanent Generation

1. What are the different types of **Garbage Collectors** or implementation?

JVM has four types of GC implementations:

* Serial Garbage Collector
* Parallel Garbage Collector
* CMS Garbage Collector
* G1 Garbage Collector

**Serial GC** - The serial collector is the default for client style machines in Java SE 5 and 6. With the serial collector, both minor and major garbage collections are done serially (using a single virtual CPU). In addition, it uses a mark-compact collection method.

The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines.

Another popular use for the Serial GC is in environments where a high number of JVMs are run on the same machine. In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer. And the Serial GC fits this trade-off nicely.

To enable the Serial Collector use:

**-XX:+UseSerialGC**

**Parallel GC** - The parallel garbage collector uses multiple threads to perform the young generation garbage collection. By default, on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection.

The Parallel collector is also called a **throughput collector**. Since it can use multilple CPUs to speed up application throughput. This collector should be used when a lot of work need to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

**-XX:+UseParallelGC**

**Concurrent Mark Sweep (CMS) GC** - The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads. Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.

**Note**: CMS collector on young generation uses the same algorithm as that of the parallel collector.

**-XX:+UseConcMarkSweepGC**

**G1 GC** - The Garbage First or G1 garbage collector is available in Java 7 and is designed to be the **long term replacement for the CMS collector**. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector that has quite a different layout from the other garbage collectors described previously.

**-XX:+UseG1GC**

Refer - <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

1. How do you trigger the Garbage Collection from Java code?

You, as Java programmer, cannot force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.

Before removing an object from memory garbage collection thread invokes **finalize()** method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.

Additionally, there are methods like **System.gc() and Runtime.gc()** which is used to send request of Garbage collection to JVM but it’s not guaranteed that garbage collection will happen.

1. Is it possible to <Resurrect> an object that became eligible for Garbage Collection?

When an object becomes eligible for garbage collection, the **GC has to run the finalize method on it**. The **finalize method is guaranteed to run only once**, thus the **GC flags the object as finalized** and gives it a rest until the next cycle.

In the **finalize method** you can technically **“resurrect” an object**, for example, by **assigning it to a static field**. The object would **become alive** again and non-eligible for garbage collection, so the GC would not collect it during the next cycle.

The object, however, would be **marked as finalized**, so when it would become **eligible again**, the **finalize** method would **not be called**. In essence, you can turn this “**resurrection**” trick **only once for the lifetime of the object**. Beware that this ugly hack should be used only if you really know what you're doing — however, understanding this trick gives some insight into how the GC works.

1. Difference between **PermGen** and **Metaspace**?

Starting with Java 8, the **Metaspace replaces the PermGen** – bringing some substantial changes.

**PermGen (Permanent Generation)** is a special heap space separated from the main memory heap. The JVM keeps track of loaded class **metadata** in the PermGen. Additionally, the JVM stores all the **static content** in this memory section. This includes all the static methods, primitive variables, and references to the static objects. Furthermore, it contains data about bytecode, names and JIT information. **Before Java 7, the String Pool was also part of this memory**.

The default maximum memory size for 32-bit JVM is 64 MB and 82 MB for the 64-bit version.

However, we can change the default size with the JVM options:

**-XX:PermSize=[size]** is the initial or minimum size of the PermGen space

-**XX:MaxPermSize=[size]** is the maximum size

With its limited memory size, PermGen is involved in **generating the famous OutOfMemoryError**. Simply put, the **class loaders aren't garbage collected** properly and, as a result, generated a memory leak.

**Metaspace** is a new memory space – starting from the Java 8 version; it has replaced the older PermGen memory space. The most significant difference is how it handles the memory allocation.

As a result, **this native memory region grows automatically by default**. Here we also have new flags to tune-up the memory:

* MetaspaceSize and MaxMetaspaceSize – we can set the Metaspace upper bounds.
* MinMetaspaceFreeRatio – is the minimum percentage of class metadata capacity free after garbage collection
* MaxMetaspaceFreeRatio – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space

Additionally, the garbage collection process also gains some benefits from this change. **The garbage collector now automatically triggers cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size**. Therefore, with this improvement, JVM reduces the chance to get the OutOfMemory error. Despite all of this improvements, we still need to monitor and tune up the metaspace to avoid memory leaks.

1. Describe **Strong, Weak, Soft and Phantom References** and their role in Garbage Collection?

By default, every object we create in a Java program is strongly referenced by a variable:

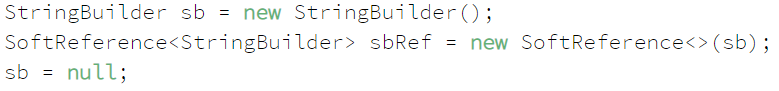


In the above snippet, the new keyword creates a new StringBuilder object and stores it on the heap. The variable sb then stores a **strong reference** to this object. What this means for the garbage collector is that the particular StringBuilder object is not eligible for collection at all due to a strong reference held to it by sb. The story only changes when we nullify sb like this:



After calling the above line, the object will then be eligible for collection.

We can change this relationship between the object and the garbage collector by explicitly wrapping it inside another reference object which is located inside **java.lang.ref package**. A **soft reference** can be created to the above object like this:



In the above snippet, we have created two references to the StringBuilder object. The first line creates a strong reference sb and the second line creates a soft reference sbRef. The third line should make the object eligible for collection, but the garbage collector will postpone collecting it because of sbRef. The story will only change when memory becomes tight and the JVM is on the brink of throwing an OutOfMemory error. In other words, objects with only soft references are collected as a last resort to recover memory.

A **weak reference** can be created in a similar manner using WeakReference class. When sb is set to null and the StringBuilder object only has a weak reference, the JVM's garbage collector will have absolutely no compromise and immediately collect the object at the very next cycle.

A **phantom reference** is similar to a weak reference and an object with only phantom references will be collected without waiting. However, phantom references are enqueued as soon as their objects are collected. We can poll the reference queue to know exactly when the object was collected.

1. What are the tools available for analyzing java memory?

**Jvisualvm** – comes with java

**VisualGC** – Helps to see the GC generation memory usuage