**Core Java vs Advanced Java**

We have only j2se, j2ee and j2me in java world.

J2SE covers - OOPS concept & implementation, Data types, Expressions (including Lambda), Control Flow (if, else, switch, for, while etc.), Classes, Objects, Inner Classes, Wrapper Classes, Packages, Enum Types, Generics, Strings, Numbers, Collection API, and JDBC API etc.

The fundamentals concepts of java which deals basically with OOPS concepts and their implementation in language **can be called core java** and the remaining can be called **advanced java**.

The **Core Java** comprises the Single Tier Architecture. It comprises util, lang, awt, io and net packages. It covers OOPS Concepts, Wrapper Classes, Special Operators, Data types, Exception Handling, Collection API, and JDBC API. It means "stand-alone" java application.

**Advanced Java** is the next advanced level concept of Java programming. It basically uses Two Tier Architecture i.e Client and Server. It covers the Swings, Socket Programming, AWT, Thread Concepts etc. It relates to specialization in domains such as web, networking, and data base handling. It is used for developing the web based application and enterprise application.

**J2SE vs J2EE vs J2ME**

**J2SE(Java Platform, Standard Edition)**

Also known as **Core Java**, this is the most basic and standard version of Java. It’s the purest form of Java, a basic foundation for all other editions.

It consists of a wide variety of general purpose API’s (like java.lang, java.util etc). J2SE is mainly used to create applications **for Desktop environment**. It consist all the basics of Java the language, variables, primitive data types, Arrays, Streams, Strings, Java Database Connectivity(JDBC) and much more. This is the standard, from which all other editions came out. JVM is developed using J2SE.

**J2ME (Java Platform, Micro Edition)**

Mainly concentrated for the applications running on embedded systems, mobiles and small devices. Targeted to address Constraints - limited processing power, battery limitation, small display etc. They use web compression technologies to reduce network usage and hence avail cheap internet accessibility.

Old Nokia phones, which used Symbian OS, used this technology. Most of the apps, developed for the phones (prior to smartphones era), were built on J2ME platform only (the .jar apps on Nokia app store).

**J2EE(Java Platform, Enterprise Edition)**

The Enterprise version of Java has a much larger usage of Java, like development of web services, networking, server side scripting and other various web based applications. J2EE is a **community driven edition**, i.e. there is a lot of continuous contributions from industry experts, Java developers and other open source organizations.

J2EE uses many components of J2SE, as well as, has many new features of its own like Servlets, JavaBeans, Java Message Services, adding a whole new functionalities to the language. J2EE uses HTML, CSS, JavaScript etc., so as to create web pages and web services. It’s also one of the most widely accepted web development standard.

What distinguishes it from other languages (like .net, php etc.) is the versatility, compatibility and security features.

**Java Card:**

This edition was targeted, to run applets smoothly and securely on smart cards and similar technology. **Portability** and **security** were its main features.

**JavaFX:**

Another edition of Java technology which is now merged with J2SE 8. It is mainly used, to create rich GUI (Graphical User Interface) in Java apps. It replaces Swings (in J2SE), with itself as the standard GUI library. It is supported by both Desktop environment as well as web browsers.

**How does LinkedList work?**

LinkedList => Doubly-linked list implementation of the List, Queue and Deque interfaces, as Deque extends the Queue interface. Implements all optional list operations, and permits all elements (including null).

**This implementation is not synchronized.** It *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the list (best done at creation time).

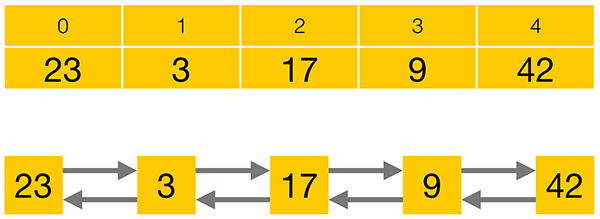
List list = Collections.synchronizedList(new LinkedList(...));

The iterators returned by this class's iterator and listIterator methods are ***fail-fast*:** if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html).

It's a series of instances of a private nested class Entry, which has next, previous and element references. LinkedList is a chain of entities in which every entity knows about next-one, so get(index) operation requires iterating over this chain with counter. But this list optimized for adding and deleting by position.

**Vs** **Linked List data structure:** it is an abstract concept, independent of any specific programming language. The LinkedList Java class is a concrete implementation of this abstract concept.

**Vs ArrayList:** it is based on an Array data structure, while LinkedList is based on a Doubly Linked List data structure:



Compared to a LinkedList, storing elements in an ArrayList consumes less memory and generally gives faster access times. Adding or removing elements is usually faster for a LinkedList, but the performance loss for iterating to the correct position often prevails over the performance gain in adding or removing an element.

LinkedList also implements the Queue and the Deque interfaces which give it some additional functionality over ArrayList.

Queue interface consists of three simple operations:

* **add an element** to the end of the Queue
* **retrieve an element** from the front of the Queue, without removing it
* **retrieve and remove an element** from the front of the Queue.

Queue interface offers each of its operations in two flavours –**one method that will throw an Exception, and one that will return a special value in certain cases**:

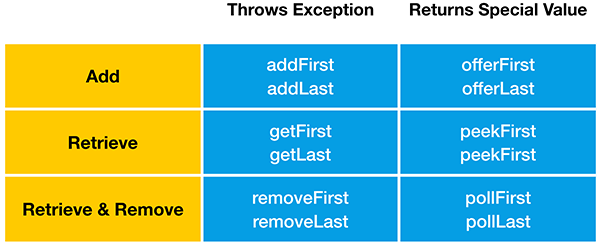


“**add**” will throw an Exception when the Queue is full, while “**offer**” will return false in this case. LinkedList, like most Queue implementations, has an unlimited capacity, so it will never be full. **ArrayBlockingQueue**, on the other hand, is a Queue implementation that has a limited capacity.

**retrieve** (get) an element from the front of the Queue, without removing it.

Finally, you can retrieve and remove an element from the front of the Queue. If the Queue is empty, **remove** will throw an Exception, while **poll** will return false.

**Deque** is the short form of “Double Ended Queue”, so it is a Queue that can be accessed from either end.



LinkedList has an unlimited capacity, so it will never be full. **LinkedBlockingDeque**, on the other hand, is a Deque implementation that may have a limited capacity.

**Stack data structure:** Deque interface also supports the methods of the Stack data structure, “**push**” “**peek**” and “**pop**”. Therefore java.util.LinkedList can also be used as Stack.

“**push**” adds an element to the top of the Stack. Equivalent to “addFirst” method

“**peek**” retrieves but does not remove. Equivalent to “peekFirst” method

“**pop**” retrieves and removes an element from the top of the Stack. Equivalent to the “removeFirst” method.

**Collections intro:**

A **collection** - sometimes called a **container** - is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

**Collections Framework?**

A collections framework is a **unified architecture for representing and manipulating collections**. All collections frameworks contain the following:

* **Interfaces:** => **abstract data types** that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations:** => **concrete implementations** of the collection interfaces. In essence, they are **reusable** **data structures**.
* **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. In essence, algorithms are reusable functionality.

**Benefits of the Java Collections Framework**

* **Reduces programming effort:** By providing useful data structures & algorithms and by facilitating interoperability among unrelated APIs.
* **Increases program speed and quality:** provides high-performance, high-quality implementations of useful data structures and algorithms.
* **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth.
* **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output.
* **Reduces effort to design new APIs:**  Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
* **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable.

**Collection Interfaces:**

The *core collection interfaces* encapsulate different types of collections:



* A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth.
* Hierarchy consists of two distinct trees - a Map is not a true Collection.
* All core collection interfaces are generic. Ex. public interface Collection<E>... Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

**Collection** - the root of the collection hierarchy. A collection represents a group of objects known as its **elements**. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

**Set** - a collection that **cannot contain duplicate** elements. This interface models the mathematical set abstraction and is used to represent sets, such as the courses making up a student's schedule, or the processes running on a machine.

**List** - an **ordered collection** (sometimes called **a sequence**). Lists **can contain duplicate** elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

**Queue** - a collection used to **hold multiple elements prior to processing**.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

**Deque** - a collection used to hold multiple elements prior to processing.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends.

**Map** - **an object that maps keys to values**. A Map **cannot contain duplicate** **keys**; each key can map to at most one value.

The last two core collection interfaces are merely sorted versions of Set and Map:

**SortedSet** - a Set that maintains its **elements in ascending order**. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

**SortedMap** - a Map that maintains its mappings in **ascending key order**. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

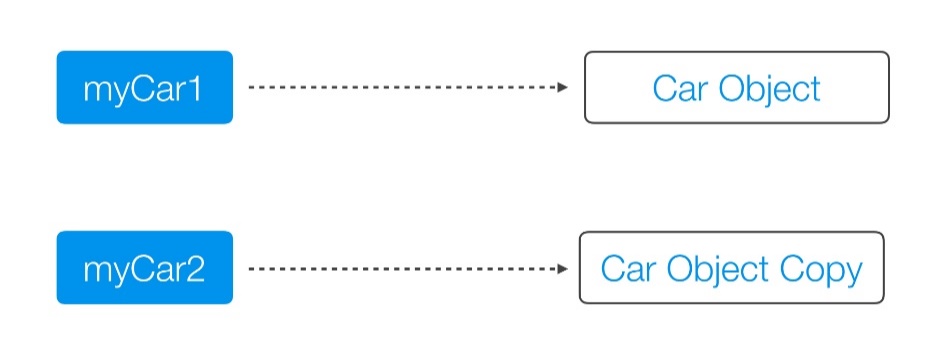
**Difference between shallow cloning and deep cloning of objects?**

**Copy:**

* A **reference copy**, as the name implies, creates a copy of a reference variable pointing to an object.



* An **object copy** creates a copy of the object itself.



Both a **Deep Copy** and a **Shallow Copy** are **types of object copies**. An object copy, usually called a **clone**, is created if we want to modify or move an object, while still preserving the original object.

**A shallow copy** of an object copies the ‘main’ object, but doesn’t copy the inner objects. The **‘inner objects’ are shared** between the original object and its copy. For example, in our Person object, we would create a second Person, but both objects would share the same Name and Address objects.

public class Person {

private **Name** name;

private **Address** address;

public Person(Person originalPerson) {

this.name = originalPerson.name;

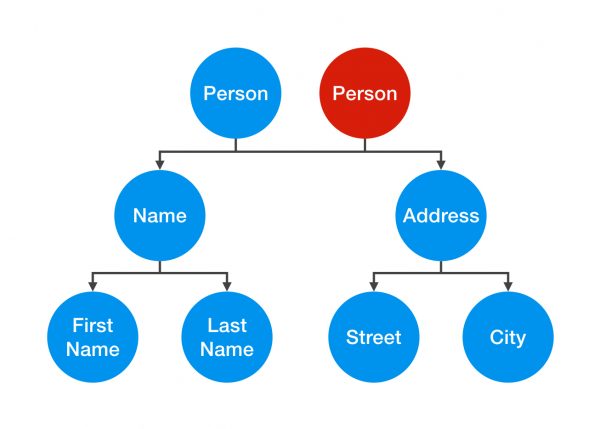
this.address = originalPerson.address;

}

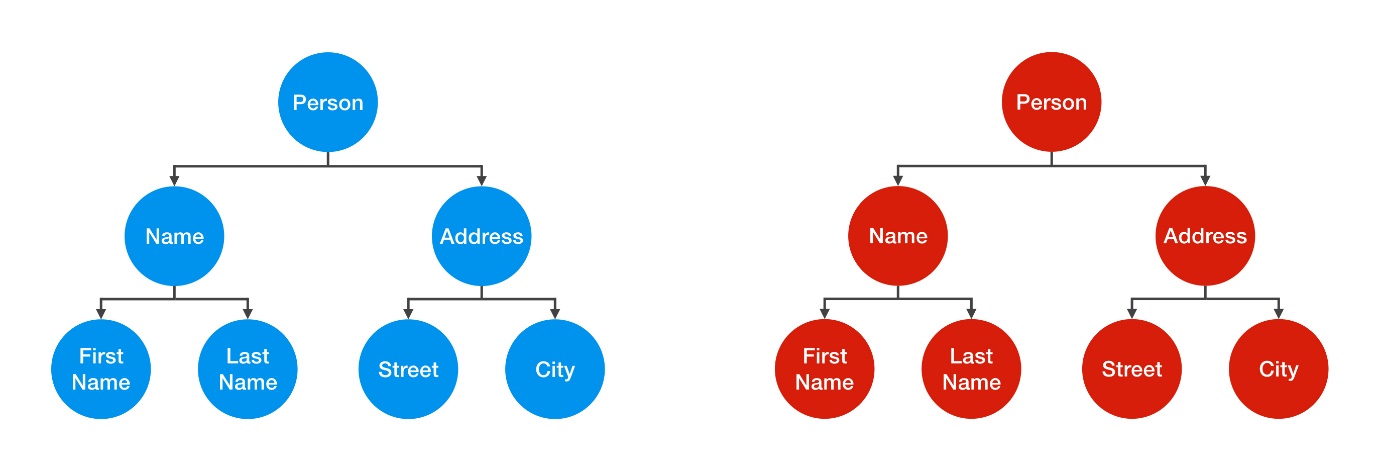
[…]

}

The problem with the shallow copy is that the two objects are not independent. If you modify the Name object of one Person, the change will be reflected in the other Person object.



A **deep copy** is a**fully independent copy of an object**. If we copied our Person object, we would copy the entire object structure.



A change in the Address object of one Person wouldn’t be reflected in the other object.

public class Person {

private Name name;

private Address address;

public Person(Person otherPerson) {

this.name = new Name(otherPerson.name);

this.address = new Address(otherPerson.address);

}

[…]

}

However, that’s not the end of the story. To create a true deep copy, we need to keep copying all of the Person object’s nested elements, until there are only primitive types and “**Immutables**” left.

public class Street {

private String name;

private int number;

public Street(Street otherStreet){

this.name = otherStreet.name;

this.number = otherStreet.number;

}

[…]

}

In Street object, int number is a primitive value and not an object. It’s just a simple value that can’t be shared, so by creating a second instance variable, we are automatically creating an independent copy.

String is an Immutable. Therefore, you can share it without having to create a deep copy of it.

The default behavior of an **object’s clone()** method **automatically yields a shallow copy**. So to achieve a deep copy the classes must be edited or adjusted. **clone()** method returns ‘Object’ as type and you need to explicitly cast back to your original object.

The class which you want to be cloned should implement clone method and overwrite it. It should provide its own meaning for copy or to the least it should invoke the **super.clone().** Also you have to implement **Cloneable marker interface** or else you will get CloneNotSupportedException. When you invoke the super.clone() then you are dependent on the Object class’s implementation and what you get is a shallow copy.

When you need a **deep copy** then you need to **implement it yourself**. When you implement deep copy be careful as you **might fall for cyclic dependencies**. If you don’t want to implement deep copy yourselves then you can **go for serialization**. It does **implements deep copy implicitly and gracefully handling cyclic dependencies**.

**Collections Overview:**

Collection Interfaces - Divided into 2 groups. The most basic interface, java.util.Collection, has the following descendants:

**Can we have interfaces with no defined methods?**

The interfaces with no defined methods act like markers. They just **tell the compiler** that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. **Marker interfaces** are also known as **“tag” interfaces**.

**What is the difference between “==” and equals() method?**

The **== (double equals)** returns true, **if the variable reference points to the same object** in memory. This is called “**shallow comparison**”.

The **equals() method** calls the user implemented equals() method, which compares the object attribute values. The equals() method provides “**deep comparison**” by checking if two objects are **logically equal** as opposed to the shallow comparison provided by the operator ==.

If equals() method does not exist in a user supplied class then the inherited **Object class's equals() method** will be called which evaluates if the references point to the same object in memory. In this case, the **object.equals() works just like the "==" operator**.

**Can I use a Class Object as Key in HashMap?**

Yes. In order to be used as a HashMap key, the class has to implement hashCode() and equals() methods to reflect "equality" of two objects.

If you create two different instances with

Key a = new Key("xyz");

Key b = new Key("xyz");

and expect them to be equal and work in a HashMap, you have to override hashCode() so that it returns the same value in both instances, and equals() returns true when comparing them.

If the object identity is based on the string value, then

@Override

public int hashCode()

{

return theStringValue.hashCode();

}

and

@Override

public boolean equals(Object o)

{

return this.theStringValue.equals(o);

}

should work.

In eclipse, just right click on class and go to Source>Generate hashCode() and equals()… choose the attributes you want to include in these methods and the code will be auto-generated.

After overriding hashcode and equals method, you need to use your object while getting data from hashMap. Ex.

HM.get(new Account(2193,"Uri"));

A great care must be exercised if mutable objects are used as map keys. You should make the keys immutable so they do not change.

Joshua Bloch says on Effective Java:

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

If two objects are equal, their hashcodes must be equal as well.

Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be stored in the collection, and the hashcode is used again to help locate the object in the collection.

Hashing retrieval is a two-step process.

1. Find the right bucket (using hashCode())
2. Search the bucket for the right element (using equals() )

**How does HashMap work?**

It works on the **hashing principle**.

* **Hashing** is the mechanism of **assigning unique code to a variable** or attribute using an algorithm **to enable easy retrieval**. A true hashing mechanism should **always return the same hashCode()** when it is applied to the same object.
* Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap.
* **hashCode() method** of object class returns the memory reference of object in integer form. In HashMap, it is used to calculate the bucket and therefore calculate the index.
* **equals method** is used to check that 2 objects are equal or not. HashMap uses equals() to **compare the key** whether they are equal or not.
* A **bucket** is one **element of HashMap array**. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized.
* **Index Calculation in Hashmap:**

Hash code of key may be large to create an array => may cause outOfMemoryException. Hence, we generate index to minimize the size of array.

index = hashCode(key) & (n-1); where n is number of buckets or the size of array. 16 is the default size. & is bitwise AND operator.

The JAVA HashMap class implements the interface Map<K,V>. The main methods of this interface are:

* V put(K key, V value)
* V get(Object key)
* V remove(Object key)
* Boolean containsKey(Object key)

HashMaps use an **inner class** to store data: the **Entry<K, V>**. This entry is a simple key-value pair with two extra data:

* a reference to another Entry so that a HashMap can store entries like singly linked lists
* a hash value that represents the hash value of the key. This hash value is stored to avoid the computation of the hash every time the HashMap needs it.

Here is a part of the Entry implementation in JAVA 7:

static class Entry<K,V> implements Map.Entry<K,V>

{

final K key;

V value;

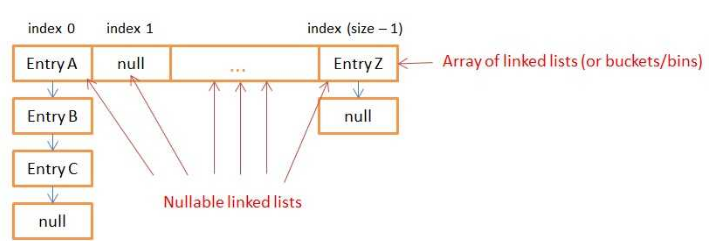
Entry<K,V> next;

final int hash;

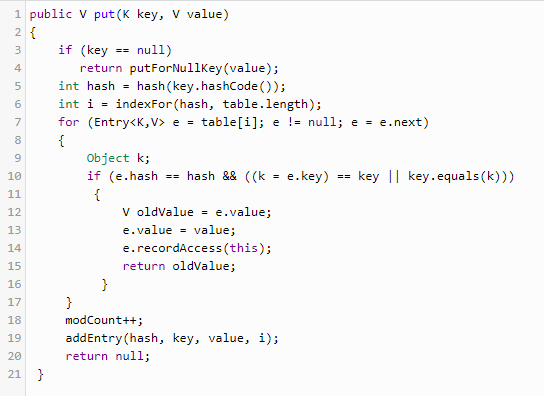
........

}

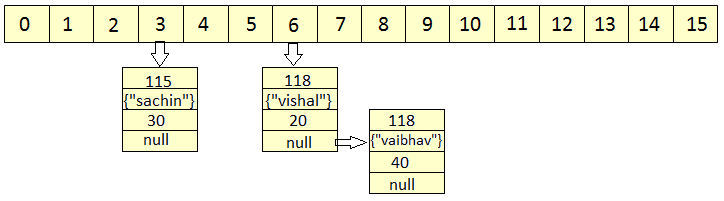
A HashMap stores data into multiple singly linked lists of entries (also called **buckets** or **bins**). All the lists are registered in an array of Entry (Entry<K,V>[] array) and the default capacity of this inner array is 16.



**How Does Put() Method Work Internally?**



* First - If the **given key is null**, it will be stored in the **zero position**. As HashMap also allows null key, and hash code of null will always be 0.
* Then it applies the hashcode to the key .hashCode() by calling the hashcode method. In order to get the value within the limits of an array, the hash(key.hashCode()) is called, which performs some shifting operations on the hashcode. Say 118
* The indexFor() method is used to get the exact location to store the **Entry** object. For 118 hashcode, it is 6.
* Then comes the most important part what happens if two different object has the same hashcode( eg : Aa, BB will have the same hashcode) and will it be stored in the same bucket. Like the next attribute of LinkedList in data structure; the **next attribute in the Entry** class points to the next object. Using this, different objects with the same hashcode will be placed next to each other.
* In the case of the **Collision**, the HashMap checks for the value of the **next attribute** if it is null it inserts the Entry object in that location, if next attribute is not null then it keeps the loop running till next attribute is null then stores the Entry object there.



* HashMap **doesn't allow duplicate keys**, even though when we insert the same key with different values, only the latest value is returned.

HashMap uses equals() method to check for the equality of key. If **key.equals(k)**is true, then it will replace the value object inside the Entry class. Otherwise connect this node object to the previous node object via linked list and both are stored in the same bucket.

* **How Does Get() Method Work Internally?**
* First, it gets the hash code of the key object, which is passed, and finds the bucket location.
* If the correct bucket is found, it returns the value (e.value)
* If no match is found, it returns null.
* **What Happens If Two Keys Have the Same Hashcode?**

The same **collision** resolution mechanism will be used here. key.equals(k) will check until it is true, and if it is true, it returns the value of it.

* HashMap **has the ability to increase its inner array** in order to keep very short linked lists. When you create a HashMap, you can specify an initial size and a loadFactor with the following constructor:

public HashMap(int initialCapacity, float loadFactor)

If you don’t specify arguments, the default initialCapacity is 16 and the default loadFactor is 0.75. The initialCapacity represents to the size of the inner array of linked lists.

Each time you add a new key/value in your Map with put(…), the function checks if it needs to increase the capacity of the inner array.

**Threshold** is **capacity multiplied by load factor** and whenever we try to add an entry, if map size is greater than threshold, HashMap **rehashes** the contents of map into a new array with a larger capacity.

HashMap only increases the size of the inner array, it **doesn’t provide a way to decrease** it.

* **HashTable** implementation is a **thread safe implementation**. But, since all the CRUD methods are synchronized this implementation is very slow.

A smarter implementation of a thread safe HashMap exists since JAVA 5: the **ConcurrentHashMap**. Only the buckets are synchronized so multiples threads can get(), remove() or put() data at the same time.

* Why Strings and Integers are a good implementation of keys for HashMap? Mostly because they are **immutable**! If you choose to create your own Key class and don’t make it immutable, you might lose data inside the HashMap.

<http://coding-geek.com/how-does-a-hashmap-work-in-java/>

* When using a HashMap, you need to find a hash function for your keys that **spreads the keys into the most possible buckets**. To do so, you need to **avoid hash collisions**. The String Object is a good key because of it has good hash function. Integers are also good because their hashcode is their own value.

## Skewed HashMap vs well balanced HashMap

If the hash function of your key is ill-designed, you’ll have a **skew** repartition (no matter how big the capacity of the inner array is). All the put() and get() that use the biggest linked lists of entry will be slow because they’ll need to iterate the entire lists.

Well balanced HashMap has a hash (of the key) function that gives better repartition for entries in the buckets.

**Notes:**

1. In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
2. If key given already exist in HashMap, the value is replaced with new value.
3. hash code of null key is 0.
4. When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

<http://www.geeksforgeeks.org/internal-working-of-hashmap-java/>

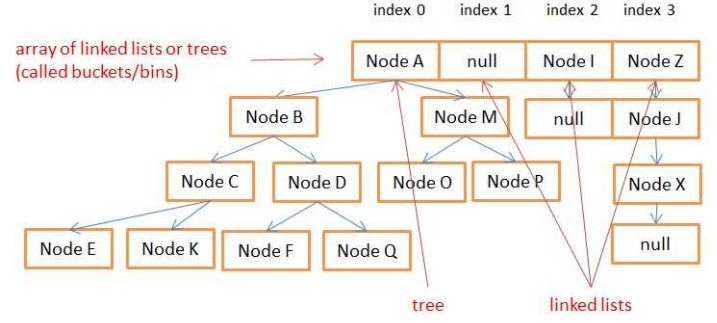
**Java 8 changes to HashMap:**

The performance has been improved by using **balanced trees** instead of linked lists under specific circumstances. It has only been implemented in the classes java.util.HashMap, java.util.LinkedHashMap and java.util.concurrent.ConcurrentHashMap.

In JAVA8, you still have an array but it now stores **Nodes** that contains the exact same information as **Entries**. Nodes can be **extended to TreeNodes**. A TreeNode is a red-black tree structure that stores really more information.

By inheritance, the **inner table can contain** both **Node** **(linked list ) and** **TreeNode** (red-black **tree**). Oracle decided to use both data structures with the following rules:

* If for a given index (bucket) in the inner table there are more than 8 nodes, the linked list is transformed into a red black tree
* If for a given index (bucket) in the inner table there are less than 6 nodes, the tree is transformed into a linked list.



**Use of Volatile Keyword in java:**

Using volatile is yet another **way of making class thread safe** (like synchronized, atomic wrapper). Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

The **values of volatile variable will never be cached** and all writes and reads will be done to and from the main memory.

Each thread has its own stack, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables in its own memory. The volatile keyword is used to say to the jvm "Warning, this variable may be modified in an other Thread". Without this keyword the JVM is free to make some optimizations, like never refreshing those local copies in some threads. The volatile force the thread to update the original variable for each variable. The volatile keyword **could be used on every kind of variable, either primitive or objects**.

**Atomic classes:**

Atomic classes **support lock-free thread-safe programming** on single variables.

Primitive variables can't be part of Synchronized block. Ex. Int, Float, Long, boolean etc. Use java.util.concurrent.atomic.AtomicInteger, AtomicLong etc. instead.

**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.

**count++** is not an atomic operation. So by the time one threads read its value and increment it by one, other thread has read the older value leading to 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 atomic operation **without usage of Synchronization**.

Ex. AtomicInteger method **incrementAndGet()** atomically increments the current value by one.

AtomicInteger count = new AtomicInteger();

count.incrementAndGet();

Atomic classes are **not general purpose replacements** for java.lang.Integer and related classes. They **do not define** methods such as hashCode and compareTo. (Because atomic variables are expected to be **mutated**, they are **poor choices for hash table keys**).

Also atomic operation concurrency classes are assumed to be more efficient than synchronization which involves locking resources.

**Vs. Wrapper class:**

Concept of atomicity comes when something is mutable. We want the operation of modifying a field/variable (could be many steps, read -> update -> write) as atomic operation. So that in multithreaded scenario, there should not be any data corruption. java.util.concurrent.atomic package does it for us. Wrapper classes are immutable, we can't modify it, need to create a new instance.

**Functional interfaces in Java EE:**

Ex. java.lang.Runnable, java.awt.event.ActionListener, java.util.Comparator, java.util.concurrent.Callable - they have only one method declared in their interface definition.

These interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**. And a popular way in which these are used is by creating Anonymous Inner classes using these interfaces.

**With Java 8** the same concept of **SAM interfaces is recreated** and are called Functional interfaces. These can be **represented using Lambda expressions**, Method reference and constructor references. **@FunctionalInterface** can be used for compiler level errors when the interface you have annotated is not a valid Functional Interface. It can also declare the abstract methods from the java.lang.Object class.

Interface can extend another interface and in case the Interface it is extending is functional and it doesn’t declare any new abstract methods then the new interface is also functional. But an interface can have one abstract method and any number of default methods and the interface would **still be called a functional interface**.

Ex.

@FunctionalInterface

**public** **interface** SimpleFuncInterface {

**public** **void** doWork();

**public** String toString();

**public** **boolean** equals(Object o);

}

Another example:

// Use Lambda operator or Arrow Token

// Method followed by Arrow Token and then the code that perform actions

Runnable runnable = () -> System.***out***.println("Welcome to Java 8 : Thread 1");

Runnable runnable2 = () -> {

//Multiliner statements - return is mandatory else can use single liner conditional if

System.***out***.println("Welcome to Java 8 : Thread 2");

}; //Ending with semicolon

**What is a serialVersionUID and why should I use it?**

SerialVersionUID is a unique identifier for each class, JVM uses it to compare the versions of the class ensuring that the same class was used during Serialization is loaded during Deserialization.

Specifying one gives more control, though JVM does generate one if you don't specify.

As per Java docs, "the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization".

**You must declare serialVersionUID because it give us more control**.

When a Serializable class object is serialized Java Runtime associates a serial version no.(called as serialVersionUID) with this serialized object. At the time when you deserialize this serialized object Java Runtime matches the serialVersionUID of serialized object with the serialVersionUID of the class. If both are equal then only it proceeds with the further process of deserialization else throws **InvalidClassException**.

So we conclude that to make Serialization/Deserialization process successful the serialVersionUID of serialized object must be equivalent to the serialVersionUID of the class. In case if programmer specifies the serialVersionUID value explicitly in the program then the same value will be associated with the serialized object and the class.

It is also possible that the environment where the object is serialized is using one JRE (ex: SUN JVM) and the environment where deserialzation happens is using Linux Jvm(zing). In such cases serialVersionUID associated with serialized object will be different than the serialVersionUID of class calculated at deserialzation environment. In turn deserialization will not be successful. So to avoid such situations/issues programmer must always specify serialVersionUID of Serializable class.

**Impact of not defining serialVersionUID in class:**

If we don’t define serialVersionUID in the class, and any modification is made in class, then we won’t be able to deSerialize our class because serialVersionUID generated by java compiler for modified class will be different from old serialized object. And deserialization process will end up throwing **java.io.InvalidClassException** (because of serialVersionUID mismatch).

If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don’t end up throwing InvalidClassException - Simply we need to define **serialVersionUID** in class.

When we Deserialize class (class which has been **modified after Serialization** and also class **doesn’t declare SerialVersionUID**) **InvalidClassException** is thrown.

When we Deserialize class (class which has been modified after Serialization and also class declare SerialVersionUID) its gets DeSerialized successfully.

**Why to use Generics?**

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

The difference is that the inputs to **formal parameters** (to method declarations) are **values**, while the inputs to **type parameters** are **types**.

Generics are nothing but **parameterized types**. Generics helps us to create a single class, which can be useful to operate on multiple data types. A class, interface or a method that operates on a parameterized type is called generics class, interface or method. Generics adds type safety. Remember that generics only works on objects, not primitive types.

Generics allows a type or method to operate on objects of various types while providing **compile-time type safety**, making Java **a fully statically typed language**.

**Benefits over non-generic code:**

* **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.**  
  The following code snippet without generics requires casting:

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 you can define an algorithm once, and you can apply it on any kind of datatype without any additional effort.

Consider the following method, printList:

public static void printList(List<Object> list) {

for (Object elem : list)

System.out.println(elem + " ");

System.out.println();

}

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>. To write a generic printList method, use List<?>:

public static void printList(List<?> list) {

for (Object elem: list)

System.out.print(elem + " ");

System.out.println();

}

Because for any concrete type A, List<A> is a subtype of List<?>, you can use printList to print a list of any type:

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

List<String> ls = Arrays.asList("one", "two", "three");

printList(li);

printList(ls);

The **unbounded wildcard type** is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. Unbounded wildcard is a useful for below scenarios:

* 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.

**What are Generic Types & Generic Methods?**

Generics was added in Java 5 to provide **compile-time type checking** and **removing risk of ClassCastException** that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety.

A generic type is a generic class or interface that is parameterized over types.

Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables. Th e most commonly used type parameter names are:

* E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
* K – Key (Used in Map)
* N – Number
* T – Type
* V – Value (Used in Map)

**Non-generic** Box class

public class Box {

private Object object;

public void set(Object object) { this.object = object; }

public Object get() { return object; }

}

**Generic** Version of the Box Class

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

**Generic methods:**

Generic methods are methods that introduce their own type parameters. The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

public class Util {

**public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)** {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

The complete syntax for invoking this method would be:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.**<Integer, String>**compare(p1, p2);

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

boolean same = Util.compare(p1, p2);

**What are Bounded Type Parameters?**

Used when you want to restrict the types that can be used as type arguments in a parameterized type.

For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is Number. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Ex.

public <U **extends Number**> void inspect(U u){

System.out.println("T: " + t.getClass().getName());

System.out.println("U: " + u.getClass().getName());

}

.

.

.

inspect("some text"); // **error: this is still String!**

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String.

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

public class NaturalNumber<T extends Integer> {

private T n;

public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {

return **n.intValue()** % 2 == 0;

}

// ...

}

The isEven method invokes the intValue method defined in the Integer class through n.

A type parameter can also have multiple bounds:

<T extends B1 & B2 & B3>

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }

**Statement Vs PreparedStatement:**

***JDBC API*** provides 3 different interfaces to execute the different types of SQL queries. They are,

1) [***Statement***](http://javaconceptoftheday.com/java-jdbc-tutorial-sql-create-insert-select-update-delete-examples/)  –  Used to execute normal SQL queries.

2) ***[PreparedStatement](http://javaconceptoftheday.com/preparedstatement-in-java/" \t "_blank)***  –  Used to execute dynamic or parameterized SQL queries.

3) ***[CallableStatement](http://javaconceptoftheday.com/java-callablestatement-stored-procedures-in-out-parameters-examples/" \t "_blank)***  –  Used to execute the stored procedures.

It is recommended to use PreparedStatement if you are executing a particular SQL query multiple times. It gives better performance than Statement interface. Because, PreparedStatement are **precompiled** and the query plan is created only once irrespective of how many times you are executing that query. This will save lots of time.

The performance of CallableStatement interface is higher than the other two interfaces. Because, it calls the stored procedures which are already compiled and stored in the database server.

|  |  |  |
| --- | --- | --- |
|  | ***java.sql.Statement*** | ***java.sql.PreparedStatement*** |
| 1 | Statement is used for **executing a static SQL statement in java JDBC**. | PreparedStatement is used for **executing a precompiled SQL statement in java JDBC.** |
| 2 | java.sql.Statement **cannot accept parameters at runtime in java JDBC**. | java.sql.PreparedStatement can be **executed repeatedly**, it **can accept different parameters at runtime in java JDBC**. |
| 3 | java.sql.Statement is **slower** as compared to PreparedStatement in java JDBC. | java.sql.PreparedStatement is **faster** because it is used for executing precompiled SQL statement in java JDBC. |
| 4 | No such protocol in Statement in java. | Prepared statements are executed through a **non sql binary protocol.**  In binary protocol, communications to the server is **faster** because **less data packets are transferred**. |
| 5 | java.sql.Statement is **suitable for executing DDL commands** - [**CREATE**](http://www.javamadesoeasy.com/2015/07/jdbc-statement-example-execute-create.html), drop, alter and truncate in java JDBC.. | java.sql.PreparedStatement is **suitable for executing DML commands** - [**SELECT**](http://www.javamadesoeasy.com/2015/07/jdbc-preparedstatement-example-execute_31.html)***,*** [**INSERT**](http://www.javamadesoeasy.com/2015/07/jdbc-preparedstatement-example-execute_66.html)***,*** [**UPDATE**](http://www.javamadesoeasy.com/2015/07/jdbc-preparedstatement-example-execute_50.html) **and** [**DELETE**](http://www.javamadesoeasy.com/2015/07/jdbc-preparedstatement-example-execute_75.html) in java JDBC.  It is used for any kind of SQL queries which are to be executed multiple times. |
| 6 | Statement **can’t be used for storing/retrieving image and file in database** (i.e. using BLOB, CLOB datatypes) in java JDBC. | PreparedStatement **can be used for**  [**storing**](http://storing/)**/**[***retrieving***](http://www.javamadesoeasy.com/2015/07/jdbc-retrieve-image-from-database-by.html) **image and**  [***Storing***](http://www.javamadesoeasy.com/2015/07/jdbc-insertstoresave-file-in-database.html) **/**[***retrieving***](http://www.javamadesoeasy.com/2015/07/jdbc-retrieve-file-from-database-by.html) **file in database**  (i.e. by using [BLOB, CLOB](http://www.javamadesoeasy.com/2015/07/difference-between-clob-and-clob-data.html) datatypes) in java JDBC. |
| 7 | java.sql.Statement **does not have** **setArray method in java JDBC.** | java.sql.PreparedStatement **can be used for setting** **java.sql.Array using setArray method.**  While sending it to database the driver converts this java.sql.Array to an SQL ARRAY. |
| 8 | java.sql.Statement **enforces SQL injection**, because we end up using query formed using **concatenated SQL strings in java JDBC**.  Example in java JDBC >  String s1= "select \* from EMPLOYEE where id = ";  **int** i1 = 2 ;  stmt.executeQuery(s1 + String.*valueOf*(i1)); | java.sql.PreparedStatement **prevents SQL injection**, because text for all the parameter values is escaped in java JDBC.  Example in java JDBC >  prepStmt = con.prepareStatement("select \* from EMPLOYEE where ID=? ");  prepStmt.setInt(1, 8);  Here comes one very **important** question, are **PreparedStatement vulnerable to SQL injections in java?**  **YES**, when we use **concatenated SQL strings** rather than using input as a parameter for preparedStatement |
| 9 | java.sql.Statement **does not provide addBatch() method, it provides only addBatch( String sql ) method.**  Hence, same SQL query **can’t** be **executed repeatedly in** Statement in java JDBC**.** | java.sql.PreparedStatement interface **extends** Statement interface and inherits all methods from Statement and additionally adds **addBatch()** method.  **addBatch()  method -** adds a set of parameters to the PreparedStatement object's batch of commands in java JDBC.  Hence, same SQL query **can** be **executed repeatedly in** PreparedStatement in java JDBC**.** |
| 10 | Statement  makes **code less readable and understandable - We may need to write concatenated SQL strings** | PreparedStatement **makes code more** readable and understandable - We **need not to write concatenated SQL strings**, we can use queries and pass **different parameters at runtime using** setter methods. |
| 11 | java.sql.Statement **does not provide such methods in java JDBC.** | java.sql.PreparedStatement provides methods like **getMetadata()** and **getParameterMetadata()**  **getMetadata() -** Method retrieves  ResultSetMetaData object that contains information about the **columns of the ResultSet object** that will be returned when PreparedStatement object is executed.  **getParameterMetadata() -** method retrieves the number, types and properties of PreparedStatement object's parameters in java JDBC. |

**Java Heap Space vs Stack Memory**

**Java Heap Space**

Java Heap space is **used by java runtime** to allocate memory to **Objects and JRE classes**. Whenever we create any object, it’s always created in the Heap space.

Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has **global access** and can be referenced from anywhere of the application.

**Java Stack Memory**

Java Stack memory is **used for** **execution of a thread**. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

Stack memory is always **referenced in LIFO (Last-In-First-Out) order**. Whenever a method is invoked, **a new block** is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.

As soon as method ends, the block becomes **unused** and become **available for next method**. Stack memory size is very less compared to Heap memory.

**Heap and Stack Memory in Java Program**

public class Memory {

public static void main(String[] args) { // Line 1

int i=1; // Line 2

Object obj = new Object(); // Line 3

Memory mem = new Memory(); // Line 4

mem.foo(obj); // Line 5

} // Line 9

private void foo(Object param) { // Line 6

String str = param.toString(); //// Line 7

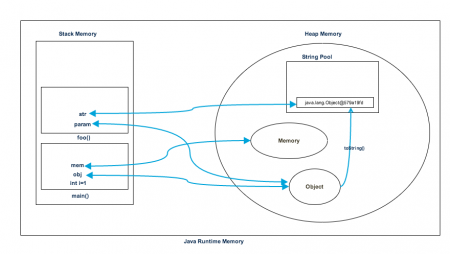
System.out.println(str);

} // Line 8

}

Let’s go through the steps of execution of the program.

* As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in line 3, it’s created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
* Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
* A string is created in line 7, it goes in the String Pool in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.



**Difference between Java Heap Space and Stack Memory**

1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
2. Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
3. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
4. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally.
5. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
6. We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.
7. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
8. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.
9. Where is it stored? Variables that are allocated on the stack are accessible directly from memory, and as such, these can run very fast. Accessing objects on the heap, on the other hand, takes more time.
10. When does the allocation happen? On the stack, memory allocation happens when the program is compiled. Meanwhile, on the heap, it begins when the program is run.

**JDBC Drivers**

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. **Type 1, JDBC-ODBC Bridge plus ODBC Driver**:

JDBC-ODBC bridge driver **uses ODBC driver** to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

+ Easy to use. Can be easily connected to any database.

- Performance degraded because JDBC method call is converted into the ODBC function calls.

- The ODBC driver needs to be installed on the **client** machine.



1. **Type 2: A native API partly Java technology-enabled driver:**

The Native API driver **uses the client-side libraries** of the database. The driver converts JDBC method calls into native calls of the database API.

+ Better in performance than type 1.

- The Native driver & Vendor client library need to be installed on the each **client** machine.



1. **Network Protocol driver, Type 3: Pure Java Driver for Database Middleware:** The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is **fully written in java**.

+ No client side library is required.

- Network support is required on client machine.

- Requires database-specific coding to be done in the middle tier.



1. **Thin driver, Type 4: Direct-to-Database Pure Java Driver:**

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as **thin driver**. It is fully written in Java language.

+ Better performance than all other drivers.

+ No software is required at client side or server side.

- Driver depends on Database.



**In my project**, we have used **datasource in weblogic** for getting DB connection. It internally uses **Type 4 (Thin) driver**.

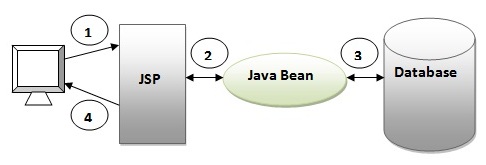
For test automation & FTP/Email Framework, program uses **type 4 (JDBC Thin) driver** as well i.e. oracle.jdbc.OracleDriver or updated oracle.jdbc.driver.OracleDriver.

**Model 1 vs Model 2(MVC) Programming Models**

**Model 1 Architecture:**

**Servlet technology** doesn't create process, rather it creates thread to handle request. The **advantage** of creating thread over process is that it doesn't allocate separate memory area. **Problem** in Servlet technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

**JSP** overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



1. Browser sends request for the JSP page
2. JSP accesses Java Bean and invokes business logic
3. Java Bean connects to the database and get/save data
4. Response is sent to the browser which is generated by JSP

+ Easy and quick to develop web application

- Navigation control is decentralized: If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.

- Time consuming: You need to spend more time to develop custom tags in JSP (this is to avoid scriptlet).

- Hard to extend: It is better for small applications but not for large applications.

**Model 2 (MVC) Architecture:**

Model 2 is based on the MVC (Model View Controller) design pattern. Model The model represents the state (data) and business logic of the application. View The view module is responsible to display data i.e. it represents the presentation. Controller The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.

The Struts framework provides the configurable MVC support.



+ Navigation control is centralized: Now only controller contains the logic to determine the next page.

+ Easy to maintain, extend, test

+ Better separation of concerns

- We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

**Polymorphism example**

**Polymorphism** in java is a concept by which we can perform a single action by different ways.

There are **two types** of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

**Upcasting** - When reference variable of Parent class refers to the object of Child class, it is known as upcasting.

class A{}

class B extends A{}

A a=new B();//upcasting

Reverse process is **downcasting**.

Dog d=(Dog)new Animal(); //Compiles successfully but ClassCastException is thrown at runtime

static void method(Animal a) {

if(a instanceof Dog3){

Dog3 d=(Dog3)a;//downcasting

System.out.println("ok downcasting performed");

}

}

static void method(Animal a) {

Dog4 d=(Dog4)a;//downcasting

System.out.println("ok downcasting performed");

}

**Ex of runtime polymorphism:**

* Bank (Parent); SBI, ICICI, AXIS (Children) - float getRateOfInterest() is the overridden method.
* Shape(Parent); Circle, Triangle, Rectangle (Children); void draw() is the overridden method.
* Animal (Parent); Cat, Lion, Dog (Children) void eat() is the overridden method.

**Why we cannot override static method?**

because static method is bound with class whereas instance method is bound with object. Main method also can't be overriden as it is also a static method.

**Method Overloading:** class has multiple methods having same name but different in parameters.

There are **two ways to overload** the method in java: By changing number of arguments, By changing the data type of the arguments.

In java, Method Overloading is not possible by changing the return type of the method only - java compiler renders compiler time error if you declare the same method having same parameters.

You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only.

**Polymorphism**

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism – compile time polymorphism and runtime polymorphism. **Compile time polymorphism** is achieved by method overloading.

**Method overloading** - methods' name is same but arguments are different. Here compiler will be able to identify the method to invoke at compile time, hence it’s called compile time polymorphism.

**Runtime polymorphism** is implemented when we have “IS-A” relationship between objects. This is also called as **method overriding** because subclass has to override the superclass method for runtime polymorphism. Compiler is not able to decide which class method will be invoked. This decision is done at runtime, hence the name as runtime polymorphism or **dynamic method dispatch**.

**Inheritance**

Inheritance is the object oriented programming concept where an object is based on another object. Inheritance is the mechanism of code reuse. The object that is getting inherited is called **superclass** and the object that inherits the superclass is called **subclass**.

We use **extends** keyword in java to implement inheritance.

**Association**

Association is the OOPS concept to define the RELATIONSHIP between objects. Association defines the multiplicity between objects. For example Teacher and Student objects. There is **one to many** relationship between a teacher and students. Similarly a student can have one to many relationship with teacher objects. However both student and teacher objects are independent of each other.

**Aggregation**

Aggregation is a special type of association. In aggregation, objects have their own life cycle but there is an ownership. Whenever we have “HAS-A” relationship between objects and ownership then it’s a case of aggregation. Ex: Employee has an Address.

**Composition**

Composition is a special case of aggregation. Composition is a more restrictive form of aggregation. When the **contained object in “HAS-A” relationship can’t exist on its own**, then it’s a case of composition. For example, House has-a Room. Here room can’t exist without house.

**Java 7 Features**

<https://way2java.com/java-versions-2/jdk-1-7-features/>

<http://javarevisited.blogspot.in/2014/04/10-jdk-7-features-to-revisit-before-you.html>

Java 6 was nothing on feature; it was all about JVM changes and performance.

|  |  |  |
| --- | --- | --- |
| String in Switch | * Earlier to **JDK 1.**7, **switch** expression takes**int** values or convertible to **int**. * From JDK 1.7, switch accepts string objects also as expression. | * equals() and hashcode() method from java.lang.String is used in comparison, which is case-sensitive. * Java compiler can generate more efficient code than using nested if-then-else statement. |
| Automatic Type Inference in Generic object instantiation  (Diamond operator, <> , in collection classes) | * Empty angle brackets (known as **diamond operator**), **<>**, can be used in specifying generic type instead of writing the exact one. | * Specify types on now only needed on left hand side:   Ex.  Map<String, List<String>> employeeRecords = new HashMap<>();  List<Integer> primes = new ArrayList<>(); |
| Automatic Resource Management | * Before JDK 7, we need to use a finally block, to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. * JDK 7 introduces a try-with-resources statement, which ensures that each of the resources in try(resources) is closed at the end of the statement by calling close() method of AutoClosable. * Any resource (class) that implements interface "java.lang.AutoCloseable" is eligible as a resource statement to write in try block. | * Now in Java 7, you can use try-with-resource feature to automatically close resources, which implements AutoClosable and Closeable interface e.g. Streams, Files, Socket handles, database connections etc. * Since Java is taking care of closing opened resources including files and streams, may be no more leaking of file descriptors and probably an end to file descriptor error. Even JDBC 4.1 is retrofitted as AutoClosable too. * The close() method of AutoCloseable is called implicitly to close the handles. close() method java.lang.Closeable interface is very different from this. |
| Handling multiple exceptions in a single catch block | * A single catch block can handle more than one exception types **separated by pipe ( | ) symbol**. * Super class exception must be caught separately (it is a constraint) i.e . Alternatives in a multi-catch statement cannot be related by sub classing. | * **try** { ...... } **catch**(**ClassNotFoundException**|**SQLException** ex) { ex.printStackTrace(); } * } catch (FileNotFoundException | IOException ex) {   OR  } catch(ArithmeticException | RuntimeException e) {  will throw compile time error. |
| Fork Join Framework | * The fork/join framework is an implementation of the **ExecutorService interface** that allows you to take advantage of multiple processors available in modern servers. | * It is designed for work that can be broken into smaller pieces recursively. * The goal is to use all the available processing power to enhance the performance of your application. |
| Underscore in Numeric literals | * Can insert underscore(s) '\_' in between the digits in an **numeric literals (integral and floating-point literals)** to improve readability. | * long creditCardNumber = 1234\_4567\_8901\_2345L; //16 digit   long ssn = 777\_99\_8888L;  double pi = 3.1415\_9265;  float pif = 3.14\_15\_92\_65f; |
| Integral Types as Binary Literals | * The **integer whole numbers** like byte, short, int and long can be expressed in binary format also with a prefix of 0b or 0B. | * Earlier, we have **0 prefix for octal** and **0x prefix for hexa** and no prefix for binary. |
| Static Blocks | * Earlier to JDK 1.7, to print **static blocks**, no main() method is required. | * From JDK 1.7, if no main() exists, static blocks will not be executed. |
| G1 Garbage Collector | * Garbage first ( G1) performs clean-up where there is most garbage. * It's said that G1 is quite predictable and provides greater **throughput for memory intensive applications**. | * To achieve this it split Java heap memory into multiple regions as opposed to 3 regions in the prior to Java 7 version (new, old and permgen space). |
| More Precise Rethrowing of Exception | * The Java SE 7 compiler performs more precise analysis of re-thrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the **throws clause of a method declaration.** | * Before JDK 7, re-throwing an exception was treated as throwing the type of the catch parameter. * public void obscure() throws IOException{    try {  new FileInputStream("abc.txt").read();  new SimpleDateFormat("ddMMyyyy").parse("12-03-2014"); } catch (Exception ex) {  throw ex;  } }  Earlier this code wouldn’t have been compiled. The throw in catch would have been required to be modified to after removing throws clause from method declaration:  throw new RuntimeException(exception); |

**Abstract class vs Interface - & when to use what?**

**Abstract Class**

An abstract class is a class that is declared abstract - it **may or may not** include abstract methods. Abstract classes **cannot be instantiated**, but they can be subclassed. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Abstract classes are typically **used as base classes** for extension by subclasses. **Consider using abstract classes if** any of these statements apply to your situation:

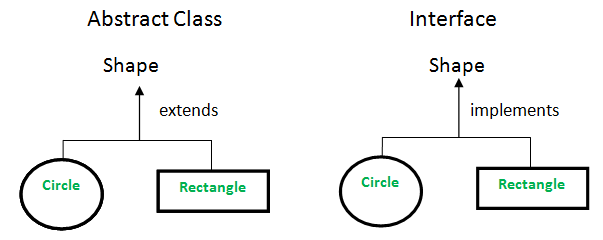
1. You want to share code among several closely **related** classes. You can put these lines of code within abstract class and this abstract class should be extended by all these related classes.
2. You expect that classes that extend your abstract class have many common methods or fields or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

**Interface**

An interface is just the declaration of methods of an Object, it’s not the implementation. In interface, we define what kind of operation an object can perform. These operations are defined by the classes that implement interface. Interfaces form **a contract between the class and the outside world**, and this contract is enforced at build time by the compiler.

A Java class can have **only one superclass**, but it can implement **multiple interfaces**. Thus, if a class already has a different superclass, it can implement an interface, but it cannot extend another abstract class. **Consider using interfaces** **if** any of these statements apply to your situation:

1. It is **total abstraction**. All methods declared within an interface must be implemented by the class(es) that implements this interface.
2. You expect that **unrelated classes** would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
3. You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
4. You want to **take advantage of multiple inheritances**.



**Abstraction vs Encapsulation**

1. **Abstraction** focuses on the outside view of an object (i.e. the interface) **Encapsulation** (information hiding) prevents clients from seeing its inside view, where the behavior of the abstraction is implemented.
2. Abstraction solves the problem in the design side while Encapsulation is the Implementation.
3. Abstraction lets you focus on what the object does instead of how it does it. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
4. The part that is hidden relates to encapsulation while the part that is exposed relates to abstraction.
5. In abstraction we hide something to reduce the complexity of it (implementation hiding) and in encapsulation we hide something to protect the data (info hiding).

**Abstraction** is the concept of hiding the internal details and describing things in simple terms. For example, a method that adds two integers. The method internal processing is hidden from outer world. There are many **ways to achieve abstraction** in object oriented programming, such as encapsulation and inheritance.

**A java program** is also a great example of abstraction. Here java takes care of converting simple statements to machine language and hides the inner implementation details from outer world. You can **use abstraction** using Interface and Abstract class in Java.

=> Abstraction is getting interested from reality.

=> Implementation hiding.

**Encapsulation** is the technique used to implement abstraction in object oriented programming. Encapsulation is used for access restriction to a class members and methods.

=> Wrapping of data and methods in single entity referred as class.

=> Information Hiding.

**Access modifier** keywords are used for encapsulation in object oriented programming. For example, encapsulation in java is achieved **using** private, protected and public keywords.

**JVM Architecture**

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (Write Once Run Anywhere)**,which runs on a **VM**. The **Compiler** compiles the Java file into a Java **.class** file, then that **.class** file is input into the JVM, which Loads and executes the class file.

**JVM Architecture Diagram**

[](http://www.javainterviewpoint.com/wp-content/uploads/2016/01/JVM-Architecture.png)

**How Does the JVM Work?**

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. **Class Loader Subsystem**
2. **Runtime Data Area**
3. **Execution Engine**

**1. Class Loader Subsystem**

Java's [**dynamic class loading**](http://www.javainterviewpoint.com/use-of-class-forname-in-java/) functionality is handled by the class loader subsystem. It loads, links and initializes the class file when it refers to a class for the first time at **runtime**, not **compile time.**

After loading .class file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in java.lang package.

**1.1 Loading**

**Classes will** **be loaded** by this component. Three class loaders which will help in achieving it are:

1. **Boot Strap**[**Class Loader**](http://www.javainterviewpoint.com/)**–** Responsible for loading classes from the bootstrap path, nothing but **rt.jar.**Highest priority will be given to this loader. It loads core java API classes present in JAVA\_HOME/jre/lib directory.
2. **Extension Class Loader** – Responsible for loading classes which are inside **ext** folder **(jre\lib).**
3. **Application Class Loader** –Responsible for loading **Application Level classpath**, path mentioned Environment Variable etc.

The above **Class Loaders** will follow **Delegation Hierarchy Algorithm**while loading the class files.

JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.

**1.2 Linking**

1. **Verify** – It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception **java.lang.VerifyError.**
2. **Prepare** – JVM allocates memory for class variables and initializing the memory to **default values**.
3. **Resolve** – All **symbolic memory references** are replaced with the **original references** from **Method Area**.

**1.3 Initialization**

This is the final phase of Class Loading, here all [**static variables**](http://www.javainterviewpoint.com/use-of-static-keyword-in-java/)will be assigned with the **original values, and the**[static block](http://www.javainterviewpoint.com/java-static-import/)**will be executed.**

**2. Runtime Data Area**

The Runtime Data Area is divided into 5 major components:

1. **Method Area** – All the **class level data** will be stored here, including **static variables**. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the **Objects** and their corresponding**instance variables** and **arrays** will be stored here. There is also one Heap Area per JVM. Since the **Method** and **Heap areas** share memory for multiple threads, the data stored is not thread**safe.**
3. **Stack Area** – For every thread, a separate **runtime stack** will be created. For every **method call**, one entry will be made in the stack memory which is called as **Stack Frame**. All **local variables** will be created in the stack memory. The stack area is thread safe since it is not a shared resource. The Stack Frame is divided into three sub entities:
   1. **Local Variable Array** – Related to the method how many **local variables** are involved and the corresponding values will be stored here.
   2. **Operand stack** – If any intermediate operation is required to perform, **operand stack** acts as runtime workspace to perform the operation.
   3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate**PC Registers,** to hold the address of **current executing instruction** once the instruction is executed the PC register will be **updated** with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method **stack will be created.**

**3. Execution Engine**

The bytecode which is assigned to the **Runtime Data Area** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
   1. **Intermediate Code generator** – Produces intermediate code
   2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
   4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling **"System.gc()"**, but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI)**: **JNI** will be interacting with the **Native Method Libraries** and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries**: It is a collection of the Native Libraries which is required for the Execution Engine.

**Comparator vs Comparable interfaces:**

Java provides **Comparable interface** which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. Comparable interface has **compareTo(T obj)** method which is used by sorting methods. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as argument.

But, in most **real life scenarios**, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. This is the situation where **we need to use Comparator interface** because Comparable.compareTo(Object o) method implementation can sort based on one field only and we can’t chose the field on which we want to sort the Object.

**Comparator interface** **compare(Object o1, Object o2)** **method** need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if first argument is less than the second one and returns zero if they are equal and positive int if first argument is greater than second one.

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

**Covariant return type?**

=> returning own Class reference or its child class reference.

* The covariant return type in java, allows narrowing down return type of the OVERRIDDEN method.
* The covariant return type always works only for non-primitive return types.

It was not possible to override any method by changing the return type before Java 1.5 version. The only limitation is child’s return type should be sub-type of parent’s return type (non-primitive type only).

**Advantages:**

* It helps to avoid confusing type casts present in the class hierarchy and thus making the code readable, usable and maintainable.
* We get a liberty to have more specific return types when overriding methods.
* Help in preventing run-time ClassCastExceptions on returns.

Ex:

class A{

A get(){return this;}

}

class B1 extends A{

B1 get(){return this;}

void message(){System.out.println("welcome to covariant return type");}

public static void main(String args[]){

new B1().get().message();

}

}

Both methods (message in parent & child) have different return types but it is **method overriding**. It is possible because of covariant return type.

**Static var/method in inheritance?**

* Child can access directly the Static var/method from Parent class if it's not defined with private access specifier.
* You can use Parent Class Name or Child class name object to use it, however a WARNING will be shown that it "should be accessed in a static way".

**Can we define Generic Class/Interface in java?**

YES. We can define our **own classes with generics type**. A **generic type** is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter.

public class GenericsType<T> {

private T t;

public T get(){

return this.t;

}

public void set(T t1){

this.t=t1;

}

public static void main(String args[]){

GenericsType<String> type = new GenericsType<>();

type.set("Pankaj"); //valid

GenericsType type1 = new GenericsType(); //raw type

type1.set("Pankaj"); //valid

type1.set(10); //valid and autoboxing support

}

}

If we **don’t provide the type** at the time of creation, compiler will produce a warning that “GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized”.

We can use @SuppressWarnings("rawtypes") annotation to suppress the compiler warning,

When we don’t provide type, the type becomes **Object** and hence it’s allowing both String and Integer objects but we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

**Generic Interface**

**Comparable** interface is a great example of Generics in interfaces and it’s written as:

package java.lang;

import java.util.\*;

public interface Comparable<T> {

public int compareTo(T o);

}

In similar way, we can create generic interfaces in java. We can also have **multiple type parameters** as in **Map** interface.

Again, we can provide parameterized value to a parameterized type also, for example new HashMap<String, **List<String>**>(); is valid.

Generics also comes with its own **naming conventions**. Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

* **E** – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
* **K** – Key (Used in Map)
* **N** – Number
* **T** – Type
* **V** – Value (Used in Map)

**Java Generic Method**

Sometimes we don’t want whole class to be parameterized, in that case we can create java generics method. Since constructor is a special kind of method, we can use generics type in constructors too.

Ex:

//Java Generic Method

public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T> g2){

return g1.get().equals(g2.get());

}

boolean isEqual = GenericsMethods.<String>isEqual(g1, g2);

//above statement can be written simply as

isEqual = GenericsMethods.isEqual(g1, g2);

//This feature, known as **type inference**, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

//Compiler will infer the type that is needed

**Return in Java:**

The return keyword is used to return from a method when its execution is complete. When a return statement is reached in a method, the **program** returns to the code that invoked it.

A method **can return a value or reference type or does not return a value.** If a method does not return a value, the method must be declared **void** and it doesn’t need to contain a return statement.

If a method declare to return a value, then it must use the return statement within the body of method. The data type of the return value must match the method’s declared return type.

**return;** is **valid** and does not throw compile time error. Any statement post that will be unreachable & will throw compile time error.