# Java Notes

# Contents

# Preface

## Who is this book for?

If answer to below questions if yes, then yes:

- You know the basics of programming language?
- You want notes to review before attending interview?
- You want notes to keep handy while programming, developing and designing?
- You want to recollect all important topics for interview in less time like few days?
- You don't want to learn 100 or 1000 interview questions with answers?
- You don't want to spend months for learning concepts and their application?

## This book doesn't suit you if?

- You are new to Java?
- You are preparing for Java certification?
- You are Delivery manager?
- You are looking for Complete Reference or Head First?
- You are looking for complete list & details of Java API
- You are expecting documentation for JDK kit.
- You are looking for designing concepts or Architect's hand book.
- You want to go deep in each and every topic of Java.

## What to expect from book?

A handy notes for Java, which will help in daily programming, development and also when you are giving or taking interview; quick reference guide for daily programming & development. This book is designed to help Java developers from beginner to intermediate to nearly expert level, unless you really expert.

## Using Book:

Book is divided into 20 chapters with two additional chapters in around 50 pages, and covers none of the topics related more to Java EE like JSF, EJB, JMS, and others. One can jump directly to any chapter. No chapter requires deep understanding of previous chapter. It is not required to read all chapters sequentially, and that are notes all about, you can jump anywhere. Indeed, it will be good to read chapters one by one. Some chapters finish in one page only, but some last more than two pages, depending on the popularity of subject in interviews. It is strongly recommended to take a break between two chapters.

## About Author:

I am Abhishek Upadhyay with more than 10 years of IT industry experience with 5 companies including Amdocs, IBM, and Barclays. I used Java/Java EE for developing CRM, Stock exchange, Energy management, Banking Application, and Enterprise Integration.

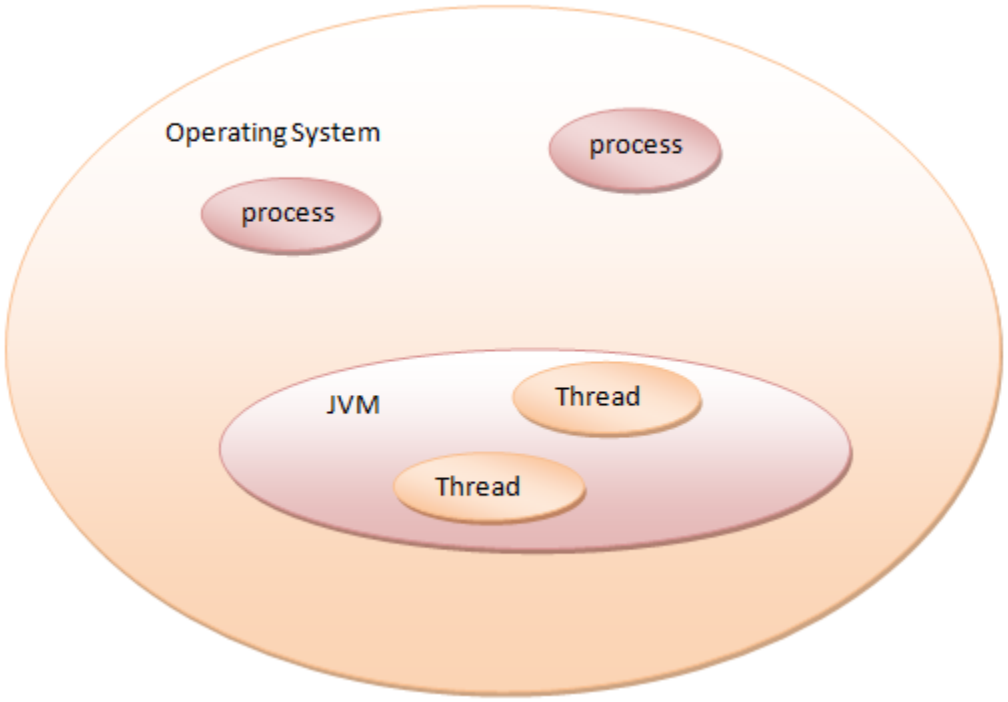
## Feedback:

Comments, Suggestions, feedback, and corrections are heartily welcome and highly valuable. If you are reading this book and any idea of improvement comes in your mind, please do share with me. I am always checking my personal blog for comments. You can leave a comment on twitter – @shekup or drop me a mail at shek.up@gmail.com. If you need more clarification on any topic or want to keep updated with free e-books, keep visiting my blog - shekup.blogspot.com

## Further Reading:

I highly recommend reading books – "A Programmers Guide to Java Certification" by Khalid A. Mughal & Rolf W. Rasmussen, "Java Concurrency in Practice" by Brian Goetz, "Java Threads" by Scott Oaks & Henri Wong, "Effective Java" by Joshua Bloch, javaworld web site, and tutorials.jenkov.com. Short & Concise tutorials can also be found on shekup.blogspot.com on some topics.

## Java Environment




**JRE includes JVM** – Java Virtual Machine – makes Java - machine and OS independent.

> JVM is a software (an implementation) that contains everything to run a java program[i].

Programs inside the VM are limited to resources & generalization provided by VM.

> The Java Virtual Machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time[ii].

Its JVM's job to load class files and execute the byte codes they contain. A Java program (.java) is compiled into class file. A class file contains JVM instructions (bytecodes), symbol table (like we use TreeMap or HashMap), and other ancillary (additional) information. If you only want to run Java programs and don't develop them, you only need JRE. To develop Java applications you need JDK, which includes JRE.

**JCP (Java Community Process)** is the process that allows interested parties to develop standard technical specification for Java. JCP process involves **JSR (Java Specification Request)**: formal document that narrates the proposed specification. JSR is also actual description of final specification of Java platform. JSR defines the Java API's contained in JDK in any edition: Java ME, Java SE, & Java EE.

Packages starting java.*, javax.*, and also belonging to org.omg.*, org.w3c.*, and org.xml.*.

**Java and Process:** A computer system has many processes, even on single core, and on multiple execution cores. Multiple processes can run concurrently on single core in time-sliced manner giving a feel of multiprocessing. A process is self-contained execution environment with private resources and memory, and its same thing seen in task manager in Windows; JVM run as a single process. It's possible to start a new process from within JVM using ***ProcessBuilder.start()***, which returns an instance of Process class to control process, obtain information of process, input to process, output to process, kill process, etc., but this operation to create an Operating System process is highly system-dependent. A process can contain one or more threads. Processes and threads within process communicate via signals - asynchronous notification of an event. By default, processes don't share memory. Shared memory among processes that is a several processes can read & write in a common memory segment subject to some synchronization, is fast way of inter-process communication compared to signal.

A Java application or Java program will get one JVM and its own OS level process. All threads started within Java program will belong to single process on most platforms. So, from task manager we can kill processes not threads, but from within a Java program (thread), one can selectively kill other threads.

A Java program has at least one thread, called main thread, from where other threads are created. Threads use mutex (mutually exclusive), which is a lock, used to lock a resource that a thread is using and doesn't want other threads to modify/use that resource. In Java, mutex or lock is a Java object. Once a thread is done with its work, it can release the resource, by releasing the lock. Probably threads are waiting to acquire the resource. One thread will be selected from all those waiting threads, and selected thread will be given lock.

Each thread maintains its own stack – a stack of elements – each element relates to a method call. Each new method call pushes new record (element) on the top of stack and record holds information like variables and storage. Method on the top is the one currently executing.

## Java Class

A java project is modularized using packages such as customer package, account package, and so on. Java came at the same time when internet was rising, which reflects in the naming pattern of package. Java package names generally start with com, because companies use reversed internet domain, so if Internet domain name of a company ECOM is http://customers.ecom.com/, package name will be *com.ecom.customers*. Inside package are classes and/or interfaces. Naming convention for classes and interfaces is like adjectives (Serializable, Cloneable, and Runnable) for interfaces and nouns (Customer, Employee, User) for classes. Interfaces are contracts and guarantee a behavior. An object is instance of class. Assume, *Product* is an interface and multiple type of products (phone, tablet, book, and so on). Product interface defines the contract (such as *payment*, which is a method that defines how payment will be done for product). Any class that wants to represent product should have relevant name and implement the interface *Product*.

*Phone implements Product*
*Tablet implements Product*
*Book implements Product*

*Phone*, *Tablet*, and *Book* represent different type of product and each class also provide implementation of method *payment*(). Company ECOM might be selling different type of phones: iPhone, Nokia, and so on and each individual phone will be represented in run time (real time) by an object (instance of class Phone)

*Phone iPhone = new Phone ()*
*Phone nokia = new Phone ()*
*Phone lg = new Phone ()*

Contracts can also be defined in an Abstract class instead of an Interface. Only reason to create Abstract class is to share some code between closely related classes, that is, if Product is an abstract class then it will contain some implemented methods.

A class will have constructor, variables, methods, and control statements.

> A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type.[iii]

**Constructors** cannot be inherited or overridden; constructors can be overloaded in the same class. Multiple constructors of same class can form local chain, i.e. one constructor can be called form another using keyword this(). Use of this should be first statement. Similarly constructor of superclass can be called from subclass using super(). Use of super should be first statement. This implies this & super both can't appear in same constructor. Appropriate parameters should be passes to this() or super().
If constructor in subclass does not have an explicit call to super(), or if constructor at the end of *this* chain does not have an explicit call to super(), then compiler will inset super() – with no parameters, to invoke default constructor of superclass.

Instance Variable, Class variables, local variables, and parameters are four kinds of variables.

Different kind of **Control flow statements**: Decision making statements (if-else, switch), Looping statements (for, while, do-while), and Branching statements (break, continue, and return). A good program will always use correct flow statement, and can never complete without control statements & variables.

**Methods** can be overloaded within class, and can be overridden in extending class. Overriding methods is the powerful OOPs feature of Java. Overloaded methods differ in either number of parameters or type of parameters or order of parameters, while keeping the method name same. Varargs (…) enables use of arbitrary number of arguments, and method is called variable arity method –

```
printObjects(Object … data) {
}
```

Client program calls printObject(String str1, String str2) // printObject({str1, str2}) will be passed.

Variables are not overridden in Subclass, instead they are hidden.

Static method cannot override an inherited instance method; instead it can hide the static method in superclass if exact requirements match.

A **protected** method is available to all classes in same package, and to subclass in any package. By default a member is only available in same package only.

At any time only one thread can execute a **synchronized** method of any object.

**Underlying Java Classes:** Among all fundamental Java API classes, **Object** is the most used, since all Java Classes extend Object class. Object class contains – clone(), equals(), finalize(), getClass(), hasCode(), notify(), notifyAll(), toString(), and overloaded wait() operations.

Some very commonly used fundamental classes are wrapper classes (Boolean, Character, Byte, Short, Integer, Long, Float, Double), String, StringBuilder, StringBuffer, Thread, Exception, Collection, Hashtable, InputStream, Reader, OutputStream, Socket, Math, and so on.

Both *StringBuffer* & *StringBuilder* are mutable strings. *StringBuffer* is thread safe.

Instance of class is **Object**. Most common way to initialize the object is using new keyword, but object can also be instantiated using clone(), Class.forName(), Consturctor.newInstance(), etc. In some more cases objects are instantiated like during deserialization, varargs(…), lambdas expression, etc. Fields inside objects can be initialized from inside methods, from constructors, or from static initialize block. Static initializer block is used for initializing static fields; code inside block is executed only once; and block cannot have return.

**Final:** Final class cannot be extended. Final method cannot be overridden. A final method does not improve performance. Final variable can be initialized only once. Final, blocks very important features of software, which is extension. But while designing a product which is open for extension, a need might arise to make some things final. A Final keyword makes developers or designers intension very much clear, which can't be conveyed though comments or documentation.

**Immutable:** An immutable object can be created from a class which defines all variables private, and does not expose any setter method; means there is no way to change the state of object after it is instantiated. Loophole might be object if object references other objects which are mutable, for example, an Arraylist if passed to object in a getter and Arraylist is used to define state of object, but the same Arraylist can be altered from outside and thus change the state of object. Advantage of immutable objects comes in concurrent programming.

A java class also contains **Annotations**: metadata - provide data about program and data is not part of program itself; they do not directly affect the program. For example, **@SuppressWarnings** tells compiler to suppress some warnings that it would have generated. **@Documented, @Inherited, @Retention** and **@Target.** Each method should include the **@exception**, **@param**, and **@return** javadoc tags where appropriate. We don't remove any method from class if it is not to be used, we use **@deprecated** tag.

## Java Class Loading

Classes are generated by compiler and JAR is just container of classes. A Java class loader (some subclass of ClassLoader) loads the java class into JVM. If ClassLoader is asked to load a class - It will check if class is already loaded; if not loaded then ask parent to load the class; if parent can't load, it will attempt to load the class. When a class is loaded, all classes it refers are also loaded. Unreferenced classes are not loaded until the time they are referenced.

Lot happens when a Java class is loaded like decoding the binary format and verifying it, checking compatibility, instantiating instance of java.lang.Class to represent the class. Strategy is to look for a .class file with same name as given name and load the file. This is done by ClassLoader. Every ClassLoader has a parent, unless it is virtual machine's built in class loader "bootstrap class loader". Normally, files are loaded from local file system in a platform dependent manner, but files can also be loaded from network.

When JVM is started, the bootstrap classloader will load bootstrap classes like runtime classes in rt.jar, internationalization classes in i18n.jar, etc., Next Extension classloader to classes from lib/ext (like jdk/packages/lib/ext) directory. Third is System class path class loader, which loads classes from the CLASSPATH. Which also means classpath will only be searched if class hasn't been found among classes in rt.jar, or extensions.

Loading a class dynamically is easy. All you need to do is to obtain a ClassLoader and call its loadClass() method.

## Serialization

> To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is **serializable** if its class or any of its superclasses implements either the java.io.Serializable interface or its subinterface, java.io.Externalizable. Deserialization is the process of converting the serialized form of an object back into a copy of the object.[iv]

If requirement is not to make any variable to be part of object's serialized state, then mark it transient or static. "NonSerializableException" will be thrown if any referred object or the object itself doesn't implements "Serializable" interface. SerialVersionUID maintains version of classes, supposing class might change after serializing an object, then to avoid errors while deserializing.

**Externalizable** extends Serializable, but unlike Serializable, Externalizable has operations readExternal() and writeExternal(), and expects it is the programmers responsibility to implement these methods and explicitly mention which all variables, etc. needs to be saved, whereas Serializable will serialize all the variables and even variables in all super classes. Java compiler sees if Object implements Externalizable, it will rely totally on writeExternal(), otherwise if Serializable is implemented it will use ObjectOuputStream's writeObject() to save complete state of object unless class overrides the writeObject().

## Generics

With Java's Generics features one can set the type of the collection to limit what kind of objects can be inserted into the collection. Compiler will complain if other type inserted. One doesn't need to cast the values obtained from the collection, even if using Iterator. Additionally, new for-loop (also referred to as "for-each") which works well with generified collections. One can have Generic List, Generic Set, Generic Map, and Generic Classes. Type parameters provide a way for us to re-use the same code with different inputs. Type parameter naming convention -

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types.

Simple and Generic version of Box class[v], note the difference.

```
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.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

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

Image source - https://docs.oracle.com/javase/tutorial/java/generics/types.html

**Bounded type parameters**: There may be times 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.

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

class D <T extends A & B & C> ;

**If** public void boxTest(Box<Number> n) is defined and Box<Integer> or Box<Double> is passed,
**then** It will not be accepted
**because** Box<Integer> is not a subtype of Box<Number>; even though Integer is a subtype of Number.

Note - MyClass<A> has no relationship to MyClass<B>, even if A & B are related.

ArrayList<String> is a subtype of List<String>, which is a subtype of Collection<String>.

**? – WildCard**

To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, one would specify List<? extends Number>. The term List<Number> is more restrictive than List<? extends Number> because the former matches a list of type Number only, whereas the latter matches a list of type Number or any of its subclasses.

If a method only uses the methods provided by Object class, like toString, OR method code doesn't depend on type parameter, for example, a method which prints any object passed to it. printList(List<Object>), but this will only work for List of Objects, Instead we should write printList(List<?>).

WildCard can also be used for Lower bound, like - addNumbers(List<? super Integer> list).

## enum

Before JDK 1.5, developers heavily used static final variables, but now more sophisticated **enum** option is available, which means, replace below -

public static final SEASON_WINTER = 0;
public static final SEASON_SPRING= 1;
public static final SEASON_SUMMER= 2;
public static final SEASON_FALL= 3;

with **enum Season {WINTER, SPRING, SUMMER, FALL}**
which can be referred later like Season season = Season.WINTER;
**enum** declaration defines a full-fledged class, it allows to add methods & fields, implement interfaces, and it provides high quality implementation of Object class, like below –

```
enum Season {
            WINTER(1),
            SPRING(2),
            SUMMER(3),
            FALL(4);
            private int seasonNum;
            Season(int seasonNum){
                this.seasonNum = seasonNum;
                }
            public int getSeasonNumber(){
                return this.seasonNum;
                }
            }

// Sample use
Season season = Season.SUMMER;
int seasonNumber = season.getSeasonNumber();
```

## Equals, Hash Code, and Compare


In library, books are arranged in different compartments. Few rules can be assumed for library:

- Similar books are set in one compartment.
- In a compartment, same book is not put twice
- Books in a compartment are sorted possibly in alphabetical order.
- Not only one one book in one compartment.
- Don't just randomly put books in any compartment

And arrange books in a manner that it is easy to search, easy to put, easy to get, and easy to remove.

Equals, hascode, and Comparator help in storing objects in similar way in memory.

```
public boolean equals(Object obj)
```

IF you are overriding equals, then one should follow some guidelines –

x.equals(x) should return true,

If x.equals(y) returns true, then y.equals(x) should also return true.

If x.equals(y) is true, and y.equals(z) is true, then x.equals(z) is also true.

x.equals(null) should return false, and not NullPointerException.

The default implementation of x.equals(y) will return true only if x==y; but, if rule is two objects of any custom Release class are only equal when their versionNumber field has same value, then one must override the **equals** method. **HashCode** is an integer value based on identity of Object, and it is not dependent on the state of object. *HashCode* is used in hash based collections like HashMap. If two objects are equal then their *hashCode* should be same, which is helpful in storing objects in a HashMap, like below –

| | | | | | | |
|---|---|---|---|---|---|---|
| **HASHING** | array[0] | Key1 | Key6 | Key12 | Bucket1 | Key1, Key6, Key12 have same hashCode |
| | array[1] | Key2 | | | Bucket2 | Key2 has different hashcode and doesn't matches with any other key |
| | array[2] | Key3 | Key7 | Key15 | Bucket3 | Key3, Key7, and Key15 have same hashCode. |
| | array[3] | Key4 | Key8 | | Bucket4 | |
| | array[4] | Key5 | Key13 | | Bucket5 | |
| | . | | | | . | |
| | . | | | | . | |
| | array[n-1] | Key9 | Key11 | Keyn | Bucketn | |

Finding an object in HashMap, involves first determine *hashCode* to find the right bucket, and use equals to get exact object. To maintain the order of books in a shelf demands a small logic, which might be insertion order, or alphabetical order, or possibly year of publish order. Best part in Java collections is that one only need to tell the logic and collection will take care of arranging books. Order helps in efficient searching. Order is reflected when iterating over a sorted Collection. A number of Collections effectively place objects at right place, but they expect programmer to provide implementation of equals, *hashCode*, and compare.

Books with same hashcode are placed in same compartment. Within same compartment, books are sorted according to implementation of comparator.

While searching for book, one needs to know the hashcode to find the right compartment; then within compartment use equals to find the right book. This also implies that if two books are equal then their hashcodes must be same but vice-versa is not true.

```java
public class Book implements Comparable{
    private String title;
    // getter & setter
    public boolean equals(Object obj) {
        if(!(obj instanceof Book))
            return false;
        Book book = (Book) obj;
        if(this.getTitle().equals(book.getTitle()))
            return true;
        return false;
    }

    public int hashCode(){
        return 31*title.hashCode();
    }

    public int compareTo(Book b){
        return this.getTitle().compareTo(b.getTitle());
    }
}
```

# Collections

**Array** is one basic data structure, which is indexed collection of fixed number of elements of same data type; every element in array has position which is called index – starting from 0. Array themselves are objects and can hold either primitive data types like int, float, etc. or can hold reference, where references are all of specific reference type. Most importantly the reference can be referring another array, and thus all references are referring to another array, thus we have multi-dimensional array.

Customer [] [] custArray = new Customer [2][2], will be like and array of two arrays.

{

{new Customer (), new Customer ()},

{new Customer (), new Customer ()}

}

Arrays are most basic form of collection, where collection is a collection of objects, so that a group of objects can be treated as single unit or object, with additional features of any time adding, removing, iterating, and updating the objects within group.

A **Collection** must be iterable, so java.util.Collection interface extends the Iterable, and provides basic operations like – size(), isEmpty(), contains(), add(), remove(), containsAll(), addAll(), removeAll(), clear(), along with iterator(), which allows sequential access to the elements of collection. Iterator returned by iterator() method of Collection, provides methods – hasNext(), next(), and remove().

Some common Collection interfaces are –

## List

List is sequence of elements where each element is positioned at an index; Elements can be accessed by their index.
ArrayList, Vector, LinkedList

## Queue

Insertion at one end, Removal at another end. FIFO
PriorityQueue, BlockingQueue

## Deque (“deck”)

Extends queue and elements can added or removed at both ends
ArrayDeque, ConcurrentLinkedDeque, BlockingDeque

## Set

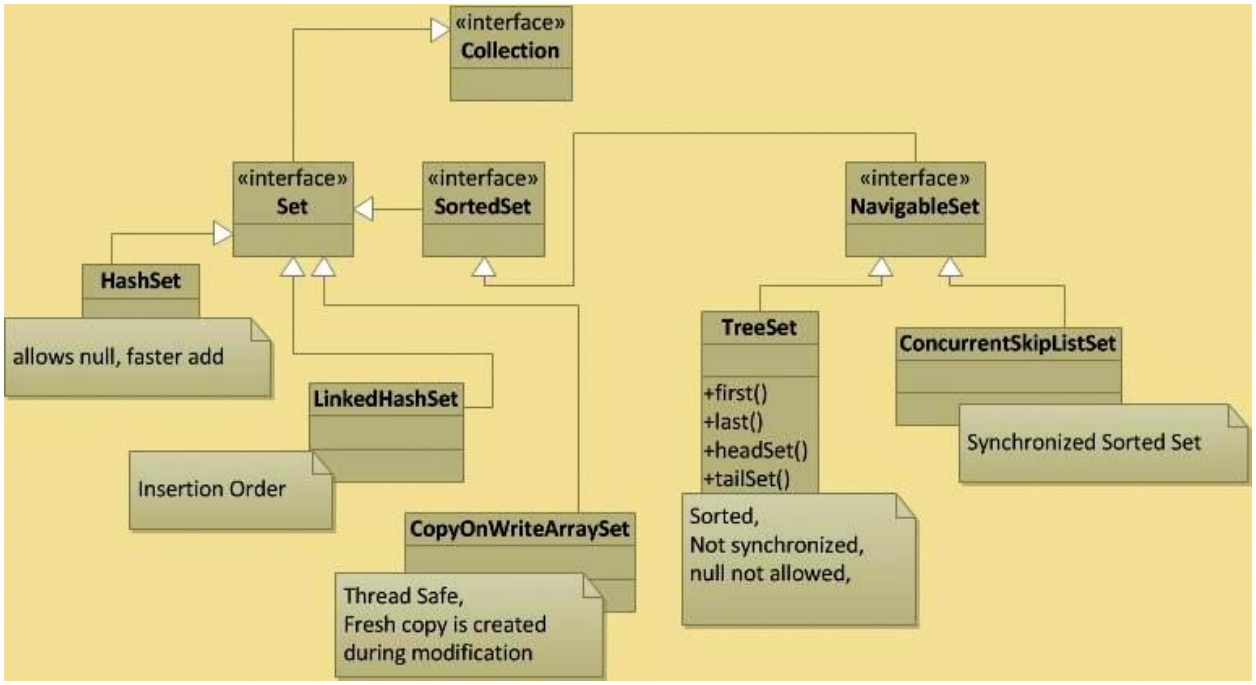
Set of unique elements

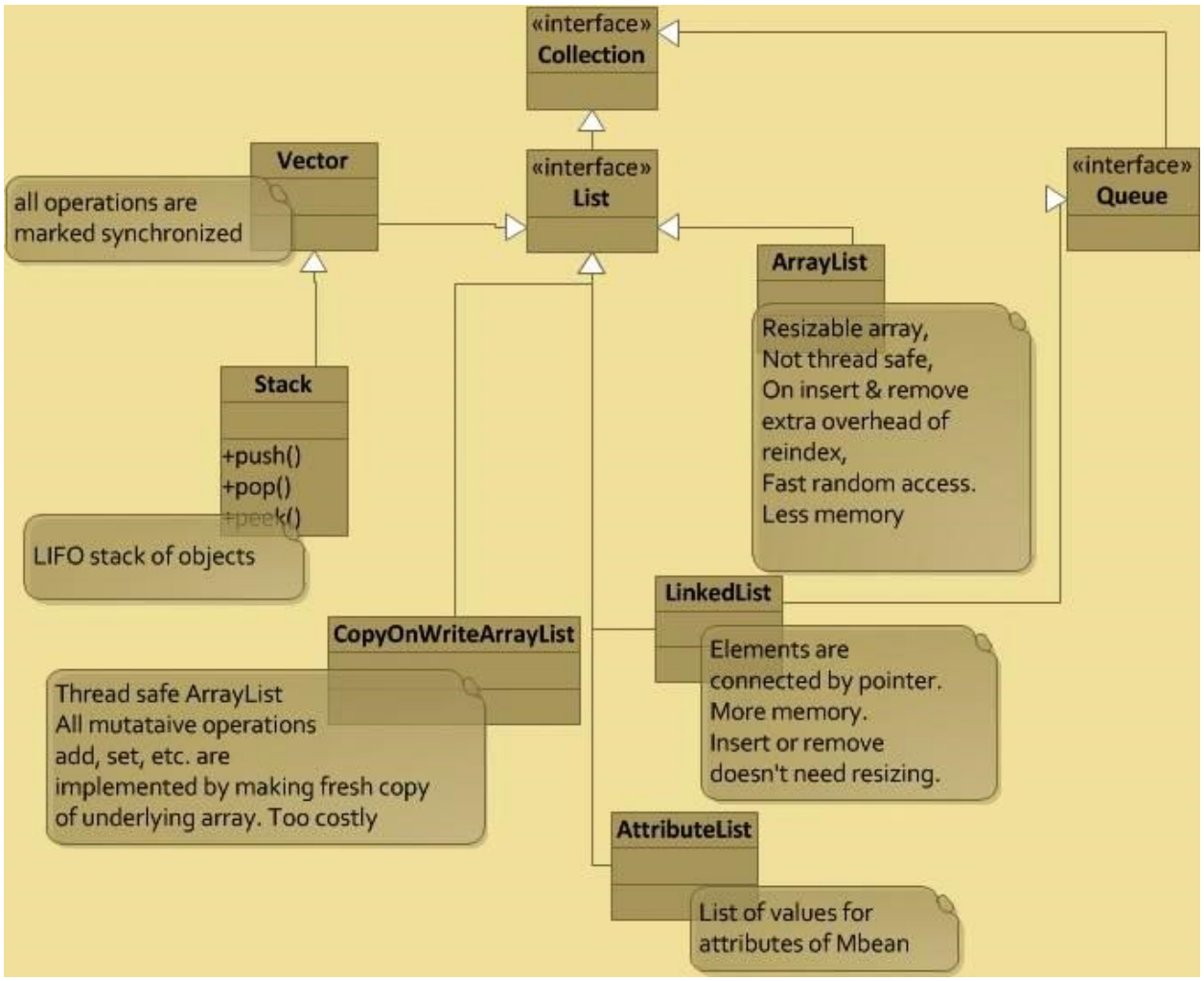
## SortedSet

Set of unique & sorted elements. Elements are ordered using comparator. Elements must implement comparable. headSet(Element e) returns elements <= e, tailSet(Element e) returns element >= e, subset(Element e1, Element e2) returns elements from e1 to e2, first() returns lowest element, and last() returns highest element.

## NavigableSet

Extended SortedSet with addional methods – lower(), floor(), ceiling(), higher(), pollFirst(), pollLast() – which help in searching within set. headset(), tailSet(), subset() accept additional parameters. In reality, we have to always use NavigableSet and never SortedSet.
Below picture shows the available Set implementation classes -

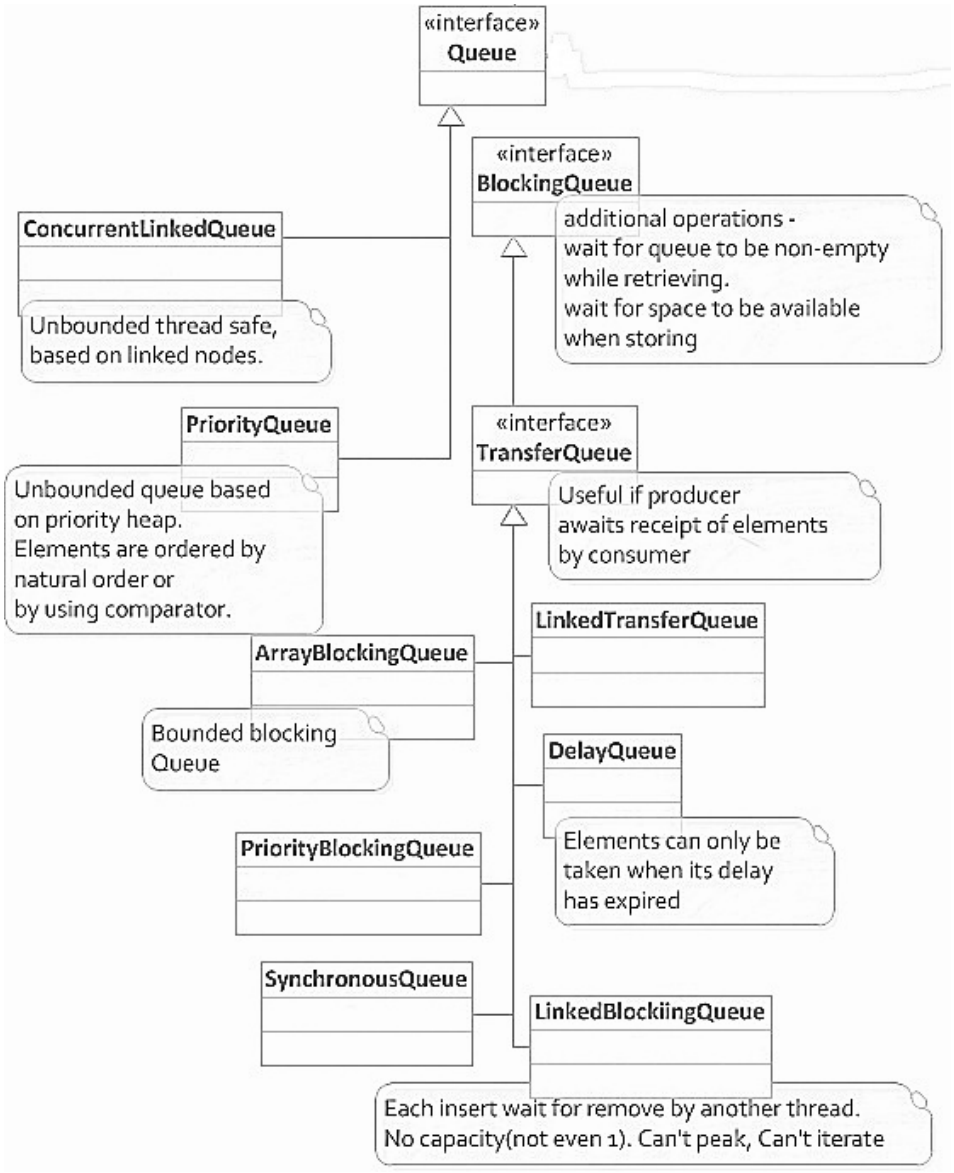




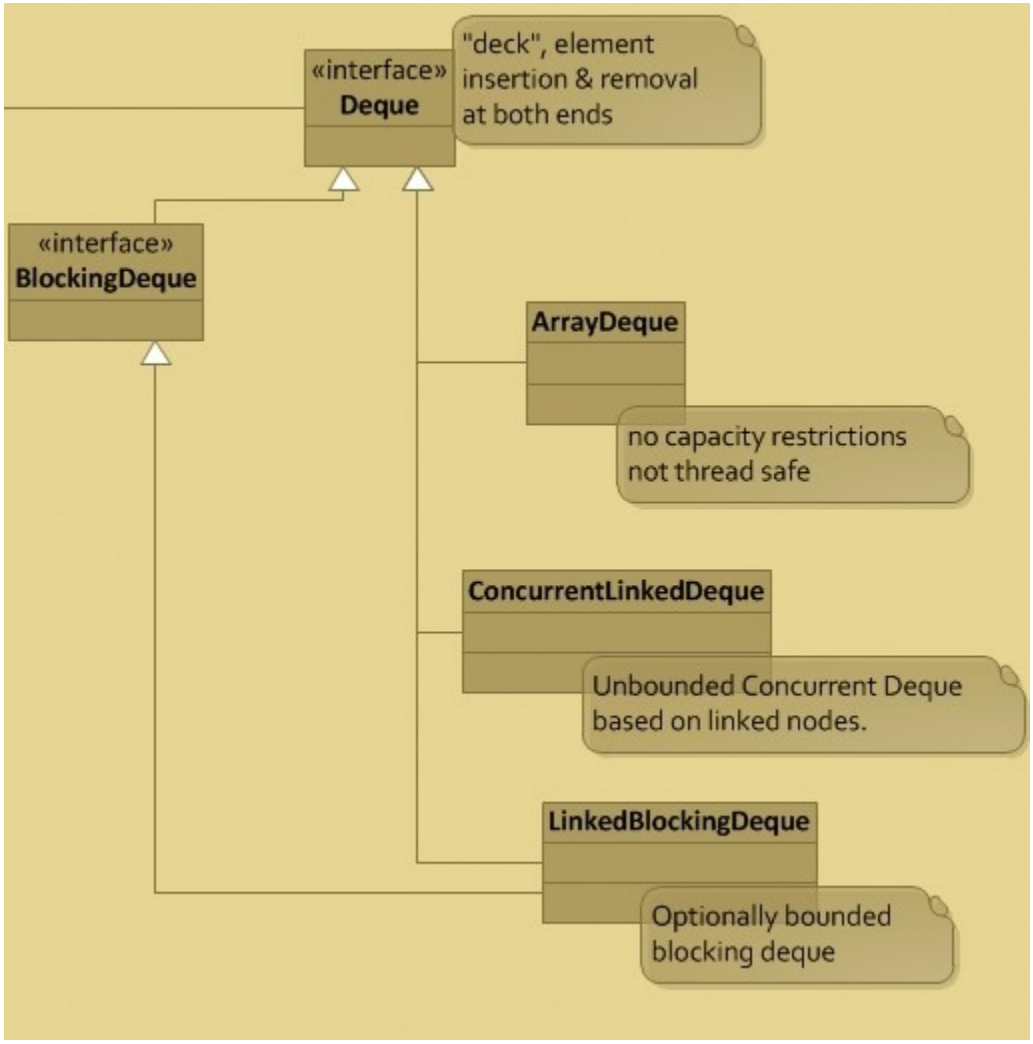

Above are available List implementation classes.

Basic difference between array and Lists is that List can dynamically resize themselves. Both Array & Arraylist give constant time performance for adding & getting element, if you know index. But Arraylist implementing Collection, provides larger set of API's. As such you can't remove element from Array, you can only set it null and make your program error-prone. ArrayList is faster for retrieval compared to LinkedList, deletion is faster in LinkedList.

Below are the Queue implementation classes –



And below are the Deque implementation classes



## Maps

| Key | Value | | |
|---|---|---|---|
| Key1 | Value1 | *Entry1* | Entry<Key1, Value1> |
| Key2 | Value2 | *Entry2* | Entry<Key2, Value2> |
| Key3 | Value1 | *Entry3* | Entry<Key3, Value1> |
| Key4 | Value3 | *Entry4* | Entry<Key4, Value3> |
| Key5 | Value2 | *Entry5* | Entry<Key5, Value2> |

**Map** maintains pairs – Key & Value pairs, where Key's have to unique, but Values do not. One Key, Value pair is called Entry<K,V>. Map doesn't extend's collection, because it is not a collection of only Value objects. Objects can be added to Map by calling put(Key K, Value V), and for retrieving one need to call get(Key K), and it will return the Value attached to the Key. Other helpful methods are remove(Key K), containsKey(Key K), containsValue(Value V), size(), isEmpty(), clear(), and putAll(Map map). We can also retrieve the Collection of Keys, or Collection of Values, or Collection of Entries from a Map. Map interface provides three collection views – Set of keys, collection of values, and Set of key-value mappings.

Set<Key> keySet()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()

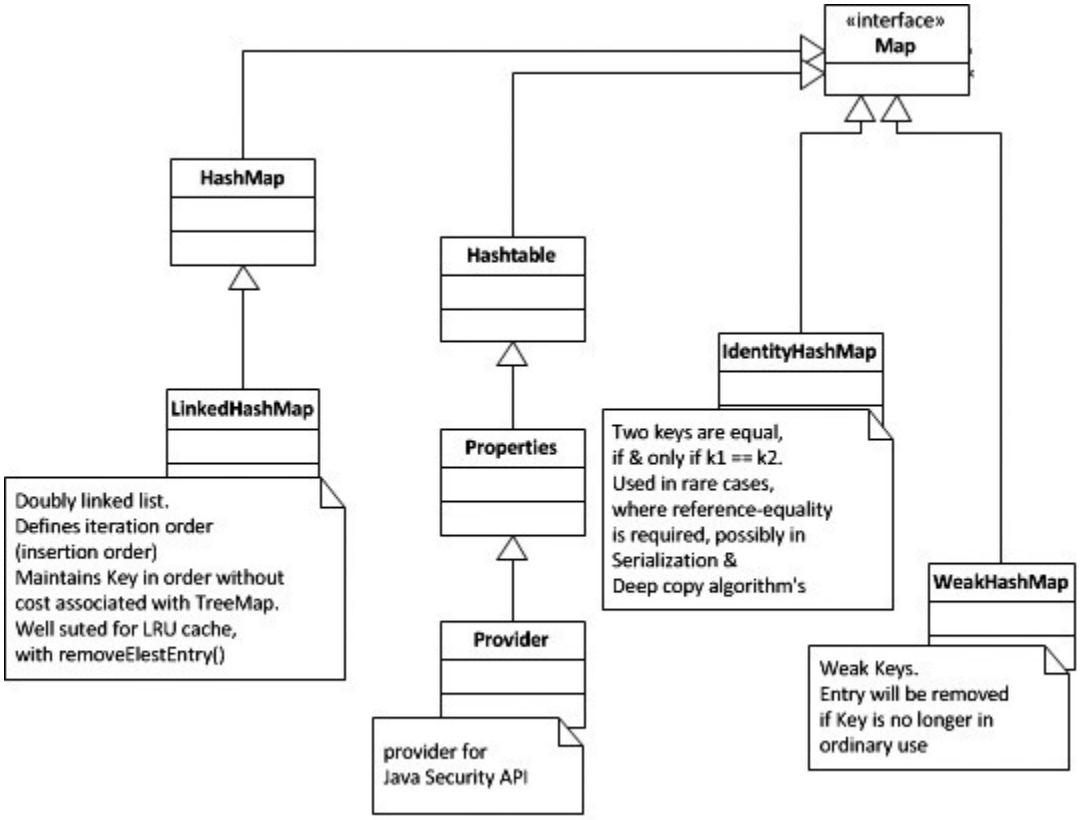
**Algorithm for storing Key, Value pair in HashMap and providing proper retrieval is quite interesting and favorite interview question:** pair is stored in HashMap. Pair is an entry to hashMap. Entries are stored in an array: an array of entries. Each Entry has a key. Calculating **hash(key.hashcode())** determines the index in array. hash method shortens the hashcode value to a valid int index. The value at any index in array is called bucket, which holds the Entry.

If for two Entries index comes out to be same, while storing, which is possible if two keys have same hashCode value. Then those two entries are stored in same bucket. Bucket is a linkedlist at any index in array, and bucket holds the Entries. So, map can have multiple entries in same bucket.

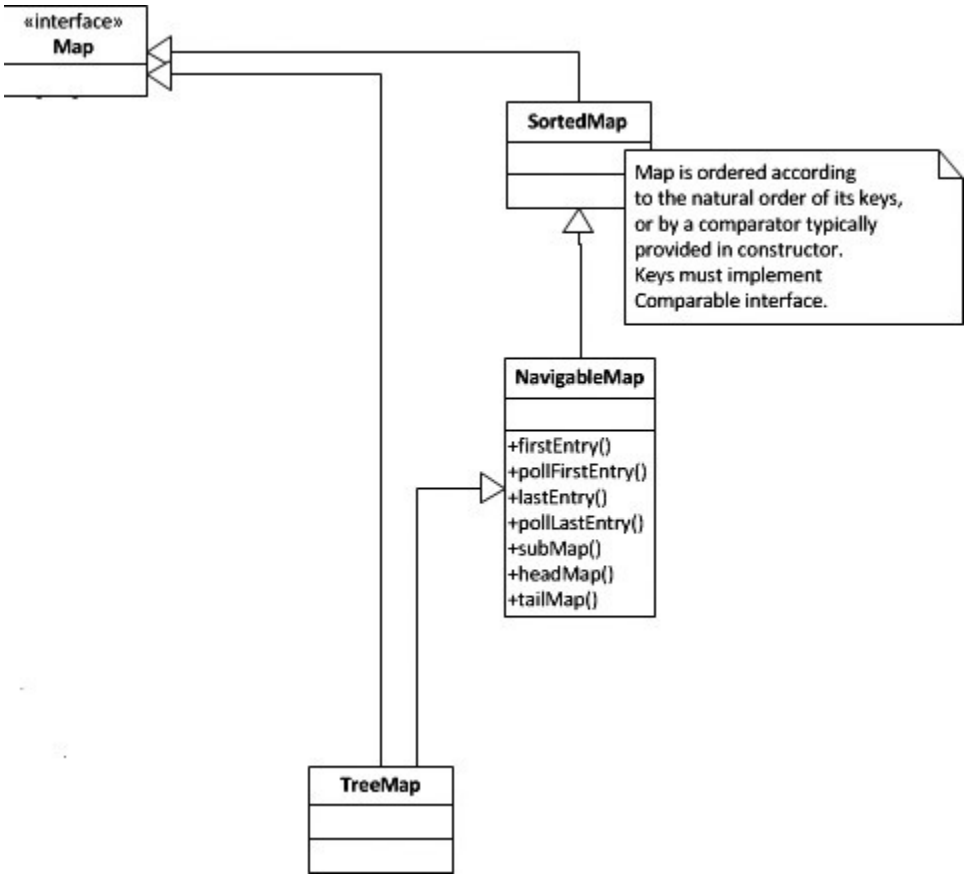
When get(Key) operation is called; following happens-
Calculating the hash(key.hashcode) one knows the index of the bucket in array.
If bucket holds more than one Key, use equals(on key) to return the exact match.

| Map Implentation | Description | Comments |
|---|---|---|
| | | |
| HashMap | Unordered map<br>Not thread-safe.<br>Permits one null key. | |
| HashTable | Unordered Map,<br>Thread Safe<br>Not null keys & Values only | Performance penalty |
| LinkedHashMap | Extends HashMap but Ordered<br>By default order is Key insertion order.<br>Ordering mode can be specified in constructor | HashMap to be preffered if Ordering is not an issue. Add, remove, search is slower compared to HashMap, due to slight expense of maintainig the linked list. Iteration is faster when compared to HashMap |

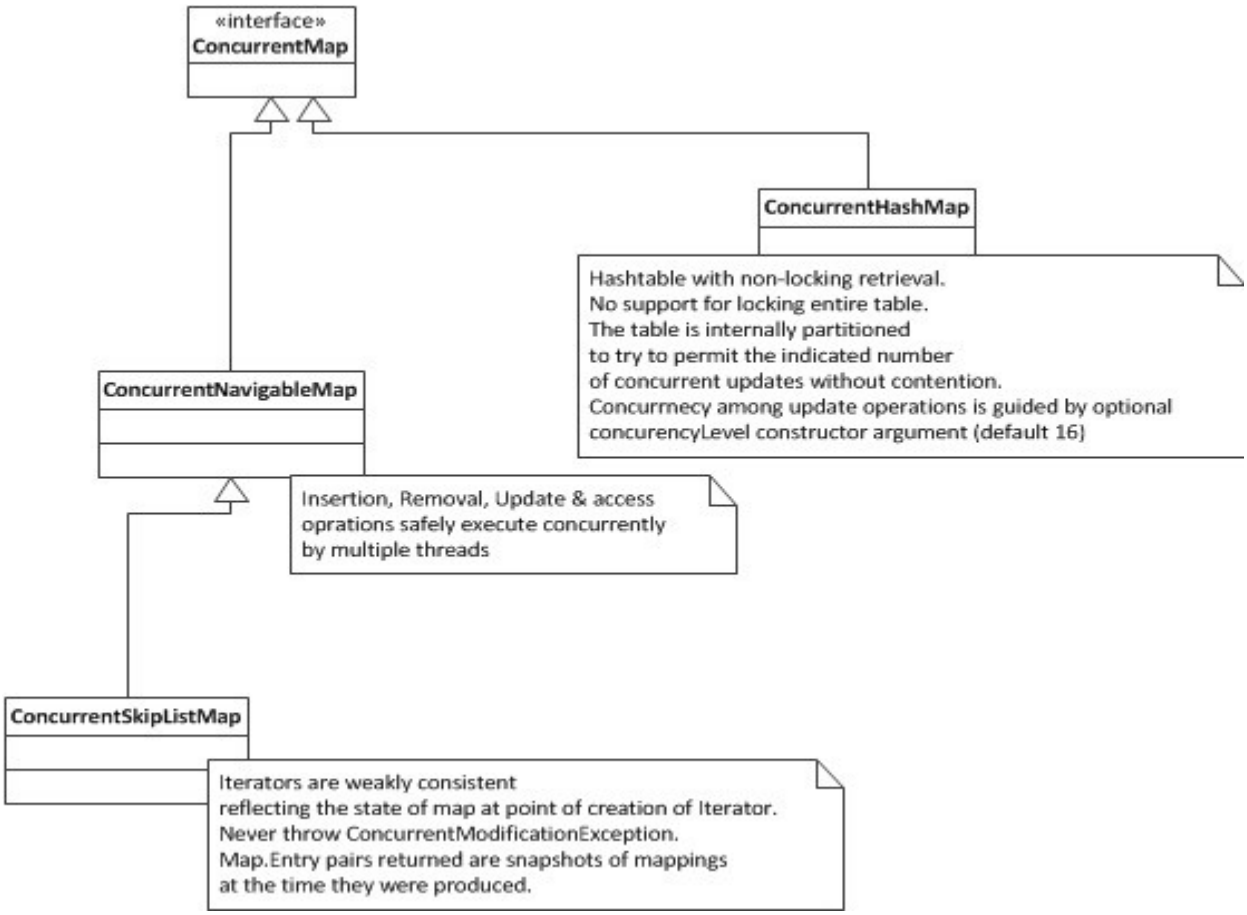
Above are the some basic maps available since very beginning of Java. But with time and as new versions came, new Map implementation are added to core java library, and some major enhancement's were done Concurrent Maps were introduced which provide thread safe maps alternative to Hashtable.



## Sorted Maps


«interface»
Map
SortedMap
Map is ordered according
to the natural order of its keys,
or by a comparator typically
provided in constructor.
Keys must implement
Comparable interface.
NavigableMap
+firstEntry()
+pollFirstEntry()
+lastEntry()
+pollLastEntry()
+subMap()
+headMap()
+tailMap()
TreeMap

## Concurrent Maps




ConcurrentHashMap is the mostly used especially if building a custom Cache for faster read and concurrent write.

## Threads

Threads are everywhere in Java program, as soon as programmer runs his or her program main thread starts along with JVM starts multiple threads; a thread in background for Garbage Collection, another thread to run finalize, and so on. A Java program is always multi threaded: Several threads running in one JVM, that is running in one process. Thread is not virtual; it is an instance of **Thread** class; It has a method **run ()**: contains code that will run.

Creating a thread and running is as simple as below –

```
public class MyThread extends Thread {
    public void run() {
        // ... do Something
    }
}
```

```
// Start the thread
(new MyThread()).start();
```

calling the **start ()** method of Thread class starts the thread, and code inside **run ()** will start executing. Only problem with above code is that, MyThread can't extend any other class; therefore, Java provides a decent alternative –

```
public class MyRunnable implements Runnable {
    public void run(){
        // do Something
    }
}
```

**Thread** class is still required to start execution of code inside **run ()** –

```
(new Thread(new MyRunnable())).start();
```

Write a task with run() method implemented, and submit it to a Thread class. Best part is one can abstract thread management (creating & destroying threads) from rest of the application. Either write code like "new Thread()" whenever need is to create a new thread, or use java provided utilities to manage the creation of thread's and only care about writing the actual tasks, i.e. implementation of Runnable. Executers & Thread pools are Java API's for managing & launching threads for large scale Java applications.

A huge Task can be broken into multiple small tasks and run parallel in multiple threads to fully utilize hardware and to increase responsiveness. But this will bring not only complexity but also some challenges, like –

**Classes should be thread safe:** A class shared by multiple threads should behave correctly when accessed from single thread or multiple thread. Class should encapsulate some synchronization technique, so that clients need not bother and synchronize.

**read-modify-write:** ++i is a concise syntax, but not atomic. It involves a sequence of steps: read current value, add one to it, and store the new value. If i is a counter with initial value 1, and increments when each new thread comes, and If two threads come at the same time, each read value 1, and each set the new value as 2, which instead should be 3, if it is a counter.

**data-race:** Two or more threads access the same memory, where one of those thread is writing, and threads are not using locks, THEN order of access of memory location by threads is non fixed, and so is the result; its nondeterministic algorithm.

**Race Condition:** Right outcome depends on right timing or lucky timing. Example is **check-then-act:** You checked you have sufficient balance; you initiated a payment; but in between balance changed, because of some periodic transfer of payment; and your newly initiated payment failed.

**Java memory model:** Threads are allowed to hold the values of variables in machine register. A thread running in loop, might check the value of a done flag before each run, but might not see the changed value by another thread.

Java provides elegant solution: **volatile** variables - if variable is marked is volatile, it is always read from main memory, and when variable is written it is always stored in main memory. Volatile variables are helpful, when operations are atomic, like while(done) {…}, or done=true. But if operations are not atomic, like i++, just making variable volatile won't help prevent race conditions.

An object's state is its data, values stored in instance variables, static variables, and also stored in fields in dependent objects. If an object's state is shared it means any such variable can be accessed by multiple threads. Making object thread safe means using some synchronization technique to coordinate access to mutable state, and primary mechanism for synchronization in Java is **synchronized** keyword, which provides exclusive locking. If a variable can be modified and read in multiple methods, we mark all those methods as synchronized and thus it is not possible to execute a method in one thread while any of those synchronized method is already running in another thread. But it might come as surprise that even after synchronizing all the methods as in Vector class, is not enough to make compound actions atomic. Following example I took from "Java Concurrency in Practice" book –

Merely synchronizing every method, as Vector does, is not enough to render compound actions on a Vector atomic:

```
If(!vector.contains(element)) vector.add(element);
```

The attempt at a put-if-absent operation has a race condition, even though both *contains* and *add* are atomic. While synchronizing methods can make individual operations atomic, additional locking is required when multiple operations are combined into a compound action.

A synchronized block is needed here.

**Lock:** Please note all the data protection requirements can be accomplished by synchronized keyword. Synchronized allows serial access either to block of code, or methods of an object. The synchronization tools introduced in Jave 5 implement *Lock* interface which has two important methods: **lock()** & **unlock()**. **lock()** method on *Lock* object can be used to grab a lock, and unlock to release the lock.

Big difference is now lock object is explicitly visible, one can store it, one can pass it, one can reuse it and one can discard it. Unlike synchronized where lock is attached to object and synchronized(this) is common way to acquire lock. With Lock, it is possible for two objects to share same lock and for one object to have multiple locks. Unlike synchronized, where lock is automatically released, with Lock object's lock() & unlock(), it is more explicit and we can establish lock scope from single line to multiple

methods & objects.  Also for Lock object it doesn't matter if the method is instance or static. Indeed all complicated cases have to be handled by Lock object. But, the slight overhead of two methods, synchronized can still be preferred if only few lines of code need to be synchronized.

Other powerful methods of Lock class can't ignored are tryLock() - if want to perform some tasks and if can't acquire lock; getHoldCount()- for Reentrant locks; isLocked(); getQueueLength()- get estimate of number of threads waiting for lock; and newCondition() - to create **Condition** object, which brings power to create more than one condition objects per lock object.

Need to note, one can't use wait () & notify () of Lock object. *Condition* (with useful await () & signal ()) object represents the condition that's represents the lock. Multiple conditions can be created targeting separate group of threads. Its complex, but bottom line is Lock object represents Lock, but synchronization lock associated with Lock object is different.

```
Lock l = new ReentrantLock();
Condition c = l.newCondition();
..
run() {
    l.lock();
    while(running){
        if(done){
            c.await(); // done so await,
                       // unless someone else
                       // sets undone
        }else{
            // if not done, handle
        }
    }
    l.unlock(); // will normally go in finally block
}

unDone(){   // called by someone who wishes to set undone.
    l.lock();
    c.signal();
    l.unlock();
}
```

Before actually start using synchronization techniques, know that some things are generally thread safe:

- ThreadLocal
- Local variables
- Immutable objects
- Atomic Integers

- Safe publication of mutable objects
- Concurrent collections

**Semaphores:** Class implements grant & release algorithm, a lock class with attached counter but no attached condition variables. It is used to control number of threads working in parallel, with each thread previously called **acquire ()** to get permit to run.

**Barrier:** Barrier is the point where threads wait. When barrier is created a number is passed to constructor that specifies the number of threads participating. Threads come & wait at barrier until number of threads waiting reaches number n, then possibly an action can run (optionally **BarrierAction** can be passed to Barrier constructor), and then all threads are released.

**Countdown Latch:** Class with a counter and method to decrement the counter, threads come & wait till counter reaches zero and get released. Multiple threads or even same thread can call the method countDown() multiple times, to decrement the counter. *BarrierAction* can't be defined here and also getCount() doesn't gives exact picture of number of threads waiting, because countdown() can even be called by non-waiting thread.

**Exchanger:** Synchronization tool to exchange data between pair of waiting of threads. Two threads are paired in the order of arrival, any two threads which call **exchange (V x)**.

**Reader/Writer Locks:** One thread is allowed to write data at a time, but multiple threads can read simultaneously.

**Callable:** Can do everything Runnable does, but difference is call() method returns object & can throw exception. Indeed Java folks probably didn't wanted to change signature of run() method, so this more powerful call() method came.

(Oaks & Wong, 2009), (Goetz, 2006)

## Executor Service

**java.util.concurrent.ExecutorService** is similar to thread pool, which can execute tasks in background. **ExecuterService** implementation in Java is **ThreadPool**: **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor**.

A thread T can delegate some of its tasks t1, t2; to **ExecuterService** and continue with other tasks, and T execution becomes independent of t1 and t2, and possibly T, t1, and t2 can run in parallel.

Delegating a task to ExecutorService is simple:
1. Instantiate ExecutorService
2. Submit a task and optionally
3. Get notified when task completes.

*ExecutorService* will be a *Threadpool*: A fix number of running threads which can accept a task (which provides *run ()* method) and thus minimize the overhead of thread creation each time a task needs to be run.
If we got Multiple Tasks, yeah we have to be multitasking, not exactly we, it's our program.
Assume we have 4 users for our application which sequentially but time difference is in milliseconds. We have a task to run for each user. We can handle multiple tasks at a time, and we got two approaches

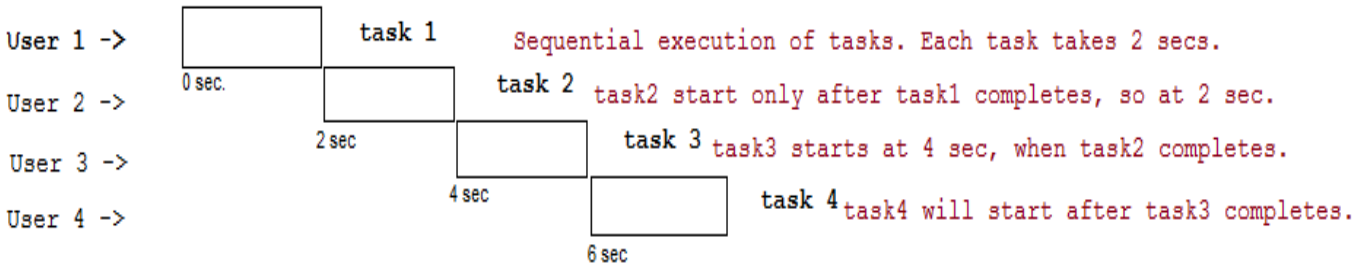
### Approach 1-

Assume a task takes 2 seconds to complete and for each user a new thread is started, which runs the task. If 4 users come at the same time and 4 threads are started, each thread will take 2 seconds. If machine is one CPU, then all 4 threads will start and run in a time sliced manner according to OS scheduling. OS will schedule and run 4 threads in its own way, in which multiple outcomes are possible regarding completion time of each thread. One thread will run for some time (like few milliseconds) and then another but at a time only one will run. Possibly, all four threads complete execution almost at same time (7.7, 7.8., 7.9, and 8 seconds). Issues in this approach are:

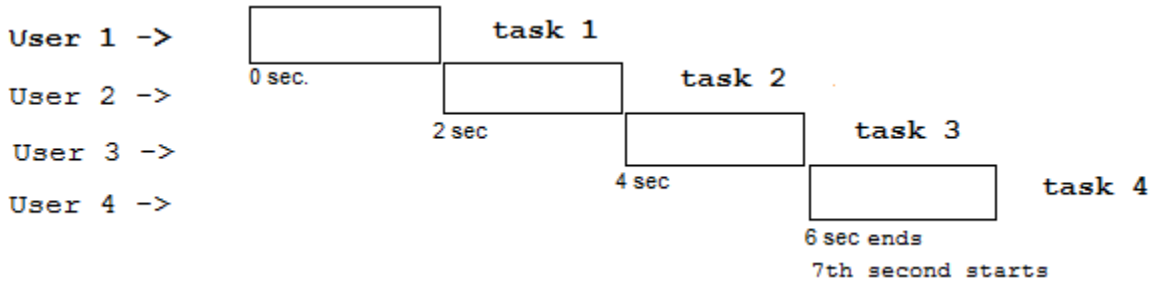
- Threads start at nearly same time in some random order and they will end in nearly same time but in different random order.
- As number of users will increase, then waiting time (completion time of each thread) will increase

### Approach 2 -

Another approach is executing tasks sequentially, so that order is guaranteed and waiting time is guaranteed (2 seconds for each user)




What is shown in above picture is – Sequential execution of tasks, each task takes 2 seconds. Task2 starts only after task1 completes, so task2 starts at end of 2nd second; similarly, task3 starts at end of 4th second. Task4 starts at 7th second, which means User4 has to wait for 6 seconds.




Each time a new user comes, a task is submitted to the thread for execution. Task completes in 2 seconds and while thread is running any new user is put in queue. Therefore, even if 4 users came at same time, User 1's task starts execution and rest 3 users are placed in queue. Users are served on first come first served basis. Queue will have a limit and any incoming user will be rejected, when queue is full. Issues in this approach are:

- Server/Application has limits
- User coming later will have to wait a lot.

Scaling the application will require two things:

- Rather than only one thread we will create a thread pool (fix number of pools) and run multiple tasks in parallel, utilizing multiple CPU's.
- Increase the size of Queue

If 4 threads can run in parallel and queue has size of 100; then if 100 users come at same time; then 4 threads will start running in parallel and 96 users will be placed in queue. Thus, 100 users will be served in 50 seconds. Application can be further more scalable with creating more instances of application and improving the performance of application.

(Oaks & Wong, 2009)

## Thread Pool configuration

Most important is to decide on two numbers - Number of threads in thread pool & Queue size.
Below key points need to considered while creating & sizing thread pools -

- If thread pool is too big, threads will compete for CPU, memory & resources and might result in resource exhaustion & errors.
- If thread pool too small, then CPU, processes, resources will go unused.
- Understand the tasks: Do they need resources like JDBC connection or MQ connection ?; Do they perform lot of computation ?; Do they perform intensive I/O ?; Do they need significant memory ?; OR they are just few lines of statements ?.
- Know the number of processes available or number of CPUs - Runtime.getRuntime().availableProcessors()

Pool size should be configurable based of N_CPU and how compute intensive tasks are.
Several calculations are available, but for normal tasks we can define thread pool size to be N_CPU+1, and N_CPU-1 for highly compute intensive tasks. (Goetz, 2006)
A thread pool with N max threads, will keep on creating threads as long as it gets tasks, till number reaches N. If pool has N threads and all threads are busy, any new task and all subsequent tasks will be placed in queue. Queue can resize dynamically, but if queue is full and cannot be expanded, the any incoming task will be rejected. Ideally Queue used by Pool's should be bounded queue (fixed capacity), but optionally it can be unbounded (unlimited capacity) or even Synchronous Queue (0 capacity) - task is run immediately or rejected immediately.

> Java class ThreadPoolExecuter executes each submitted task using one of possibly several pooled threads. Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.[vi] ScheduledThreadPoolExecutor can additionally schedule commands to run after a given delay, or to execute periodically.[vii] ScheduledThreadPoolExecutor extends ThreadPoolExecuter and implements ScheduledExecutorService. ThreadPoolExecuter implements Executer - An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor[viii] is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new(RunnableTask())).start() for each of a set of tasks, you might use:
>
> ```
> Executor executor = anExecutor;
> executor.execute(new RunnableTask1());
> executor.execute(new RunnableTask2());
> ```
>
> Reference: Oracle Java docs

## Creating ThreadPool

ThreadPoolExecutor are normally configured using Executor factory methods.

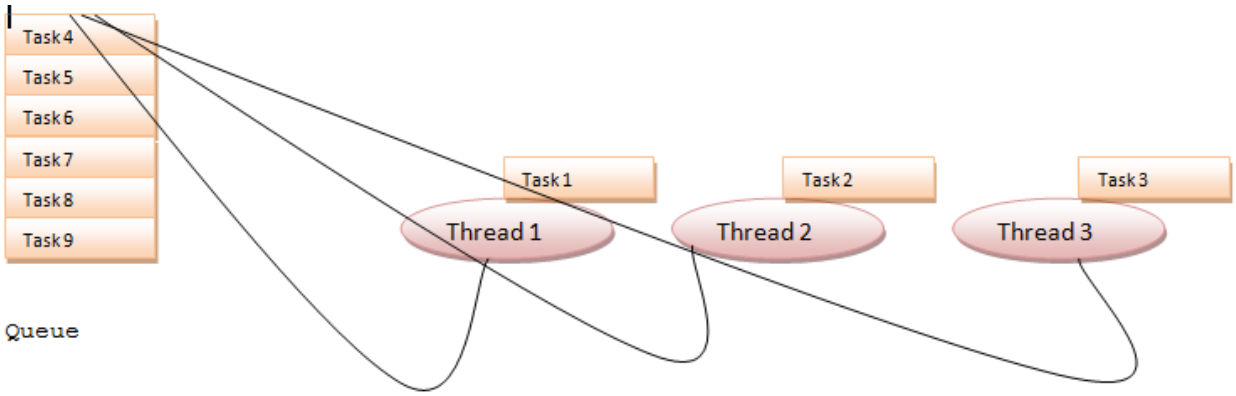
| | |
|---|---|
| newFixedThreadPool(int n) | Creates thread pool with max. n threads & unbounded queue. |
| newSingleThreadExecutor() | Single worker thread operating off an unbounded queue.<br>Tasks gauranteed to be executed sequentially |
| newCachedThreadPool() | Creates new thread as needed, but will reuse previously constructed threads when they are available.<br>Threads not used for 60 seconds are terminated & removed.<br>Improved performance if many short lived asynchronous tasks. |
| newSingleThreadScheduledExecutor() | Single thread executor that can schedule commands to run after given delay. |
| newScheduledThreadPool() | Creates thread pool that can schdeule commands to run after a given delay. |

## ThreadPoolExecutor

Automatically adjusts the pool size based on bounds (maxPoolSize and corePoolSize) set. Normally minimum number of idle threads (corePoolSize) is maintained.
New threads with NORM_PRIORITY & non deamon status are created if new tasks arrived.
Excess threads (threads more than corePoolSize, and idle) are terminated after keepAliveTime.
Any blockingQueue may be used to hold submitted tasks.
Rejected tasks are handled by RejectedExecutionHandler.
Hook methods: beforeExecution() & afterExecution() can be called before & after execution of each task.

## ForkJoinPool

Fork/join algorithm is a way of divide and conquer in which if a task is big then break it into new subtasks (fork), solve each subpart in parallel, and when all subtasks are done join them (join). (Lea, 2000). For example, sorting a huge array can be broken into multiple parts and joined later.[ix] Fork/Join is different from other Executer Service by virtue of employing work-stealing, that is a thread if free should pull a subtask of another busy thread. Normally, several active threads will pick tasks from a Queue in FIFO order.



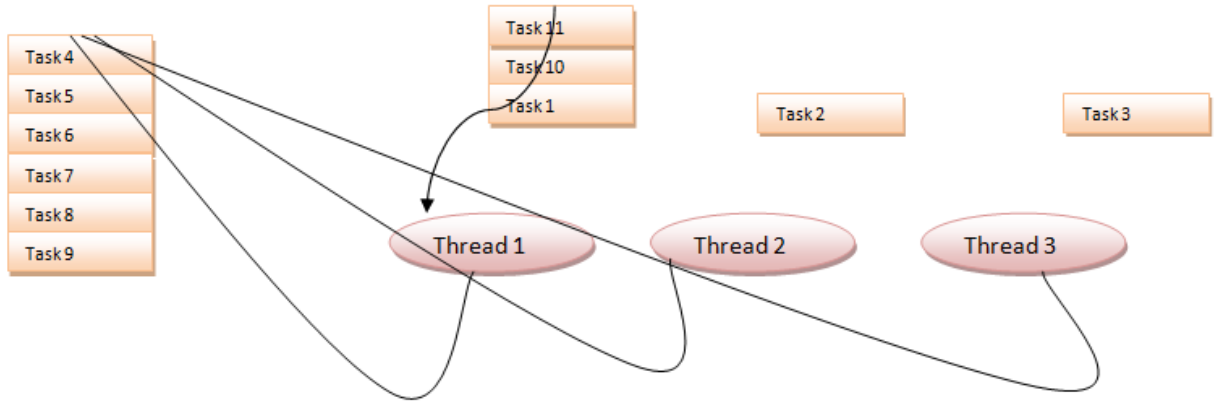

Not all tasks are equal and some thread will be suck in long running tasks. In Fork/Join algorithm bigger tasks can be broken into several subtasks and sub tasks can be completed independently and later joined after completion. For example:

- Break Task1 (a big task) into Task10 and Task11
- Put Task10 and Task11 in a queue, not global queue but a local queue of thread (assume Thread 1) executing Task1.
- Other free threads can pick task from global queue or pick from local queue of Thread 1.
- Once Task10 and Task11 are finished, join them in order to finish Task1.

Subtasks should finish first; therefore, subtasks are put in Deque.

"Apart from Global Queue, Each Worker thread maintains a Deque. Tasks are popped from Queue in FIFO manner, but popped from Deque in LIFO manner by its worker thread".

Picture is like below -



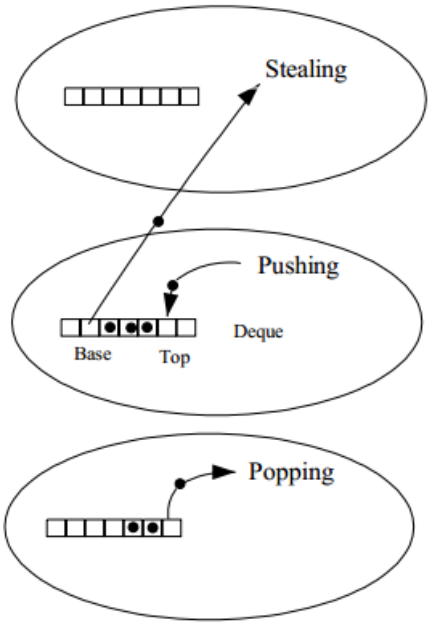

Why we used Deque rather than Stack is interesting, because in heart Fork/Join Algorithms is Work-Stealing algorithm, which means If a worker thread doesn't have a task to run, it will steal a task from another thread's Deque.

A worker thread pick tasks from its own Deque in FIFO manner, but picks tasks from another thread's Deque in LIFO manner, and it is mentioned in the Doug Lea's paper -

## 2.1 Work−Stealing

The heart of a fork/join framework lies in its lightweight scheduling mechanics. `FJTask` adapts the basic tactics pioneered in the Cilk work−stealing scheduler:

- Each worker thread maintains runnable tasks in its own scheduling queue.
- Queues are maintained as double−ended queues (i.e., deques, usually pronounced "decks"), supporting both LIFO `push` and `pop` operations, as well as a FIFO `take` operation.
- Subtasks generated in tasks run by a given worker thread are pushed onto that workers own deque.
- Worker threads process their own deques in LIFO (youngest−first) order, by popping tasks.
- When a worker thread has no local tasks to run, it attempts to take ("*steal*") a task from another randomly chosen worker, using a FIFO (oldest first) rule.
- When a worker thread encounters a `join` operation, it processes other tasks, if available, until the target task is noticed to have completed (via `isDone`). All tasks otherwise run to completion without blocking.
- When a worker thread has no work and fails to steal any from others, it backs off (via yields, sleeps, and/or priority adjustment − see section 3) and tries again later unless all workers are known to be similarly idle, in which case they all block until another task is invoked from top−level.




If more information needed then google the Doug Lea's paper on Fork-Join.

## JMX

Using JMX, any resource is instrumented by one or more Java objects known as Managed Beans or MBeans. Mbeans are registered with MBean server (which is managed by JMX agents). Remote applications connect to JMX Agent via JMX connectors. Use JMX to detect memory & thread usage, generate heap dump. JMX is used for monitoring, like monitoring properties changes, which means monitor properties file. Monitor for events, performance, errors, statistics, memory, properties,etc.

JMX can be packaged in EAR. MBeans can generate notifications, for example to signal a state change, a detected event, or a problem.

> JSR 262 defines - "It will provide a way to use the server part of the JMX Remote API to create a Web Services agent exposing JMX instrumentation, and a way to use the client part of the API to access the instrumentation remotely from a Java application. It will also specify the WSDL definitions used so that the instrumentation will be available from clients that are not based on the Java platform"

An MXBean (@MXBean used over interface) is a new type of MBean that provides a simple way to code an MBean that only references a pre-defined set of types. In this way, one can be sure that MBean will be usable by any client, including remote clients, without any requirement that the client have access to model-specific classes representing the types of your MBeans. The exact mapping rules appear in the MXBean specification, but to oversimplify we could say that complex type beans like MemoryUsage (example given in JavaDocs) get mapped to the standard type CompositeDataSupport. Below example is taken from Java docs and explains the code for MBeans and MXBeans both and also the difference. MemoryPool is a managed object which has complex MemoryUsage object.

### Standard MBean

```
    // Standard MBean
public interface MemoryPoolMBean {
    String getName();
    MemoryUsage getUsage();  // A complex object
    //...
}

// Client code
String name = (String)
            mBeanServer.getAttribute(objectName, "Name");
MemoryUsage usage = (MemoryUsage)
            mBeanServer.getAttribute(objectName, "Usage");
long used = usage.getUsed(); // a getter of MemoryUsage object
                             // getUsed() returns memory used.
```

## MXBean code

```
      // MXBean
 public interface MemoryPoolMXBean {
    String getName();
    MemoryUsage getUsage();  // A complex object
    //...
}

// Client code
String name = (String)
            mBeanServer.getAttribute(objectName, "Name");
CompositeData usage = (CompositeData)
            mBeanServer.getAttribute(objectName, "Usage");
long used = (Long) usage.get("used");
```

## Internationalization

This is extremely useful Java feature for developing products which should adapt to different languages & regions. Like same application when viewed in Germany the labels should be German; and when same application viewed in USA the labels, messages etc should be in English, indeed in en_US. Dates & currencies also appear in local format, like 100,000.00 in USA and 100.000,000 in Spain. Adding a new language support doesn't requires recompilation. A program will have different labels for different countries, defined in properties file, and program will choose the label based on location.

To run the below program create four properties files: Messages_en_US.properties, Messages_fr_FR.properties, and so on –

```
String[] languages = { "en", "fr", "de", "ru" };
String[] countries = { "US", "FR", "DE", "RU" };
for (int i = 0; i < 4; i++) {
    Locale locale = new Locale(languages[i], countries[i]);
    ResourceBundle messages = ResourceBundle.getBundle("Messages",
            locale);
    String greetingMessage = messages.getString("greetingMessage");
}
```

Locale is created with combination of language and country. Locale is passed to other objects which do real work. Any random combination of language & country in locale, might fail those objects. It can also be concluded from program that locale specific data is contained in ResourceBundle, in fact a set of ResourceBundle classes with same base name. Below code shows how to format numbers:

```
String[] languages = {"en", "fr", "de"};
String[] countries = {"US", "FR", "DE"};
for(int i=0; i<3;i++){
    String language = languages[i];
    String country = countries[i];
    Locale locale = new Locale(language, country);;
    Double amount = new Double(100010.99);
    NumberFormat numberFormatter;
    String amountOut;
    numberFormatter = NumberFormat.getNumberInstance(locale);
    amountOut = numberFormatter.format(amount);
    System.out.println(amountOut + "    " + locale.toString());
}
```

```
Console
<terminated> I18N [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Aug 16, 2014, 9:41:25 AM)
100,010.99    en_US
100 010,99    fr_FR
100.010,99    de_DE
```

Above were some examples, but Java Internationalization is a very powerful tool and opens many avenue's: format currencies, Date, time, messages, and more -

- Character class provides character comparison methods which can be used independent of language.
- Collator class facilitates string comparisons for different languages.
- BreakIterator can give character, word, sentence, and line breaks to help in locating the boundaries in texts.

# I/O

At the core of Java Input/Output is I/O stream: **Input Stream -** Read sequence of data (byte by byte) one by one and **Output Stream:** Write sequence of data (bytes) one by one.

A program to read data from file is very simple: Use FileInputStream (byte stream class)

```
FileInputStream fis = new FileInputStream("input.txt");
```

**read()** method returns a byte of data, and returns -1 if the end of file is reached. The method blocks if no input is yet available. **fis.close()** is important and must be written in finally block.

While reading texts from file, character reader more suitable than byte reader; therefore, **FileReader** and **FileWriter** are used instead of **FileInputStream** or **FileOutputStream**. *FileReader* reads one character (16 bits) at a time, instead of *FileINputStream* which reads one byte (8 bits) at a time.

Java provides Buffered I/O streams, which read & write data in memory buffer, and call native API (disk or network access) when buffer is empty or full.

```
BufferedReader inputStream = new BufferedReader(new FileReader("input.txt"));
```

Scanner is further useful class to break input into tokens, default delimiter is whitespace.

```
new Scanner (new BufferedReader (new FileReader ("input.txt")));
```

java.io.Console introduced in 1.6, represents the System.console(), provides methods to read & write on console, provided JVM is started from interactive command line.

To read & write double for example, use DataInputStream & DataOutputStream

```
new DataInputStream(new BufferedReader(new FileReader("input.txt"))).readDouble();
```

During Serialization, *ObjectInputStream* & *ObjectOutputStream* are used to read & write objects.

## NIO

**New IO** is new api and is different from IO api, in that it is buffer oriented (not stream oriented) and it can be non-blocking (Java IO threads blocks while read() or write()).

**Channels** & **Buffers** are concepts in NIO. Channel is like bi-directional stream, in which data can be read or written but only through Buffer object. Buffer is object which holds some data with get/set operations. Data is read into buffer by channel, and data is read from buffer. Benefit is buffer has variables position (index in array), limit (how much is left), and capacity (of buffer); In buffer one can move back & forward; Buffer provides several get and put operations to read & write to buffer; One can slice buffer and create duplicate buffer; Scatter/gather which means read & write using multiple buffers.

One can **lock** the entire file or portion of file (specify start & end of portion), reason might be to perform atomic write. One can even acquire shared lock, so that others can also acquire shared locks.

A thread requests data from Channel and gets what is available or nothing, unlike IO call which remains blocked until data is available.

A thread can read data from multiple channels using **Selector**. Register multiple channels with a Selector, thread monitors selector, thread selects the channel that has data to be read OR selects the channel which is ready for writing.

**NIO package**

Basic classes of NIO package are Path and Files.

Path represents the path in file system, including file name & directory structure. One can create path, modify path, merge paths, compare paths, etc. using Path class.

Files provide powerful static methods for reading, writing, deleting, i.e. manipulating files, verifying: if file exists or not, and reading complete file (if file is small) in one go: readAllBytes(), create regular and temporary files. Files also provides metatdata, information about files via methods like size(), isDirectory(), getLastModifiedTime(), setOwner(), etc.

But pivot is SeekableByteChannel, a ByteChannel with position, providing capability to move to different positions in a file, and thus facililates Random Access.

FileChannel implements SeekableByteChannel with additional features, such as mapping a region of file directly into memory for faster access (helpful for large files), locking portion of file, data can be written to absolute position irrespective of current position, and bytes can be transferred from a file to come other channel, and is safe to be used by multiple threads.

FileChannel can be obtained from FileInputStream, FileOutputStream or RandomAccessFile.getChannel().

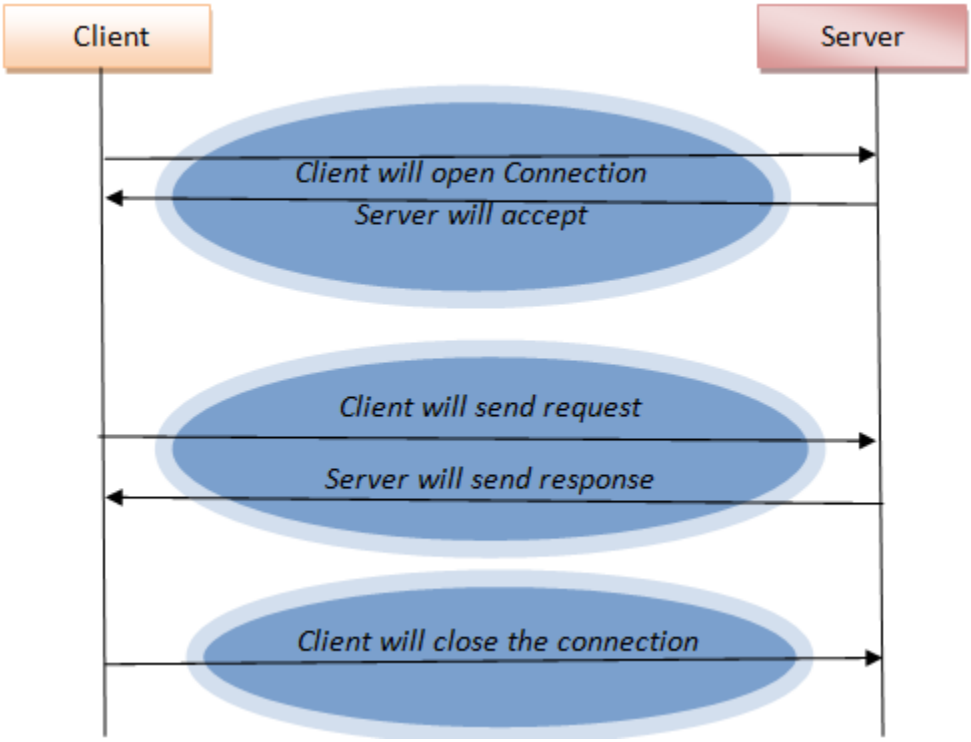
RandomAccessFile is large array of bytes with cursor.

Code for reading data using FileChannel is simple –

```
RandomAccessFile aFile     = new RandomAccessFile("nio.txt", "rw");
FileChannel      channel   = aFile.getChannel();
ByteBuffer       buf       = ByteBuffer.allocate(24);
int              bytesRead = channel.read(buf);
```

## Socket

Java Socket API is used to write a low-level network communication in a client-server application AND a typical Single client and server communication will happen like below -




Above picture displays the Client-Server communication.

### Client.java

```
Socket client = new Socket(host, port);
OutputStream out = client.getOutputStream();
out.write("request");
client.close();
```

### Server.java

```
ServerSocket s = new ServerSocket(port);
Socket c = s.accept();
InputStream in = c.getInputStream();
OutputStream out = c.getOutputStream();
while(in.readLine()){
        out.write("response");
}
s.close();
```

Above is a Single threaded Server which accepts a client's connection and process client request, later client will close connection and Server will serve next client, Sometime later Server can be closed. Visibly we can have performance problems because server will process only one client at a time, which can be resolved if Server process client's requests in separate thread. Idea is to create a new thread for each client, and close the thread when client disconnects. If number of client increases it will be noticeable the number of threads, which we will want to limit, i.e. number of parallel client requests handling, therefore, a thread pool is suitable here.

A multithreaded Server code will look like below –

```
while(! isStopped()){
     Socket clientSocket = null;
     clientSocket = this.serverSocket.accept();
    this.threadPool.execute(
        new HandleClientRunnable(clientSocket,
                    "Thread Pooled Server"));
 }

HandleClientRunnable implements Runnable {
    Socket clientSocket;

    HandleClientRunnable(Socket clientSocket){
        this.clientSocket = clientSocket;
    }

    public void run() {
        InputStream input  =
                clientSocket.getInputStream();
        OutputStream output =
                clientSocket.getOutputStream();
        //...Read input
        //...Write output
        //...Close input & output stream
    }
}
```

UDP can be wise choice over TCP especially if performance is more critical than reliability. Till now we saw Java Networking API for TCP and while working with UDP, we first create DatagramPacket using InetAddress and send/recive it over DatagramSocket, as simple as below –

```
DatagramSocket datagramSocket = new DatagramSocket();
byte[] buffer // byte array
InetAddress receiver = InetAddress.getLocalHost();
DatagramPacket dgPacket = new DatagramPacket(
    buffer, buffer.length, receiver, 80);
datagramSocket.send(dgPacket);
```

Java programs interacting with internet may use URL class to connect to a resource with web address, using simple code below –

```
URL url = new URL("http://shekup.blogspot.com");
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
int data = in.read();
```

Above code sends a GET request to mentioned URL. Please note URL is preferred over Socket, when connecting to Web.

## Exception Handling

Exception is an object. All exception types are sub class of Throwable: Exception & Error are two main sub classes of Throwable. The constructor takes String message, which is the detailed description of exception that this Exception will deal with, and getMessage() called on this Exception class will return that message only. printStackTrace() is more powerful and gives actual runtime stack trace (method invocation sequence) and highly helpful in debugging for the reason of exception and to reach exact line of code from exception originated.

**Errors** are irrecoverable so never explicitly caught, like VirtualMachineError, LinkageError, NoClassDefError, StackOverflowError, OutOfMemoryError, etc. There is no point in catching such errors and move forward. Errors are **UnChecked** Exceptions, there is no need for compiler to check if they are properly dealt.

**Exception** class defines exception conditions that program should deal with, i.e. catch it and handle it, and continue with execution. In some exceptional scenarios like Programming error, you may not need to catch exception. A programming error or faulty design should not be catch and handle by program in runtime, it should also be handled by default exception handler. **RuntimeException** and all its subclasses: ArithmeticException, IllegalStateException, NullPointerException, etc. are programming bugs. RuntimeException & its subclasses are **UnChecked** Exceptions.

All other Exceptions are **Checked** Exceptions. Compiler will throw error if they are not dealt; means the method throwing such exception should **catch** it, or include it in **throws**.

Apart from normal *try-catch-finally* statement, *try*-with-resources was introduced in Java 7. Previously finally block was used for closing resources like BufferedReader. Now in Java 7, BufferedReader implements interface AutoCloseable, so code like below –

```
try (BufferedReader br =
                    new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
```

BufferedReader will be closed whether try ends normally or it throws exception.

# Garbage Collection

**Object** is an instance of class and is referred by a reference. An object can have more than one reference, is called reachable. Objects consume memory and stored in heap. An object in heap, which has no reference, is non-reachable. Garbage Collector will reclaim the memory of those non-reachable objects, and thus thereby continuously freeing the memory. But it still remains the programmer's responsibility to make object non-reachable if it is no longer needed. Runtime stack of thread contains the activation records (method calls) and they contain the reference to the object's storage in heap.

**Garbage Collector (GC)** will call the object's finalize() method always, before actually destroying the object. Any exception thrown in **finalizer** will be ignored. Inside the finalize() method, we can resurrect the object, however finalize() method will be only called once, thus if object again becomes eligible for garbage collection, finalize() won't be called. Indeed object will be garbage collected even though it is non-reachable is not guaranteed, since it is not guaranteed that Garbage collector will run, and call of finalize() method is also not guaranteed. Java provides gc() and runFinalization() operations to run GC and to run pending finalization, even though it's not recommended to call these methods from programs. There is no guarantee about order in which objects will be garbage collector, or order in which finalize() will be called, and lastly it is also not guaranteed that continuous GC runs will prevent OutOfMemory.

## Garbage Collection Algorithm

An algorithm to clean the garbage involves identifying the garbage, removing the garbage and do not leave memory fragmented. Several algorithms include reference counting, tracing collector, Mark-sweep, copying, Mark-compact, or combination of more than one. Many studies say as many as 98% objects die **young**, shortly after their creation, some are **old** objects which survive many wall-clock seconds, and some never die - **permanent**. We should run garbage collectors more frequently for young objects compared to old objects; we can use different algorithm for collecting young objects and different algorithm for collecting old; if any young object passes several clock cycles we can assume it to be old; and these assumptions lead to Popular technique Generational Collection: memory is divided into several regions (Young, Old, and Perm); separate regions hold objects of different ages. When object is created it is put into Young region, and if it survives several garbage collection cycles, it is moved to Old generation. Garbage Collection cycles run less frequently in Old space. In reality lot more complexities are involved: like how much memory should be kept for young, old, and perm; which algorithm to use to collect garbage in each generation; if we want collectors to run in parallel; If we can afford a stop-the-world collection and pauses; and more. All these complexities lead to several options which we can provide to Garbage Collector, and possibly tune it.

Among all those Garbage collection algorithms, we will only discuss **Garbage First (G1)**. Complete heap is divided into fix size regions, i.e. one large contiguous space divided into many (target 2048) fix sized (1MB to 32MB) regions. Some regions belong to Young generation & some belong to old, that means it is generational, but young or old generation regions are not contiguous, and size of individual

generation is dynamic. Often Young generation consists of Eden (creation) and Survivor (used for copying, before promoting to old generation).

Along with Eden, Survivor, and old, also are humongous region and Unused region. If object size is more than 50% of the size of the region, it is stored in humongous region. If object size is more than region, multiple regions can be concatenated, but never make objects so big. Freeing up the humongous region is not so optimized, so you may think of increasing the size of region using option -XX:G1RegionSize, especially if you are dealing with mostly large objects.

At any moment, Java heap may look like below –

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| O | | O | E | O | E | | | Eden |
| | E | E | | S | | | | Survivor |
| E | O | S | | O | | | | Old |
| | O | E | E | O | | | | Humongous |
| S | S | O | O | | O | | | |
| O | O | | H | | E | | | |
| H | H | | | | O | | | |

Some Key concepts involved are -

- On minor GC (multi threaded, stop the world, young generation collection), live objects are copied from eden & survivor to unused/available regions which become new survivor regions.
- Some number of eden regions remain eden for next round of allocation, until another minor GC occurs.
- Number of eden regions depend on pause time targets and some internal heuristics. Number of eden regions can change between minor GC cycles.
- Once overall heap occupancy reaches a threshold, a concurrent cycle is initiated - start of old generation collection - based on region liveness information - The regions with least amount of live objects are collected first (Garbage First).
- Collection of old generation happens at the same time as minor GC -Mixed GC - Collection of old generation and young generation at the same time.
- Remembered Set (RSet) - Number of object references into a given region.
- Collection Set(CSet) - The set of regions to be collected on a given GC.
- IHOP - Entire java heap occupancy. If reaches certain percentage limit, a concurrent cycle is initiated.

Below are some basic command line options –

-XX:+UseG1GC, “bare minimum”
-Xms, -Xmx, if heap size desired differs from default chosen by the JVM
-XX:MaxGCPauseMillis, if desired pause time different from 200ms
-XX:InitiatingHeapOccupancyPercent (aka IHOP), default is 45% of total Java heap occupancy

## Security

Language design decisions can really influence both the kind of bugs that can appear in a program, as well as the impact that those bugs can have on the overall security of the system. A lot of the trend today is towards having more automation at the language level-- having the language do more for you. And by being able to do that, reduce the risk of security problems. Using standard libraries or packages will reduce the programmatic errors. The choice to use Java safeguards from few security problems that other languages, dynamic languages, suffer such as Python and JavaScript, because Java is a type safe – every object and variable has a type. Java is a mature language and it is well documented the behavior. It is programmer's responsibility to take extra care of unchecked exceptions, deadlocks, race conditions, and so on that may lead to undefined behavior of the Java program.

Java provides a customizable "sandbox" for Java Applets in which untrusted Java programs run. The sandbox prohibits reading or writing to the local disk, making a network connection to any host except the host from which the applet came, creating a new process, loading a new dynamic library and directly calling a native method. By making it impossible for downloaded code to perform certain actions, Java's security model protects the user from the threat of hostile code. The fundamental components responsible for Java's sandbox are:

- Safety features built into the Java virtual machine (and the language)
- The class loader architecture
- The class file verifier
- The security manager and the Java API

One of the greatest strengths of Java's security model is that two of the four components shown in the above list, the class loader and the security manager, are customizable. While the Java security architecture can protect users and systems from hostile programs downloaded over a network, it cannot defend against implementation bugs that occur in trusted code.

Secure Coding Guidelines[x] has very detailed instructions on writing secure code starting from fundamentals: design APIs to avoid security concerns, avoid code duplication, restrict privileges, establish trust boundaries, encapsulate, and document safety related information.

**Denial of Service**: Inputs to the system can cause expensive compute cycles or resource consumption such as CPU, memory, disk space, and file descriptors. It is reasonable to check the input for what it is and what it is requesting. Always resources and resource limit checks should not suffer from integer overflows.

**Sensitive Information**: Sensitive information should not leak from exception stack trace, logs, heap dumps, and memory

**Injection**: Valid format of input, avoid dynamic SQL, sanitize data before including into HTML or XML, untrusted data from command line, most restrictive configuration possible for XML parser, and so on.

**Accessibility**: Limit the accessibility of packages, classes, interfaces, methods, and fields; Limit exposure to class loader; Limit the extensibility of classes and methods with final if required; Understand how superclass behavior affects sub classes.

Other major topics from Security perspective are Immutability, Deserialization, and Access Controls.

## Security in Java EE

Basic concepts of security are authentication (authenticate user by his or her username and password), authorization (if user is authorized to access the resource), confidentiality (data is not seen by unauthorized users), and data integrity (data is not modified during transport). Whole mechanism of security (called security realm[xi] in WebLogic) consists of users, groups, roles, policies, database, and security provider. User can be person or application and one or more users belong to group. Role is the privilege granted to user or group. Policy (like access control list) is attached to the resource that wish to be constraint. Security Provider provides the security service and it maintains a database. When the container (server) receives request for an URL, it checks if URL is in security table and URL exists in security table that means URL is constrained and policy is attached to it. Container will validate the user name and password (part of user request) to authenticate and check the role of user for authorization check.

A Java EE module such as Servlet, JSP, EJB, or Bean is a component and security (who can access the component) is defined in deployment descriptors (XML or using Annotations in Java EE 6 onwards). Developer provides the entries in the deployment descriptor and Application (or Web) Server (such as Tomcat, JBOSS, etc.) takes care of runtime authentication and authorization checks.
Following code snippet instructs Server to allow users with role *manager* and/or *admin* to access constraint resources under URL */mgr/:*

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>managers</web-resource-name>
                <url-pattern>/mgr/*</url-pattern>
                <http-method>GET</http-method>
                <http-method>PUT</http-method>
                <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
                <role-name>manager</role-name>
                <role-name>admin</role-name>
        </auth-constraint>
</security-constraint>
<security-role>
        <role-name>manager</role-name>
        <role-name>admin</role-name>
</security-role>
```

Container maps the role names in DD to the role names defined inside the container, such as in Tomcat, a file named tomcat-users.xml has roles and they get mapped to role names in DD. The term for the place in Tomcat, where containers stores user, password, and roles is realm.
When two different <auth-constraint> are applied to same resource then access is granted to union of all roles from both the <auth-contraint> elements.
*HttpServletRequest* has three methods for programmatic security: *getUserPrincipal()*, *getRemoteUser(),* and *isUserInRole().*

Servlet specification defines standard authentication mechanisms:
**BASIC**, HTTP Basic Authentication
**DIGEST**, Digest Authentication - User sends a digest of password instead of plain text.
**CLIENT-CERT**, client and server authenticate each other using certificates
**FORM**, Form based authentication
**JASPIC**, Custom such as Open ID or OAuth

```
<login-config>
        <auth-method>xxx</auth-method>
</login-config>
```

Transport security ensures that no one tamper the data during transit. Java EE allows to set transport security as CONFIDENTIAL (data is encrypted), INTEGRAL (data is not modified), or NONE (this level does not apply SSL).

```
<user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

When specified CONFIDENTIAL or INTEGRAL as transport guarantee level, the application server will use the HTTPS listener (HTTP listener with SSL enabled) to communicate with the client. (Tijms, et al., 2017)

## OOPS

**Encapsulation**(eg. Class); **Inheritance**(via implements and extends, to inherit existing properties of class); **Polymorphism**(Static-Overloading, Dynamic-Overriding); **Association**(class having reference to other classes).

**Interface** defines the contract between the class and outside world. Its implementation is the Object to provide functionality. Then we have requirement to override or change some functionality, to we extend this Object to create a sub type. If we are going to override some functionality, inherit some behavior, and you are creating a sub type, and inheritance is the design. But if you are overriding lot of functionality or you are overriding nothing, or you two are unrelated but one calls another, aggregation is better design

**Aggregation** is perfect design for situations like -

You plug some smaller things into something bigger, bigger objects calls the smaller objects back, and thus reducing coupling.

You have layered architecture, where Controller calls service Impl and also has Model objects. Controller has-a service and has-a model.

An **Iterator** inside List, doesn't exit's outside and it ends when List life ends. This is a more restrictive form of aggregation called **composition**.

## Design Patterns

**Creational Design Patterns**

Use **Factory** if you want to provide an interface for creating object. Interface instantiates appropriate object and returns to Client. Like for example I want a HandlerFactory class which will instantiate and return the appropriate handler based on the request URI.
HandlerFactory.getHandler("/payment/transport")
HandlerFactory.getHandler("/payment/telecom")
HandlerFactory.getHandler("/transfer/national")
Based on heavy load, we might consider creating pool of pre-instantiated handlers, and let HandlerFactory return handler from pool, and thus we use **Object pool** design pattern.

Sometimes it's important to have only class, and that class should control concurrent access to shared resource, we use **Singleton** design here.
CounterBean, LoggingClass, WorkManager, ConfigurationClass, and in some cases Factory class.
Creating Singleton is as simple as below –

```
public class Singleton {

    private static final Singleton instance = new Singleton;

    private SingletonExample() {
    }

    public static Singleton getInstance() {
        return instance;
    }

}
```

Singleton can also be implemented using enum. An enum value is initialized only once, and code is –

```
public enum SingletonEnum {
    INSTANCE;
 public void doSomething(){ //.. }
}

// Client Code -
SingletonEnum.INSTANCE.doSomething();
```

More Creational pattern: Abstract Factory, Builder, and prototype.

**Structural Design Patterns**

**Adapter** - Acts as bridge between two incompatible interfaces. Java class/program accepts input in one format and converts into second acceptable format for another program. Integration layer is Adapter design.

A **decorator** class wraps the original class and provides additional functionality keeping class methods signature intact.

**Facade** pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. Session Bean is a facade, hiding many details like persistence, transaction, entities, logging, etc and exposes several business operations to client.

Client wants to call Real Subject, but it calls **Proxy** as if real subject. Proxy handles the client calls and communicates with real subject, and returns to client.

Proxy might perform initialization of real subject, make remote connection to real subject, handle security, log, invoke method of Real Subject, and doSomething before returning to Client.

Other structural patterns include Bridge, Composite, Flyweight.

**Behavioral Design Patterns**

When current state defines the behavior, we use **State** design pattern. What's next depends on current state of object and may be on user input, for example, Mobile behavior changes based on theme set.

A request needs to be passed through several objects and appropriate object will handle the request. Java exception handling is an example of **Chain of Responsibility**.

**Iterator** is provided, If user needs to filter out some information from a collection.

When objects don't need to know whom they interact, or interaction is asynchronous, or a change in state of one object may affect several objects, we need a mediator, or may be a centralized control to pass information and manipulate participating objects. In **Mediator pattern**, we define an object that encapsulates how a set of object interact, like a message broker.

When we provide undo operation, we need to capture the internal state of object at some time and have ability to restore the object at later time. We need a **Momento** to store state of object, state can include any number of variables.

The **Observer** Design Pattern can be used whenever a subject has to be observed by one or more observers, like model-view-controller, where view is observer and model is observable and controller facilitates the interaction.

A **Template** method defines an algorithm in a base class using abstract operations that sub-classes override to provide concrete behavior.

## Functional Java 8

Java language is criticized for being verbose[xii].

In Java 8, write a function that adds 2 to an integer like this:

Function<Integer, Integer> fn = x -> x + 2;

(InfoQ, 2014, p. 4)

What was preventing Java from being functional: Classes the heart of Java, every type must have name, every value is either primitive or object, and functions are represented by the objects (methods of an object).

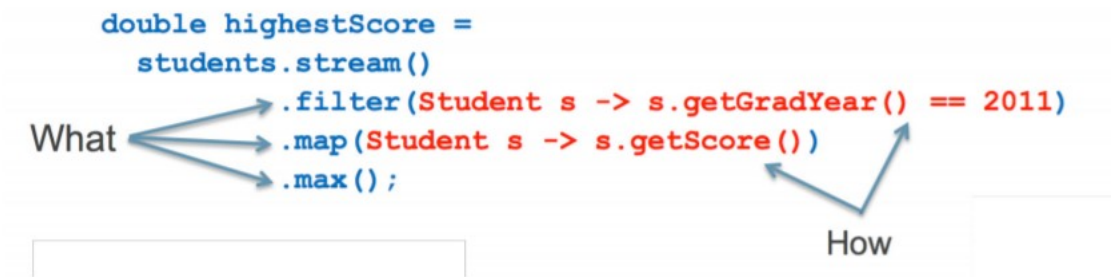
Functional interface, an interface with one method that represents abstract concept: function, action, or predicate. When referring, refer like *this function* rather that the function or method of object.

It's Java first step into functional programming. Below is the simple example of how more sophisticated code looks after using lambda.

Consider below two approaches:

```
List<Student> studentList = new ArrayList<>();

// Prepare a List
for (int i=1; i<11; i++) {
    Student s = new Student();
    s.setNumber(i);
    studentList.add(s);
}

// Print the maximum Number among all students without using Lambda
int maxNumber = 0;
for(Student s1: studentList) {
    if(s1.getNumber() > maxNumber)
        maxNumber = s1.getNumber();
}

System.out.println("Java - " + maxNumber);

maxNumber = 0;

// Print the maximum Number among all students using Lambda
Student s = studentList.stream().max((s1,s2) -> Integer.compare(s1.getNumber(), s2.getNumber())).get();
maxNumber = s.getNumber();
System.out.println("Java8 - " + maxNumber);
```

**Using normal for loop makes developer responsible for implementation and because for loop is inherently serial it is not optimized and not taking advantage of underneath – multi-core system. With lambda code is function and iteration, filtering, and accumulation can be done in library code freeing programmer to do it explicitly.** (InfoQ, 2014, p. 11)



Another example -

```
public interface Logger {
    public void log(Object o);
}
public class MDB {
    public void onMessage(Logger logger){ .....}
}
```

For using Logger inside MDB, implement Logger interface and pass it to onMessage method using Anonymous class (before Java 8), such as:

```
    public class MBD {
        public void onMessage(new Logger() {
                                public void log(Message m){
                                    System.out.print(m);
                                }
                            });
    }
```

With lambda:

```
    public class MBD {
        public void onMessage((m) -> System.out.print(m));
    }
```

**Functional Interface**

A functional interface is an interface with only one abstract method such as Callable, Runnable, ActionListener, and FileFilter. We can define a functional interface as below:

```
public interface ICompare {
      int compare (Student s1, Student s2);
}
```

FileFilte[xiii]r is another Functional interface in Java API, with only one abstract method: boolean accept (File path) and we can have a code like:

FileFilter x = f -> f.canRead();

Syntax tells compiler to create a FileFilter that filters files based on property if it can be read. Note, we didn't mention that f is a file instead compiler will infer it by the signature of FileFilter interface abstract method accept(File xxx). The code can even further be simplified using Java 8 **method references**:

FileFilter x = File::canRead;

If assume Person class that needs to be sorted by birthday:

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

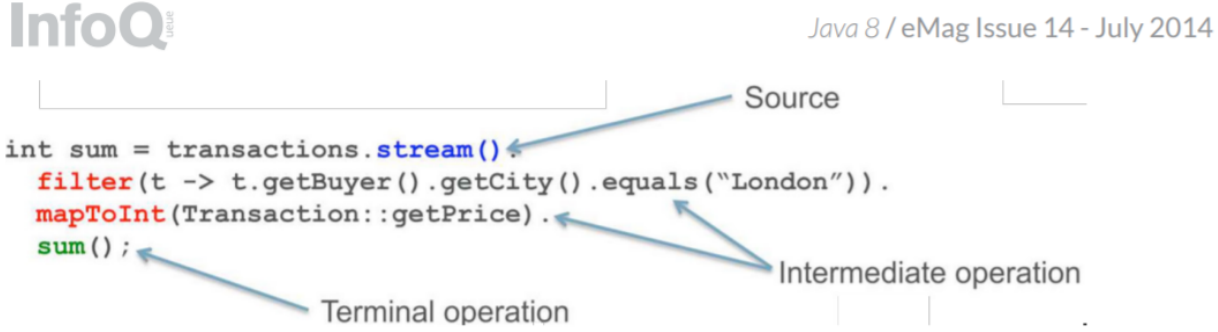
If Person class already has a method named compareByAge(), then it can be even more simpler using Lambdas:

```
Arrays.sort(rosterAsArray,
    (a, b) -> Person.compareByAge(a, b)
);
```

In the previous example, lambda expression called an existing method and for those cases, method reference enables writing easy-to-read lambda expressions for methods that already have a name, i.e. we can use method reference in lambda expression, like below

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

Java updated its existing libraries to support lambda, such as stream method (pipeline of data) in collections library, zero or more filters for filtered stream, and a terminal operation to return result




**Default method in Interface**

Now, we can add method definition in an interface, and method can be default or static and this leaves us to rethink when to use Abstract class. If we have designed an interface and we have clients extending the interface, it becomes difficult to later change the contract of interface, since all extending class will have to provide implementation for any new method added. A method with implementation can be added to an interface and won't require any client code to be changed. Extending client can use the default method or do nothing or can override the method.

One can add a static method to Interface.

Default & static enables us to add new functionality to existing interfaces without breaking extending classes.

**Aggregate operations**

Aggregate operations enable functional style operations on streams obtained from collections; particularly bulk operations. A stream is obtained from collection and several operations like filter(), map(), collect(), min(), max(), count(), reduce() can be performed on stream.

Below examples taken from Oracle docs -

```
// Below code prints name of each element in collection
collection.stream().forEach(e -> System.out.println(e.getName());


// Below code prints name of elements with sex MALE.
collection.stream().filter(e -> e.getGender() == "MALE")
                   .forEach(e -> System.out.println(e.getName()));
```

A pipeline is sequence of aggregate operations, like in above example, and includes collection, stream, operations, and terminalOperations like forEach.

```
double avg = collection        // Collection
    .stream()                  // Stream
    .filter(p -> p.getGender() == "MALE") // operation
    .mapToInt(Person::getAge) // mapToInt returns a new stream
    .average() // A reduction operation,
               // returns objects of type OptionalDouble
    .getAsDouble();
```

## Java EE notion

Java programming language is Java SE, the core java programming language that includes data types, threading, security, database access, networking, and so on. Java EE platform is built on the top of Java SE as API and runtime environment for developing and running large scale, multi-tiered, scalable, and secure enterprise network application.[xiv]

Java EE API's are built as a framework. Java EE is a platform to build a multi-tiered and distributed application. The platform provides API's to build all tiers of application[xv]:

- Client tier components that run on client machine such as web page
- Web tier components that manage the web pages, flows of web pages, and handle request & response from client tier
- Business tier component run on server and behind firewall. They have the business logic such as logic to transfer fund, retrieving customer details, and so on
- Enterprise Information System (EIS) running on EIS server such as database or file system

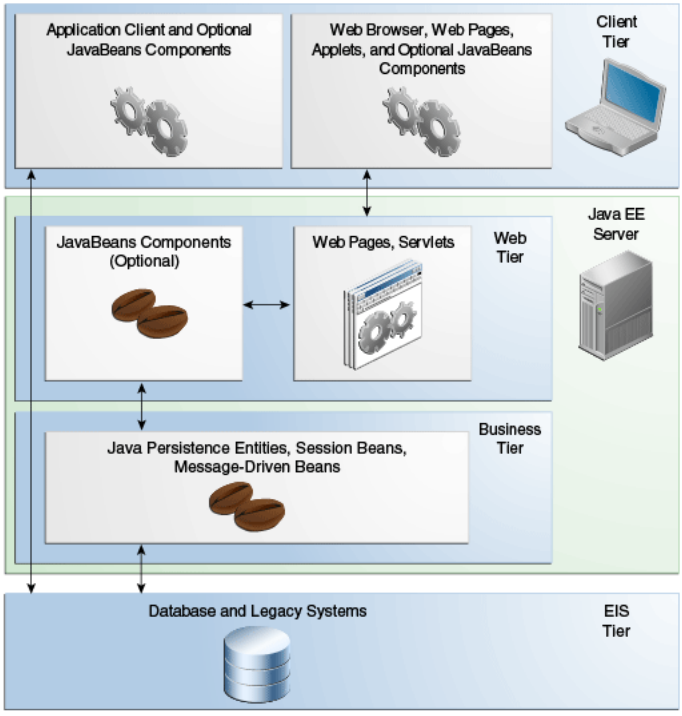



Image Source: docs.oracle.com/javaee/7/tutorial/

At a high level, few major components of Java EE are JSF, Servlet, EJB, and JMS. Though in recent days interviews and requirements are less concerned with understanding of these technologies but it is good fundamental knowledge for an architect. As many similarities can be found in JMS and latest messaging systems such as Rabbit MQ, Kafka, and so on. It is not recommended to go in details of JSF and EJB when they have very little market instead focus more on AngularJS and Spring Boot.

JSF is a framework for developing the web application – web pages and it has everything such as components, renderers, libraries, page flows, events, state, validation, conversion, navigation, and internationalization. The major drawback is pages are rendered on server side, it is tightly coupled with the backing bean, pages are xhtml, and not so adaptive & responsive. Before JSF, JSP was popular to build web pages. JSP, Servlets, and EJB combined formed the MVC2 architecture in which JSP is the view, Servlet is the controller, and EJB is the model. In most of the web applications, Model can be simple java bean. Servlets are the controllers and request from JSP comes to Servlet. Request can be filtered using filters, and events can be listened when servlet handles request or client session starts. Multiple servlets can exist in one web application but only one servlet context.

Several frameworks such as Struts, Spring Web Flow, Spring MVC, and JSF extend the basic MVC architecture of JSP, Servlet, and Java beans.

In Struts, there was only one controller servlet: *ActionServlet*. URLs were mapped to handlers(actions), and mapping was mentioned in struts-config.xml. *ActionServlet* routes requests to handlers (actions) and *ActionForm* (like JavaBean) is used to persist data between requests. All servlet objects: request, response, session, etc. are available to action class.

Spring used a central controller *DispatcherServlet* which received the request, dispatched task to *HandlerMapping* to select appropriate controller, and task of executing the business logic of controller to *HandlerAdaptor*. Controller executes business logic, sets Model, and returns view name. *DispatcherServlet* dispatches the tasks to resolve View based on view name to *ViewHandler*. View renders the model data and returns response.

*FacesServlet,* the central controller and engine of JSF framework.

## Servlet

Servlet is Java class that acts as a server for web clients. Servlet can respond to any kind of request but commonly used by web clients to send HTTP request and get response. Container (Application Server) manages the life cycle, requests, response, etc. of servlet. A servlet is defined with **@WebServlet** annotation on a pojo (java class) and it must extend **HttpServlet**. If name is not provided, the fully qualified class name is the servlet name.

```
@WebServlet("/account")}
public class AccountServlet extends HttpServlet  { .. }

@WebServlet(name="AccountServlet", urlPatterns={"/account", "/all"})
public class AccountServlet extends HttpServlet ... {
```

At least one url pattern must be declared.

Container on receiving first request for a servlet, will load the servlet class, create instance of servlet class, calls the *init* method of servlet class, and invokes the service method passing request and response objects. There onwards only service is called, but if servlet is to be destroyed then container calls destroy method of servlet class. That means we can override (optionally) *init* to perform any initialization and destroy to perform any cleanup. Service method is any method that can handle client request and return response, it is part of *GenericServlet* class, which is extended by *HttpServlet*, which is extended to write own Servlet such as *AccountServlet*. *HttpServlet* has one doXXX() method for each HTTP method: GET, POST, PUT, DELETE, and so on. Container (Server) will call the doXXX() via service method when client sends XXX request, that is to say doGet will be called when client send GET request. Applications don't override *service* method, instead override doXXX() method, and mostly doGet() or doPost(). By default, container will create only one instance of servlet per declaration. If application is marked distributed, container may have one instance per JVM. Container will instantiate multiple instances of servlet per JVM, if servlet implements *SingleThreadModel* (deprecated interface). Web container handles concurrent requests to the same servlet by concurrent execution of the service method on different threads, unless Servlet implements *SingleThreadModel* where container should guarantee only one request at a time to service method. It is recommended to use Atomic instance variables or synchronized blocks to avoid multithreading issues, but avoid *SingleThreadModel* and synchronized *doGet*() or *doPost*() methods. Any required initialization parameters are passed using *@WebInitParam*.

**ServletContext** is the interface (it has got methods that) servlet uses to communicate with container. One context per web application per JVM.

> "In the case of a web application marked "distributed" in its deployment descriptor, there will be one context instance for each virtual machine. In this situation, the context cannot be used as a location to share global information (because the information won't be truly global). Use an external resource like a database instead." - Oracle docs.

*ServeltContext* can be used to share information between servlets. Like an object can be added as attribute to context, which can be retrieved in another servlet. Servlets can share information like objects by maintaining them as attributes in one scope objects: context (*ServletContext*), session (*HttpSession*), request (*ServletRequest*), page (JSP page).

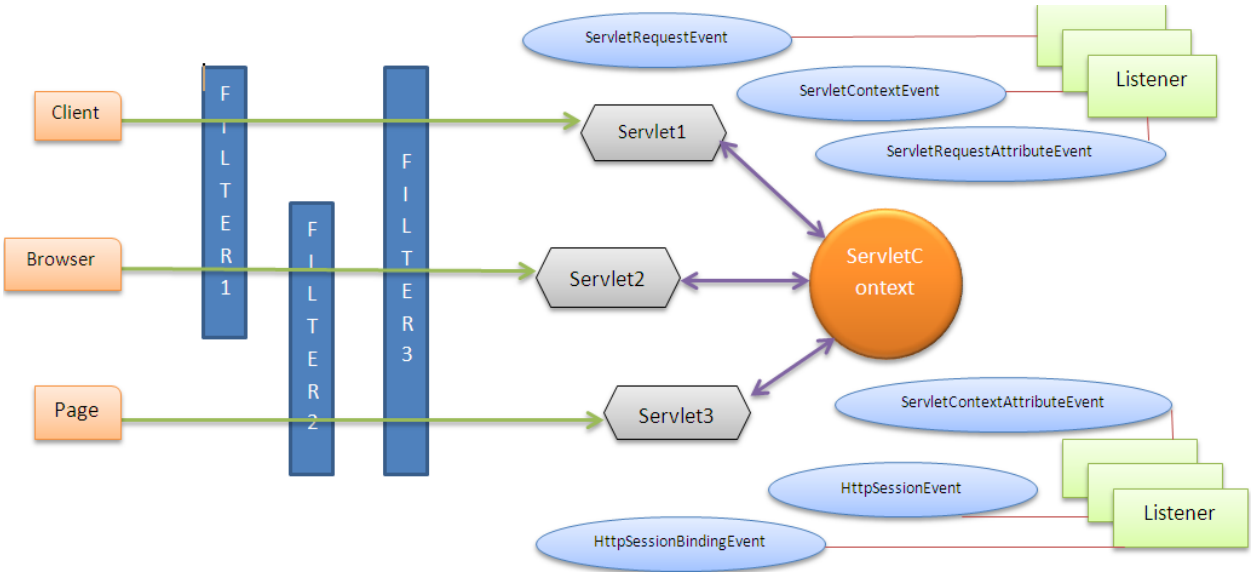
**Listeners** listen to Servlet lifecycle events by registering to servlet and react to events. Events such as context initialized/destroyed, attribute added/removed/replaced in context, session creation/invalidation/activation/passivation/timeout, attribute added/removed/replaced to session, request received, attribute added/removed/replaced to request.

```
@WebServlet(
    name = "TransferServlet",
    description = "This class handles fund transfer",
    urlPatterns = "/ft"
    initParams =
    {
       @WebInitParam(name = "minBal", value = "0"),
       @WebInitParam(name = "maxLimit", value = "50000")
    }
)
public class TransferServlet extends HttpServlet {
private AtomicInteger counter; // thread safe counter, counts number of transfer
public void doGet(HttpServletRequest req, HttpServletResponse res) {
if(!(req.getParameter("amount") > Integer.parseInt(getInitParameter("maxLimit"))) {
 ....
}
}
```

A **filter** class can modify the header and/or content of request and response. Multiple filters can be chained (by calling FilterChain.doFilter()), means filter will be called in sequence before actually request reaches servlet or response reaches client. Filter can add/modify/remove attribute to request, filter can change response, filter can perform logging like session log, filter can even block the request from reaching servlet (don't call doFilter()).

```
@WebFilter(
    urlPatterns = "/ft",
    initParams = @WebInitParam(name = "defaultCharge", value = "1"),
    description = "Filter to log and apply charges",
    dispatcherTypes = {DispatcherType.REQUEST, DispatcherType.FORWARD}
)
public class TransferFilter implements Filter {
}
```

default dispatcher is REQUEST (to be used when request comes directly from client)




**Asynchronous Servlet** was introduced in Servlet 3.0, and enhanced in Servlet 3.1 version as part of Java EE 7.

Idea is "Client sends request -> Container allocates thread (from pool) to handle request, and invokes servlet -> Servlet code calls *request.startAsync* and saves the *AsynContext* (like in list) -> Thread exits (or returns to pool) -> Connection is still open -> Some other thread uses saved AsyncContext and sends response (AsyncContext.getResponse()) and completes the process asyncContext.complete() -> Client receives the response".

So inside *doGet* or *doPost*, code is like below -

```
final AsyncContext asyncContext = request.startAsync(request, response);
asyncContext.setTimeout(10 * 60 * 1000);
list.add(asyncContext); // an option, otherwise an instance of Runnable (with AsyncContext) can be passed to a thread pool, so that another thread can handle sending response.
        // Above code will be called by thread one, handling request
        // Below code will be called by another thread sending response
asyncContext.getResponse().getWriter().println(htmlMessage); // optional
asyncContext.complete();
```

One of the new feature of Servlet 3.1 was support for Non-blocking IO (NIO). Servlet 3.0 only supported traditional IO. If requirement to read input stream from inside servlet, that is using *request.getInputStream()* and read byte by byte. If Input is blocked, then servlet is waiting.

Asynchronous servlet should not block while reading or writing; therefore, code with NIO will be

```
AsyncContext context = request.startAsync();
ServletInputStream input = request.getInputStream();
input.setReadListener(new MyReadListener(input, context));
```

where *MyReadListener* has callback methods -

*onDataAvailable* callback method is called whenever data can be read without blocking

*onAllDataRead* callback method is invoked data for the current request is completely read.

*onError* callback is invoked if there is an error processing the request.
Also Important is - The asynchronous behavior needs to be explicitly enabled on a servlet, i.e. add *asynSupported* attribute on *@WebServlet*, like below -

```
@WebSerlet(urlPatterns="/xyzAsync", asyncSupported=)
MyAsyncServlet {
//...
}
```

## EJB

EJB is the component to write business logic. The initial EJB's like EJB1 and EJB2 were overdesigned or over engineered that is why EJB got bad reputation of being heavy. EJB3 was major change in EJB after which many of the mandatory contracts were removed and annotations were introduced. From the beginning EJB's were of three types: Entity Bean, Session bean (Stateless and Stateful), and Message driven bean (MDB). Entity beans (replaced by JPA entities) and Stateful session beans were no longer used after EJB3. Stateless Session Bean (or can be called only session bean) and MDB are still used.
EJB's are of two types - Session and Message-driven, in which session bean suits for creating boundary that is session facade or service facade and Message-driven beans (MDB) acts as listener to JMS Queue or Topic. Message-driven beans are used frequently to handle requests in asynchronous manner, but note we can also have asynchronous session bean methods.
Same code can be placed in a Java class or in EJB, but in EJB some capabilities added to methods such as method can be exposed for remote lookup, exposed as Web Service, transaction capabilities are added to methods, loose coupling, and clear separation of other layers (from presentation).
App Server provides multiple options like pooling, caching so using EJB should not be performance issue.
When Server is first started, several Stateless session bean instances are created and placed in the ready pool. More instances might be created by the container as needed by the EJB container. When a bean instance is in the ready state, it can service client requests; that is, execute component methods. When a client invokes a business method, the EJB container assigns an available bean instance to execute the business method. When the EJB container decides to reduce the number of session bean instances in the ready pool, it makes the bean instance ready for garbage collection. Just prior to doing this, container will call the callback remove method. If client calls the remove method container will invalidate the bean instance from the ready pool. If all the beans in pool are active and any new client comes then it would be blocked and if transaction time outs or time out occurs, container would throw RemoteException to remote clients and EJBException to local clients.
EJB provides some options for setting the transaction through transaction attributes:
**REQUIRED** - Methods executed within a transaction. If client provides transaction, it is used; if not, new transaction generated. Commit at end of method. Default and well-suited for Session beans.
**MANDATORY** - Client of this EJB must create a transaction in which this method operates, otherwise an error. Well-suited for EJB entities.
**REQUIRES_NEW** - Methods executed within a transaction. If client provides transaction, it is suspended. A new transaction is generated, regardless. Commit at end of method.
**SUPPORTS** - Transactions optional.
**NOT_SUPPORTED** - Transactions not supported; if provided, ignored.
Transaction attributes help us create boundaries, but they don't solve the common problems faced, obsolete data or concurrent modification.
Some common problems related to entities in the architecture of Presentation, EJB, JPA are:
**Dirty Reads** - A transaction reads data written by another transaction that has not been committed yet. Because this data is uncommitted, a transaction failure would roll back these read changes. Occurs

when one transaction (T2) reads data that has been modified by previously started transaction (T1), but not committed. What happens if the T1 rolls back? T1 has incorrect data, thus "dirty read".
**Nonrepeatable reads** - A transaction rereads data it has previously read and finds that data has been modified by another committed transaction in the meantime. Occurs when one transaction (T1) reads same data twice, while another transaction (T2) modifies the data between the two reads by T1. T1 gets different value between the first and the second read, thus "nonrepeatable read".
**Phantom reads** - A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another committed transaction in the meantime. Occurs when one transaction begins reading data and another insert or delete data from the table being read.
Above mentioned problems are solved by setting the isolation level, which can be provided by App server or DB.

**Message Driven Beans**
A Message Driven Bean can also act as a facade. Client send a message to a JMS Queue, MDB's onMessage method is invoked and message is passed to MDB. onMessage can be in transaction, and if it fails, the message can be rolled back and sent back to Queue. Some precautions must be taken while using MDB -
1. The request, should be properly constructed, and multiple types of request possible (text, object, etc.)
2. MDB should be configured to not lose the message.
3. MDB supports transaction and 2-phase commit so transaction configuration can be done.
4. Exception handling – by default actual exception won't be propagated automatically to client.
5. Response and acknowledgment to client – by default no automatic response send.

Message Driven Bean(MDB) is a stateless, transaction aware J2EE component for consuming messages asynchronously. MDB's are configured to listen a queue, pick messages, and handle them. If an exception is generated, then by default it won't be handled, and thrown to container. Optionally, the MDB can consume the exception and take necessary action especially if the exception is a Business Exception. A simple rule is all unchecked exceptions (run time exceptions, errors, and environment errors) are System Exception and checked exception are Application exception[xvi]. MDB or in general EJB should catch or handle the system exception instead client should receive EJBException (subclass of RuntimeException) and container might destroy the bean instance and redirect next requests from client to another instance from pool. In case of business logic failure, client must receive application exception such as BookException and CartException (custom exceptions specific to application).
Inside Message Driven Bean we can configure the retry mechanism that should be triggered in case of exceptions. So, when exception would be generated, there would be retry.

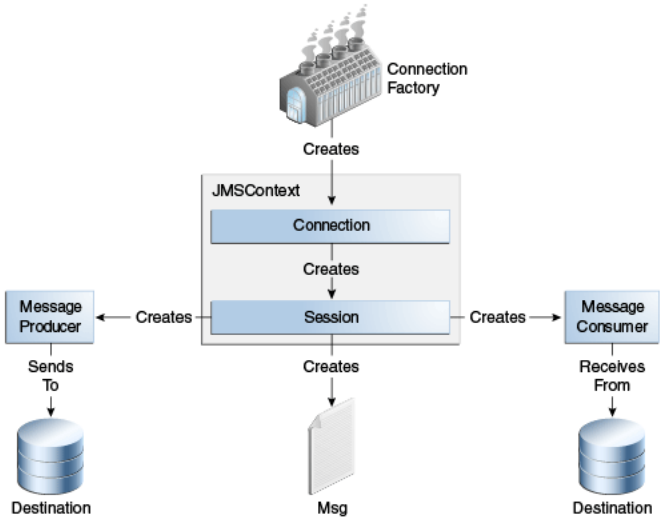
Retry can be configured using property "MaxDeliveryCnt" of ActivationConfig. We can also set the time interval between each retry, by setting property "endpointFailureRetryInterval". Any message, which has reached the max. retry count, is considered as "Poison Message". Poison message would be discarded by the JMS provider. But handling of poison message would depend on JMS provider. What we can do is configure a Exception Queue, where all the poison message would end. For using the

Exception Queue, we need to set some properties like: we set “UseExceptionQueue” as true, And set the Queue Name in “ExceptionQueueName”, And optionally we can set the property “IncludeBodiesInExceptionQueue” as true, If we want to include the message body, when it reaches the Error Queue.

```
@MessageDriven(activationConfig =
               { @ActivationConfigProperty(propertyName = "ConnectionFactoryJndiName",
                                           propertyValue = "MyQCF")
              ,
              @ActivationConfigProperty(propertyName = "DestinationName",
                                        propertyValue = "MyQ")
              ,
              @ActivationConfigProperty(propertyName = "DestinationType",
                                        propertyValue = "javax.jms.Queue")
              ,
              @ActivationConfigProperty(propertyName = "UseExceptionQueue",
                                        propertyValue = "true")
              ,
              @ActivationConfigProperty(propertyName = "ExceptionQueueName",
                                        propertyValue = "MyErrQ")
              ,
              @ActivationConfigProperty(propertyName = "IncludeBodiesInExceptionQueue",
                                        propertyValue = "true")
              ,
              @ActivationConfigProperty(propertyName = "endpointFailureRetryInterval",
                                       propertyValue = "5")
              ,
              @ActivationConfigProperty(propertyName = "MaxDeliveryCnt",
                                       propertyValue = "3")
              })
```

## JMS

JMS 2.0 is part of the Java EE 7 platform and was released in April 2013. JMS 2.0 is the first update to the JMS specification since version 1.1 was released in 2002. JMS 2.0 introduces a new API for sending and receiving messages that reduces the amount of code which a developer must write. For applications that run in a Java EE application server, the new API also supports resource injection. This feature allows the application server to take care of creating and managing JMS objects, thereby simplifying the application even further. You can use JMS in Java EE web or Enterprise Java Beans (EJB) applications or you can use it alone in a Java Platform, Standard Edition (Java SE) environment. The new API introduced in JMS 2.0 is known as the simplified API. As the name suggests, it is simpler and easier to use than the JMS 1.1 API. The simplified API consists of three new interfaces: JMSContext (replaces the separate Connection and Session objects in the classic API with a single object), JMSProducer (is a lightweight replacement for the MessageProducer object in the classic API. It allows message delivery options, headers, and other properties), and JMSConsumer (replaces the MessageConsumer object in the classic API and is used in a similar way)[xvii].




Source: https://docs.oracle.com/javaee/7/tutorial/jms-concepts003.htm

**Prevent message loss in JMS**

**Persistent Message**: A message can be persistent or non-persistent. Acc. to JMS specification, when a message is marked as persistent, the JMS provider must "take extra care to insure the message is not lost in transit due to a JMS provider failure". If the message is persistent, message is stored in the data structure representing the Queue (it can be table in database or file, etc.), otherwise message is stored in memory (in case of non-persistent). When message is sent over Queue and a consumer is connected the message will be dispatched to it. If there are no consumers connected the message will remain saved on disk until a consumer connects, where upon it will be dispatched. If the Queue Broker or Queue Manager is restarted, the non-persistent message would be lost.

**Durable Subscription**: A subscription on a topic can either be durable or non-durable. The term durable applies to the subscription, not the topic itself or the messages sent to it. And it applies only to subscriptions on topics, not subscriptions on queues. If the subscription is open(active), both non-durable and durable behave in same way. Diff. comes when the subscription is closed. Only durable Client would receive missed messages, when subscription is again open/active. In case of topics, there can be different combinations of persistent/non-persistent messages and durable/non-durable subscription. If subscription is durable, then message would be saved if the subscriber is inactive, otherwise not. If message is persistent, it would be saved on disk otherwise in memory. So, same message would be saved for durable subscribers, and won't be saved for non-durable subscriber. In case of topics there can be diff scenarios:

| | Durable Subscription | Non Durable Subscription |
|---|---|---|
| Persistent Message | Message would be saved, if subscriber is inactive.<br>Message would be saved on Disk.<br>Message is resend when subscriber becomes active again. | If the subscriber is inactive, message won't be saved for subscriber.<br>Subscriber won't receive the missed messages after it becomes active again. |
| Non Persistent Message | Message would be saved, if subscriber is inactive.<br>Message would be saved in Memory<br>Message is resend when subscriber becomes active again.<br>But if the Queue Broker or Manager is restarted before the Subscriber becomes active, then message would be lost, and won't be resend. | If the subscriber is inactive, message won't be saved for subscriber.<br>Subscriber won't receive the missed messages after it becomes active again. |

**Multi Consumer Queue** Not to be confused with Topic. Multiple consumers register with a Queue and Message Broker routes the incoming messages to different consumers in a load balanced manner: where one batch of messages is sent to One consumer, and next batch is sent to next consumer in line,

and similarly batch of messages is sent to different consumer, but same message is not sent to multiple consumers. Messages are routed to active consumers in order they registered with the broker. Idea behind is the message production rate is much higher, and there is a need of load balancing on the message consumer side. When message production rate is slow, broker might dispatch messages in uneven manner, and some consumers may never receive any message. There should be a backup consumer, if the active consumer fails. And to avoid whole batch of messages.

Message is an entity, and once client delivers the message to Message Broker inside Provider, its role is finished. Message are passed to consumer using the store and forward paradigm. To avoid losing message in case of Provider failure OR consumer failure, message should be persistent message. Message can be in many states –

- It might be prepared by Client and not yet delivered to Broker.
- Message is routed to Broker by the Message Producer.
- Message is at Broker and waiting for Consumer to consume it.
- Message has been delivered from Broker to Consumer.

When we make message Persistent or when we make Durable Subscription, we guarantee that message won't be lost from Broker. But message can be in a different state also, OR message is carrying some data or information, which should not be lost. If message is delivered from Broker, it must reach Consumer, it should not be lost in network, and if in case it is lost, there should be re-delivery.

**Guaranteed Delivery** – Persistent message is not lost by broker and durable subscribers definitely receive message. If message is lost in network, then redelivery should happen. Message Acknowledgment is key to guaranteed messaging and ensures that is message is not lost from network while travelling from producer to consumer, it would be re-delivered.

**Message Acknowledgment** - There are several types of Acknowledgement, and acknowledgement is set on session. Message Acknowledgment has different meaning for Broker and Consumer. Acknowledgment means consumer acknowledges that message is received. Acknowledgement is sent from broker, and also from consumer to broker.

**Brokers Acknowledgement** (Broker to producer) - Broker received message, stored in memory or data store (in case of persistent message), and accepts responsibility of delivering it to consumer.

**Consumer Acknowledgement** (Consumer to broker) – Acknowledgment acc to Acknowledgement Mode[xviii].

- AUTO: the session automatically acknowledges the client's receipt of message. The completion of onMessage method in listener will trigger the acknowledgment. If acknowledgement sending fails, the message will be re-deliver and application should be prepared to handle the duplicate message.
- DUPS_OK: acknowledgment is sent after consuming particular number of messages (lazy acknowledgement).

- CLIENT: Application explicitly invokes the acknowledge method of message to send acknowledgment of successfully receiving message.

Message can also be received in TRANSACTIONAL_MODE, where message is received in JTA transaction, and acknowledgement is sent only if transaction commits. If broker doesn't receive an acknowledgement, it would attempt to redeliver the message, with setting a flag JMSRedelivered on message.

## Microservices

Microservices is must be to be known by developers these days and microservices are complex. At first it seems simple exposing a REST URL but its becomes complicated when we design the complete system using microservices or transform a monolith into microservices. As number of microservices grow and architecture becomes more and more distributed, managing the system involves lot of feedback loops and boundaries. When starting with microservices the big question in mind is boundaries of services, how big or small the service will be. This is where Domain Driven Design is helpful in which boundaries are decided by domains (sub-domains) or business capabilities. Don't create microservices on resource level except in some cases. Important is Microservices are not replacement of SOA and they both work together[xix]

**What is Microservice?** Small in size, Messaging enabled, bounded by contexts, autonomously developed, independently deployable, and Decentralized. Microservices can't be achieved by focusing on particular set of patterns, process, or tools; Instead stay focused on ultimate goal – a system that can make change easier. Focus is on two key aspects Speed and Safety at scale (Nadareishvili, Mitra, McLarty, & Amundsen, 2016, p. 27).

**How microservices are scalable?** The Microservices Architecture pattern corresponds to the Y-axis scaling of the Scale Cube, which is a 3D model of scalability from the excellent book The Art of Scalability. Microservices allow functional decomposition – breaking monolith into smaller services. The other two scaling axes are X-axis scaling, which consists of running multiple identical copies of the application behind a load balancer, and Z-axis scaling (or data partitioning), where an attribute of the request (for example, the primary key of a row or identity of a customer) is used to route the request to a particular server (Richardson & Smith, 2016, p. 6).

**How big or small microservice?** While small services are preferable, remember goal is not small services instead agility. How big or small is not fixed but definitely it is not on resource level in my experience such as create 100 tables and 100 RESTful microservice with CRUD operations. Instead consider four options: Bounded Context (smallest business function or process inspired by business units within organisation), Business Capabilities (such as exchange rates), CRUD (User), and Service (Authentication, OTP, and so on)

**How microservices communicate?** Inter Process Communication via messaging (via queues or topics) and/or Remote Procedure Call (REST service call). Communication can be one-to-one (REST service or Queue) or one-to-many (pub/sub) and communication can be synchronous (REST Service) or asynchronous (Queues and Topics).

We can use a message broker to deliver event notifications from microservices in an asynchronous manner. That said, letting microservices directly interact with message brokers (such as RabbitMQ, etc.) is not a good idea. If two microservices are directly communicating via a message-queue channel, they are sharing a data space (the channel). The message-passing workflow we are most interested in, in the context of microservice architecture, is a simple publish/subscribe workflow: PubSubHubBub.

**What should be message Format?** JSON is most commonly used but trend is different now. Netflix relies on message formats like Avro, Protobuf, and Thrift over TCP/IP for communicating internally and JSON over HTTP for communicating to external consumers (e.g., mobile phones, browsers, etc.).

**How Microservices share data?** Data sharing is no-no between micro services and ideally each microservice should manage its own repository – even Queue is a form off data sharing therefore topics are preferred. If approach is that every microservice will own its database, then That service is the canonical system of record for that data. Every other copy is a read-only, non-authoritative cache. How to manage data: Synchronous lookup (services call each other whenever required Customer service calls account service for account info), Asynchronous events and local cache by services (Customer service maintains a cache of Account details and send notification to Account service when Customer info is updated so that Account Service can update its cached Customer Info), Shared metadata library (read only, generally immutable data such as list of countries), Joins (Client application or another service - Aggregator - can aggregate data from several services).

**How transactions are managed in Microservices?** Model the transaction as a state machine of atomic events - As a workflow - Transaction Saga with Routing Slip

How microservices are independently deployable? In early days microservices can be deployed as multiple wars on same application server, later each microservice with embedded service. As infrastructure matures or organizations moved to AWS, several choices on AWS are:

- Each microservice deployed in independent EC2 instance (AMI)
- Each microservice packaged in container such as Docker and multiple containers across VM. ECS or Kubernetes like tool required to manage containers
- Serverless for example AWS lambda - To deploy a microservice, package code in a ZIP file, add metadata, and upload it to AWS Lambda. AWS Lambda automatically runs enough instances of microservice to handle requests.

**How microservices are discovered by client?** IP address & host of each service at runtime and how services know each other. Kubernetes, Docker Swarm, and Mesosphere help here. In AWS, ELB is server-side service discovery approach. The AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router. ELB is commonly used to load balance external traffic from the Internet, however, one can also use ELB to load balance traffic that is internal to a virtual private cloud (VPC). A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn't a separately visible service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself. Netflix has client-side discovery custom tool: Netflix OSS along with Netflix Eureka as service registry. Kubernetes and AWS don't have explicit Service registry instead they maintain internally.

**Why we need API Gateway?** API Gateway provide secure endpoints, orchestration and transformation, reduce latency, optimize payload (response from N+1 service calls to client)

*Though what questions will be asked can be guessed any preparation for interview can fail. Apart from technical questions, behavioral questions matter lot. Person interviewing is just looking at the technical knowledge but also looking the personality with whom he or she will work day in and out; therefore, dress well, communicate well, introduce yourself well, and be develop mutual respect during interviews.*

---

*Thanks, And Good Luck for Your Interview*

---

# Bibliography

Goetz, B. (2006). *Java Consurrency in Practice.* Addison-Wesley Professional.

InfoQ. (2014). Clarifying Lambdas in Java 8. *Info Q eMag*.

Lea, D. (2000). *A Java Fork/Join Framework.* Retrieved from http://gee.cs.oswego.edu/dl/papers/fj.pdf

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture Aligning Principles, Practices, and Culture.*

Oaks, S., & Wong, H. (2009). *Java Threads.* O'Reilly Media.

Richardson, C., & Smith, F. (2016). *Microservices from Design to Deployment.* NGINX Inc.

Tijms, A., Cheng, M., Demyers, A., Liang, C., Karkala, A., & Torres, T. (2017). *Java EE Security Essentials.* Retrieved from DZONE.

[i] https://en.wikipedia.org/wiki/Java_virtual_machine
[ii] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html
[iii] https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html
[iv] https://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html
[v] https://docs.oracle.com/javase/tutorial/java/generics/types.html
[vi] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html
[vii] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html
[viii] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html
[ix] https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#parallelSort-byte:A-
[x] http://www.oracle.com/technetwork/java/seccodeguide-139067.html
[xi] https://docs.oracle.com/cd/E13222_01/wls/docs90/secintro/realm_chap.html
[xii] https://www.quora.com/Why-is-Java-so-verbose
[xiii] https://docs.oracle.com/javase/8/docs/api/java/io/FileFilter.html
[xiv] https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html
[xv] https://docs.oracle.com/javaee/7/tutorial/overview003.htm#BNAAY
[xvi] https://docs.oracle.com/javaee/6/tutorial/doc/bnbpj.html
[xvii] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/JMS2.0WebMessage/JMS2.0WebMessage.html
[xviii] https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.dev.doc/q032230_.htm
[xix] https://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html