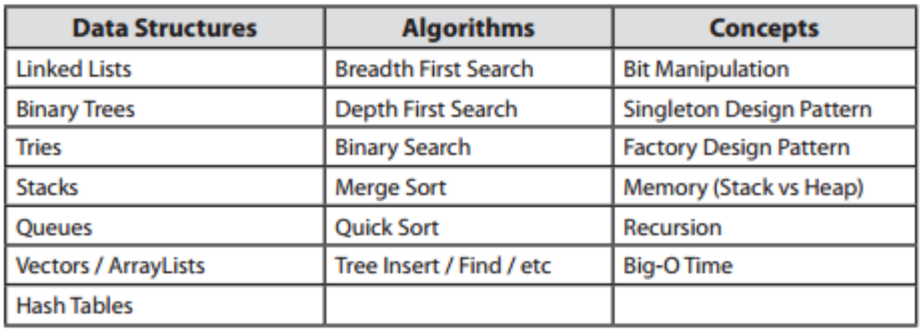
Java FAQs

1. 

Data Structures	Algorithms	Concepts
Linked Lists	Breadth First Search	Bit Manipulation
Binary Trees	Depth First Search	Singleton Design Pattern
Tries	Binary Search	Factory Design Pattern
Stacks	Merge Sort	Memory (Stack vs Heap)
Queues	Quick Sort	Recursion
Vectors / ArrayLists	Tree Insert / Find / etc	Big-O Time
Hash Tables		

2. **Java Pass-by-Value**: The key with pass by value is that the method will not receive the actual variable that is being passed – but just a copy of the value being stored inside the variable. Java will create a copy or copies of the values inside the original variable(s) and pass that to the method as arguments. Refer link 13.
   a. The difficult thing can be to understand that Java passes objects as references and those references are passed by value.
   b. E.g.

```
public void foo(Dog d) {
        d.getName().equals("Max"); // true
        d.setName("Fifi");
}
Dog aDog = new Dog("Max");
foo(aDog);
aDog.getName().equals("Fifi"); // true
```

3. **Marker (Tagging) Interfaces** - is used as a tag to inform a message to the java compiler so that it can add special behavior to the class implementing it. We cannot create custom marker interfaces. E.g. Serializable, Cloneable etc. http://javapapers.com/core-java/abstract-and-interface-core-java-2/what-is-a-java-marker-interface/
4. **Abstraction**
   Polymorphism
   Inheritance
   Encapsulation

   Pillars of OOP
5. **Polymorphism**: advice use of common interface instead of concrete implementation.
   a. Supported through method overloading (compile-time) and overriding (run-time).
   b. Where to use?
      i. Method Arguments: Always use super type then you may any of the sub-types.
      ii. Variable Names: Again always use super type along with Factory if required.
      iii. Return Type: `public TradingSystem getSystem(String name){…}`
6. **Encapsulation:**
   a. It is the process of hiding the implementation complexity of a specific class from the client.
   b. Defining private fields and providing getters and setters is not encapsulation.
   c. It's typically implemented through information hiding.
   d. With levels of accessibility (private, public etc.) we control the level of encapsulation.

   e. Hide implementation details from the client.
   f. E.g: int d = calculateDistance(x, y); // Here, calculateDistance encapsulates the calculation of the (euclidean) distance between two points in a plane: it hides implementation details. This is encapsulation, pure and simple.

7. **Abstraction**:
   a. Abstraction is where the design starts. It is a concept. It can be achieved through interface, abstract class, inheritance, access modifiers etc.
   b. Abstraction is the process of refining away all the unneeded/unimportant attributes of an object and keep only the characteristics best suitable for your domain.
   c. Data abstraction and encapsulation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details.
   d. **Abstraction** is more focused on the representation of an item whereas **encapsulation** hides details of the implementation. Encapsulation is a strategy used as part of abstraction.
   e. E.g.
      i. Onlooker: Sees a car on the road, have no idea about power steering etc.
      ii. Buyer: Gets to see more details of the car like power steering, brakes etc.
      iii. Mechanic: Gets to see the internal working of power steering, brakes etc.

   Level of abstraction is applied for each type of user.
8. <**Most Accessible**> Public -> Protected -> Package Protected (Default) -> Private
9. **Method Overriding is run-time** (dynamic binding) polymorphism. **Method Overloading is compile-time** (static binding) polymorphism.
10. **Overriding methods** in sub-class must have broader scope than overridden methods in super-class. In other words, when you override a method, you can make it less private, but not more private.
11. **Java Reflection**
    a. Ability to examine and/or modify the properties or behavior of an object at run-time.
    b. Primarily consists of:
       i. *Metadata*: like data about your Java classes, constructors, methods, fields, etc.
       ii. Functionality: to manipulate the metadata like constructors, methods, fields etc.
    c. Can slow down performance because it involves types that are resolved at run-time – and not at compile time.
    d. It exposes class internals for e.g. private fields.
12. **Inner classes**
    a. Unless the inner class is declared to be static, each instance of an inner class can be thought of as existing inside an instance of the outer class.
    b. Inner classes are non-static by default.
    c. Because the inner class instance exists inside an outer class instance, it can directly invoke the methods of outer class instance and it can also access the fields of the outer class instance.
    d. Have access to all private variables and methods of Outer Class.
13. **Inner Class vs. Nested Class**: Nested or inner classes can be static or non-static.
    a. Non-static classes are referred as inner classes.
    b. Static classes are referred as nested classes.
14. **Outer class** objects cannot directly access **inner class** methods. But this works:

    a. Animal.A object = new A(); {If inner class A is static}
    b. Animal.A object = new Animal().new A(); {If inner class A is non-static}

15. **Anonymous Class**
    a. An anonymous class is a class that has no name.
    b. Are created by extending an existing class or implementing an interface right at the point in the code where the class is being instantiated.
    c. E.g.

```
public Addition getAddition() {
        return new Addition() {
                ……
        };
}
```

16. **Synthetic:** Class, method, field and similar constructs introduced by compiler. It may not be visible in the source code. E.g. when an enclosing class accesses a private attribute of a nested class, compiler creates synthetic method for that attribute.
17. **Serialization**:
    a. Write an object into a stream, so that it can be transported through a network and that object can be rebuilt again.
    b. Need a protocol to rebuild the exact same object again. Java serialization API provides that.
    c. Respective class should implement the marker interface Serializable.
    d. Tag properties that should not be serialized as ***transient***.
18. Process vs. Thread:

Process	Thread
Processes are often seen as synonymous with programs or applications.	Threads are sometimes called lightweight processes. Threads exist within a process — every process has at least one.
Each process has its own memory space.	Threads share the process's resources, including memory and open files.
A process has a self-contained execution environment.	Threads are more efficient, but potentially problematic, communication.

19. **ThreadLocal**: Way of implementing thread-safe operation. http://javapapers.com/core-java/threadlocal/. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.
20. **Thread Locks** are made available on *per Object basis*, which is another reason wait and notify is declared in Object class rather than Thread class.
21. **Thread Monitors:** The object taken in the parentheses by the synchronized construct is called a monitor object. While a thread is executing a method of a monitor, it is said to *occupy* the monitor (mutual exclusion).
22. **Semaphores:** is a counter (integer) that allows a thread to get into a critical region if the value of the counter is greater than 0.
    a. If thread is allowed, the counter is decremented by one otherwise, the thread is waiting until it can go.

   b. When thread leaves the critical section, the counter is incremented by one to allow one more thread to pass the critical region.
   c. E.g. Semaphore sem = new Semaphore(100); // 100 is the counter initial/max value.
   d. Acquire on semaphore sem.acquire() gives permit to thread and reduces counter by 1.
   e. Release on semaphore sem.release() releases the permit and increases counter by 1. Release doesn't have to be called by the same thread as acquire.

23. **Mutex (Mutual Exclusion):** Mutex are semaphores with value of one. So only one thread can enter the critical region guarded by the semaphore.
24. **ThreadPools:** Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a ***Blocking Queue***, which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it.
25. **BlockingQueue:** a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full.
26. **Read only** or **final variables** or **immutable object** you don't need synchronization despite running multiple threads. String is inherently **thread-safe**.
27. **Immutable:** A class/object is immutable if its state cannot be changed. To make a class immutable:
    a. Remove all setters.
    b. Provide a constructor with parameters required to initialize its variables.
    c. Define all variables as final as well as the class.
    d. If one of the variables is an object like List, then in getList return a new copy or clone of the list.

```
E.g. public List<String> getList() {

        List tempList = new ArrayList();

        tempList.addAll(list);

        return tempList;

    }
```

28. **Local variables are also thread-safe** because each thread has their own copy.
29. **Object level lock vs. Class level lock**
    a. Object: Thread enters into an instance-synchronized java method
    b. Class: Thread enters into ***static*** synchronized java method.
30. **Thread doesn't need to acquire a lock** in sync. Method B if it's entering B from sync. Method A.
31. **Threads run()** method is called only after **start()** has been called on that thread. Calling start() twice on same thread will result in **IllegalThreadStateException**.
32. If you **call run() method directly** no new Thread is created and code inside run() will execute on current Thread. When program calls start() method a new Thread is created.
33. **Synchronization (Blocking mechanism)**
    a. **NullPointerException** if object used in java synchronized block is *null* e.g. synchronized (myInstance) will throws NullPointerException if myInstance is null.
    b. Can only be used to control access of shared object within **the same JVM**.
    c. Doesn't allow **concurrent read**.

d. It's **re-entrant** in nature. Going from one block to other doesn't require locking again if on same object.

e. Each instance has its synchronized methods synchronized on a different object: the owning instance.

f. For e.g. Counter counterA = new Counter();

```
Counter counterB = new Counter();
Thread  threadA = new CounterThread(counterA);
Thread  threadB = new CounterThread(counterB);
```

Notice how the two threads, threadA and threadB, no longer reference the same counter instance. Calling a synchronized method on counterA will not block a call on same method by counterB.

g. The thread waiting to acquire lock is ***BLOCKED*** until it gets one.

34. **Alternatives to Synchronization**:
    a. **Locks** – lock() and unlock() - we can now grab and release a lock whenever desired. **tryLock()** method acquires the lock only if it is available at the time of invocation (non-blocking, not waiting for the lock will be released).
    b. **(Non-blocking) AtomicBoolean, AtomicInteger, AtomicReference** etc.
    c. **Semaphores, Mutex (**Semaphore with value one.**).**
    d. **ReadWriteLock** - associated locks.
        i. Read lock is used only for read-only operations (supports *multiple* locks at the same time) and Write lock is for write operations (*exclusive* lock).
        ii. *Read lock can be acquired only if the Write lock is released.*
        iii. When we lock the write lock, it will wait for releasing of all read locks and then acquires.
35. **Volatile variables**:
    a. Do not cache value of this variable and always read it from main memory.
    b. Write to any volatile variable happens before any read into volatile variable.
    c. Ensures every thread would see up-to-date value of that variable.
    d. Communicate content of memory between threads.
    e. Unlike a synchronized block, it will never hold on to any lock or wait for any lock.
36. **Double-checked locking**
    a. Thread A notices that the value is not initialized, so it obtains the lock and begins to initialize the value.
    b. The code generated by the compiler is allowed to update the shared variable to point to a partially constructed object before A has finished performing the initialization.
    c. Thread B notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread B believes the value is already initialized, it does not acquire the lock. If B uses the object before all of the initialization done by A is seen by B the program will likely crash.
37. **wait(), notify() and notifyAll()** method of object class must have to be called inside synchronized method or synchronized block. It will throw IllegalMonitorStateException otherwise. To avoid possible race condition and possible deadlock.
38. **Why in Object Class?**
    a. More than synchronization utility they are communication mechanism between two threads in Java.

   b. Synchronized and wait notify are two different areas.
   c. Synchronization = Mutual Exclusion. Wait, Notify = Communication mechanism.
   d. Locks are made available on per Object basis.

39. **Wait() method** defined in Object class:
    a. Wait releases lock on object while waiting. This allows other threads to call wait/notify too on the same object/monitor.
    b. Waiting thread can be awaken by calling notify and notifyAll.
    c. Wait is called on Object and through synchronized context *only*.
    d. A thread that calls wait() on any object becomes inactive until another thread calls notify() on that object.
40. **Thread sleep():**
    a. Used to pause the execution. Always puts current thread on sleep.
    b. Thread.sleep() method is a static method.
    c. Sleep doesn't release lock while waiting unlike wait() which releases the lock.
41. **Thread yield():**
    a. Pause the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute.
    b. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution.
42. **Deadlocks**: Thread 1 locked resource1, and won't release it 'till it gets a lock on resource2. Thread 2 holds the lock on resource2, and won't release it 'till it gets resource1. We're at an impasse. Neither thread can run and the program freezes up.
43. **Garbage collection**
    a. It will only trigger if JVM thinks it needs a garbage collection.
    b. System.gc () and Runtime.gc () which is used to send request of Garbage collection to JVM.
    c. Does not guarantee that a program will NOT run out of memory.
    d. Before removing an object from memory GC thread invokes finalize () method of that object and gives an opportunity to perform any sort of cleanup required.
44. **Strong, Soft, Weak and Phantom References. (Strongest -> Weakest)**
    a. A **strong** reference is an ordinary Java reference.
    b. A **soft** reference is exactly like a weak reference, except that it is less eager to throw away the object to which it refers.
    c. A **weak** reference is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the gc ability to determine reachability for you, so you don't have to do it yourself.
    d. A **phantom** reference on its object is so tenuous that you can't even retrieve the object -- its get() method always returns null.
    e. E.g. *WeakReference<Widget> weakWidget = new WeakReference<Widget>(widget); then elsewhere in the code you can use weakWidget.**get()** to get the actual Widget object.* https://weblogs.java.net/blog/2006/05/04/understanding-weak-references
45. **equals() v/s ==** : == compares the reference of the variable whereas .equals() compares the values which is what you want.

46. **Default equals() behavior** – If we don't override the equals method, then it behaves same as == i.e. it returns true if both variables refer to the same object, if their references are one and the same.

47. **finalize() v/s finally v/s final**:
    a. **finalize()**: Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
    b. **Finally**: used in try/catch statement always executed, even if Exception or Error occurs.
    c. **Final**: denotes that something cannot be changed throughout the life of your program.

48. **Objects vs. Primitives**:
    a. Fields containing primitive types do not contain references, but actual numerical values like 7.
    b. double salary = 65000.00; //creates a space in memory and puts the number 65000.00 into this space.
    c. Animal obj = new Animal(); //The *obj* field doesn't hold an object; it's still just a reference. The object is somewhere else in memory.

49. **Autoboxing/Unboxing**: Java automatically converts a primitive type like <u>int</u> into corresponding wrapper class object <u>Integer</u>. Its opposite case is unboxing.

    E.g. ArrayList<Integer> intList = new ArrayList<Integer>();
    intList.add(2784); //autoboxing - primitive to object
    intList.get(0); // unboxing object to primitive

50. **Unboxing** will throw **NullPointerException** if we try to unbox a null object. E.g.

    Integer count;
    (if count < 0) {…} // will throw NPE.

51. **Strings are immutable**: the value stored in the String object cannot be changed. Refer FAQ 20.

    E.**g**.: String str = "Hello";
    str = str + "Guest"; // new String object is created internally!

52. **StringBuilder/StringBuffer:** These are mutable! Require less memory as compared to Strings (immutable). StringBuilder is asynchronized v/s StringBuffer that is synchronized.

53. **String Intern**: string interning is a
    a. Method of storing only one copy of each distinct string value, which must be immutable.
    b. Interned strings *can't be garbage collected*, so it's a potential for a memory leak.
    c. Lives in PermGen space, which is usually quite small; you may run into an OutOfMemoryError with plenty of free heap space!

54. **String Passwords**: String is immutable in java and stored in String pool. It's a security risk because anyone having access to memory dump can find the password as clear text.

55. **String Pool:** 'string pool' comes from the idea that all already defined string are stored in some 'pool' and before creating new String object compiler checks if such string is already defined**.** If so, compiler returns that string from pool instead of creating a new one. It's stored in PermGen heap.

    E.g.: String s = "a" + "bc"; // String Literal (assigned to pool in PermGen)
    String t = "ab" + "c"; // String Literal
    String u = new String("abc"); // Forces creation of new String object at new memory (in Heap).
    System.out.println(s == t); // returns **true** since s and t contain same value; the compiler returns the same string object (i.e. same hashcode) from string pool.
    System.out.println(s == u); // returns **false**. New operator forces new memory assignment in pool for u (different hashcode).

System.out.println(s.equals (u)); // returns **true**.

56. **String Literals:** represent string in double quotes like "techie"**.** String literals are created in *String pools*.
57. **String as new() vs. literal:** Using literals will always put it in pool in Perm area. With new(), a string object is created in Heap space. *Intern* () has to be called explicitly to put it in pool in Perm area.
58. **Substring Memory Leak:** Refer FAQ 19. It's fixed in Java 7.
59. **String Vs. Char[] passwords:** Strings are immutable in Java if you store password as plain text it will be available in memory until GC clears it. Access to memory dump can find the password in clear text. Char Array won't print contents of array instead its memory location get printed.
60. **Stack v/s Heap:**
    a. Stack values only exist within the scope of the function they are created in. Once it returns, they are discarded.
    b. Heap values however exist on the heap. They are created at some point in time, and destructed at another (either by GC or manually, depending on the language/runtime).
    c. Now Java only stores primitives on the stack. This keeps the stack small and helps keeping individual stack frames small, thus allowing more nested calls.
    d. Objects are created on the heap, and only references (which in turn are primitives) are passed around on the stack.
    e. So if you create an object, it is put on the heap, with all the variables that belong to it, so that it can persist after the function call returns.
61. **Heap v/s PermGen:** The heap stores all of the objects created by your Java program. GC runs periodically and frees memory if you stop using an object. Perm space is used to keep information for loaded classes and few other advanced features like String Pool (for highly optimized string equality testing), which usually get created by String.intern() methods. It's still not clear whether PermGen is part of Heap or separate. Although referred as part of it, physically it's not.
62. **Java Heap space is divided into three regions** or generation for sake of garbage collection called **New** Generation, Old or **tenured** Generation or **Perm** Space. Permanent generation is garbage collected during full GC in hotspot JVM.
63. **Class Loaders:**
    a. Java ClassLoader loads a java class file into java virtual machine.
    b. E.g. ClassLoader classLoader = MainClass.class.getClassLoader();
       Class aClass = classLoader.loadClass("com.MyClass");
    c. **ClassNotFoundException** Classloaders cause these exceptions.
    d. In Java, .class files generated by the Java compiler remain as-is until loaded into the JVM -- in other words, the loading process is performed by the JVM at runtime. Classes are loaded into the JVM on 'as needed' basis. And when a loaded class depends on another class, then that class is loaded as well.
    e. **Class Loader Types:**
        i. Bootstrap: Loads core classes from util, lang etc. packages. It's JVM dependent.
        ii. Extensions: Loads classes that are extensions to standard core java classes.
        iii. System: Classes that are available on the class path, are loaded by system loaders.
64. An **interface** can **extend** more than one parent interface.
    a. E.g. public interface Hockey extends Sports, Event

65. Class implementing **extending interface must implement methods of base interface**. E.g. If interface A extends interface B and say Class C implements A, then it's required for Class C to implement interface B methods.
66. **An interface is a type**: means that when an object of that class is created, it will also have the interface type.
67. **Static members** belong to the class instead of a specific instance. It means that only one instance of a static field exists even if you create a million instances of the class or you don't create any. It will be shared by all instances.
68. **Can't access non-static data from static context** b/c non-static variables are associated with a particular instance of object while Static isn't associated with any instance.
69. **Class Literals**: formed by taking a type name and appending ".class"; for example, String.class.
70. **Association:**
    a. Association is a relationship between two objects.
    b. One-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects.
    c. Aggregation is a special form of association.
    d. Composition is a special form of aggregation.
71. **Aggregation (HAS-A):** Less restrictive than composition.
    a. Known as "HAS-A" relationship.
    b. The containing object has a member object and the member object can survive or exist without the enclosing or containing class or can have a meaning after the lifetime of the enclosing object also.
    c. Example: Room has a table. The table can exist without the room
72. **Composition (IS-A or CONTAINS)**: More restrictive than aggregation. It's a special form of aggregation.
    a. IS-A or IS-A-PART-OF relationship.
    b. The member object is a part of the containing class and the member object cannot survive or exist outside the enclosing or containing class or doesn't have a meaning after the lifetime of the enclosing object.
    c. Example 1: Computer Science Department is a part of the College. Without college, department cannot exist.
    d. Example 2: A Person is a supervisor.
73. **Generalization (Inheritance)**
    a. The process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass.
    b. Shared characteristics can be attributes, associations, or methods.
    c. Generalization is a relationship between a Parent and its Derived class. It is nothing but inheritance.
74. **Composition (Delegation) v/s Inheritance**: Inheritance relationship makes it hard to change the interface of a superclass. In the composition approach, the subclass becomes the "front-end class," and the superclass becomes the "back-end class."
    a. It is easier to change the interface of a *back-end* class (composition) than a *superclass* (inheritance).

b. The front-end class must explicitly invoke a corresponding method in the back-end class from its own implementation of the method. This explicit call is sometimes called "forwarding" or "delegating" the method invocation to the back-end object.

c. In composition any changes to Back End (Base) class will cause a change in Front end class at least. However, it won’t affect the client using the front end class.

d. In inheritance any changes to base class can cause a ripple effect on sub classes and the clients.

e. E.g. I v/s C:-

```
class Apple extends Fruit {…}    v/s        class Apple {
                                             private Fruit fruit = new Fruit();
                                             public int peel() {
                                                     return fruit.peel();
                                             } }
```

75. **Interface v/s Abstraction**: An interface gives you freedom with regard to the base class; an abstract class gives you the freedom to add new methods later.
76. **Programming to an interface**: Allows clients to be decoupled from the implementation.   E.g. List list = new ArrayList(); can be changed to List list = new TreeList(); anytime in the code without making any changes since we are adhering to List interface and not implementation classes like ArrayList or TreeList.
77. **@Target:** Indicates the kinds of program element to which an annotation type.
78. **ElementType:** Constants are used with the {@link Target} meta-annotation type to specify where it is legal to use an annotation type. E.g. @Target(ElementType.ANNOTATION_TYPE), @Target(ElementType.METHOD) means annotation can be used on methods only, @Target(ElementType.FIELD) means on fields only etc.
79. **@Retention:** Indicates how long annotations with the annotated type are to be retained.
80. **RetentionPolicy:** Annotation retention policy. E.g. SOURCE, CLASS, RUNTIME, @Retention(RetentionPolicy.RUNTIME)
81. **HashMap vs. HashTable:** The HashMap class is roughly equivalent to HashTable, except that it is unsynchronized and permits nulls.
82. **HashMap** stores both key + value tuple in every node of linked list.
83. **Hashing** - using some function/algorithm to map object data to some representative integer value. This **hash code** (or simply hash) can be used to narrow down search when looking for an item in the map.
84. **HashMap Resizing and Rehashing –**

    a. If the size of the Map exceeds a given threshold defined by load-factor e.g. if load factor is .75 it will act to re-size the map once it filled 75%.

    b. It’s similar to ArrayList resizing itself if it touches a given threshold.

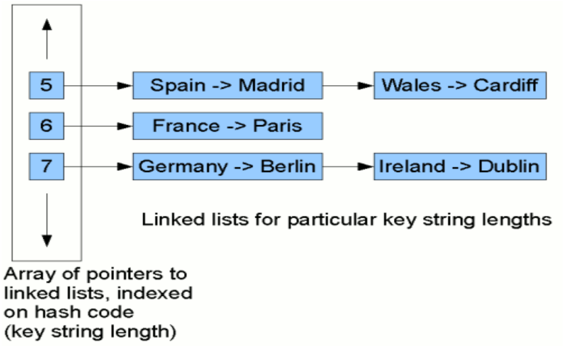
    c. Rehashing - creating a new bucket array of size twice of previous size and then start putting every old element into that new bucket array.

    d. Problems: there is a potential race condition while resizing HashMap. If two threads at the same time found that HashMap needs resizing and they both try to resize it.

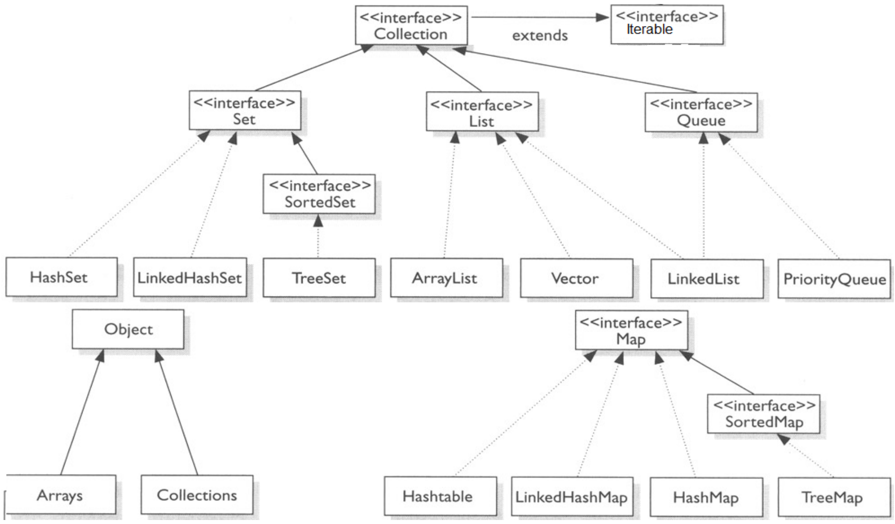
85. **Collision**: If multiple objects have same hash code then we have the problem of **Collision**.

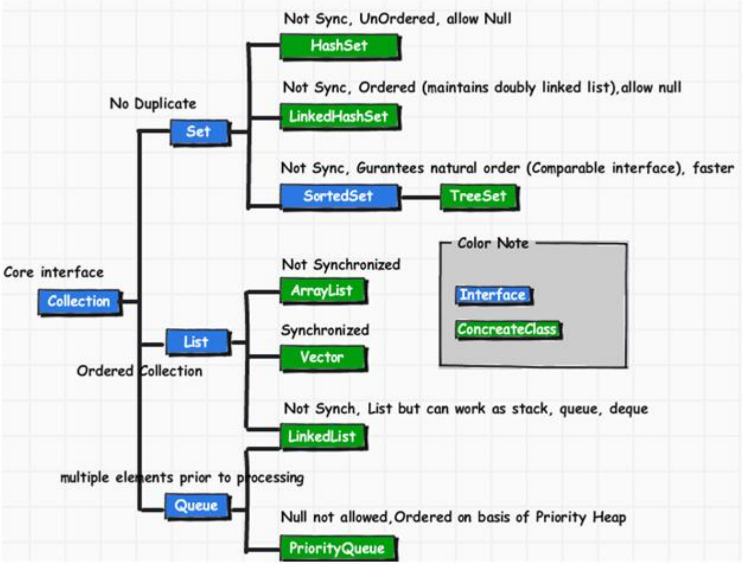
    a. Solution: array of *linked lists* rather than simply an array of keys(K)/values(V) called **Buckets (**strategy used in **HashMap)**.

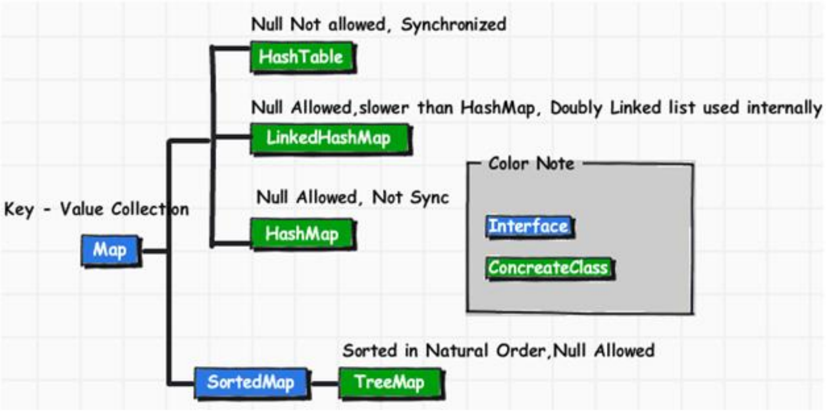
    b. To find the exact key K position in the linked list, scan elements sequentially and compare the keys using their equals() method. On a match we will return the value V.


5
6
7
Spain -> Madrid
Wales -> Cardiff
France -> Paris
Germany -> Berlin
Ireland -> Dublin
Linked lists for particular key string lengths
Array of pointers to linked lists, indexed on hash code (key string length)

86. **Custom Object as Key:** You have to override the ***equals and hashCode*** method.
87. **HashCode shouldn't change:** If your object is in a collection (as key) its hashCode shouldn't change else it won't be possible to fetch it again. To achieve this, the key object must be immutable.
88. **Collections Hierarchy:** A Collection is an interface that represents container for series of elements.



89. 
Not Sync, UnOrdered, allow Null
HashSet
Not Sync, Ordered (maintains doubly linked list),allow null
LinkedHashSet
No Duplicate
Set
Not Sync, Gurantees natural order (Comparable interface), faster
SortedSet
TreeSet
Color Note
Interface
ConcreateClass
Core interface
Collection
Not Synchronized
ArrayList
Synchronized
Vector
List
Ordered Collection
Not Synch, List but can work as stack, queue, deque
LinkedList
multiple elements prior to processing
Queue
Null not allowed,Ordered on basis of Priority Heap
PriorityQueue

90. 
Null Not allowed, Synchronized
HashTable
Null Allowed,slower than HashMap, Doubly Linked list used internally
LinkedHashMap
Color Note
Interface
ConcreateClass
Key - Value Collection
Map
Null Allowed, Not Sync
HashMap
Sorted in Natural Order,Null Allowed
SortedMap
TreeMap

91. **Collections don't allow primitives**. Containers want Objects and primitives don't derive from Object. If you add primitives to collections it will result in auto boxing/unboxing of primitive types.

92. **Set: Any Set implementation doesn't allow duplicates!**

93. **HashSet (**Unsync.**)**

    a. Implements the Set interface, backed by a hash table (actually a HashMap instance).

    b. It holds keys only.

    c. It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.

    d. This class permits the null element.

    e. It doesn't allow duplicate values.

    f. Amortized O(1) lookup (No collisions).

94. **TreeSet** (Unsync.) The elements are ordered using their natural ordering, or by a Comparator provided at set creation time. This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

95. **ConcurrentHashMap vs. HashTable vs. Collections.synchronizedMap()**

    a. **ConcurrentHashMap (**doesn't allow nulls**)** only *locks certain portion of Map* while **HashTable** *locks full map* while doing iteration. When you *read* from a ConcurrentHashMap using get(), there are *no locks.*

    b. HashMap which is non-synchronized by nature can be synchronized by applying a wrapper using **Collections.synchronizedMap**().

96. **ArrayList, HashMap *resize*** itself when the load factor reaches a certain threshold.

97. **LinkedHashMap** (best for LRU cache): It will iterate in order in which items were inserted.

98. **TreeMap** will iterate according to the natural ordering of keys according to compareTo method.

99. **Write operation on HashMap can trigger** resizing of HashMap and if re-size is done by multiple thread at same time that could lead to **race condition in HashMap**.

100. **Vector vs. ArrayList**: Vector is synchronized whereas ArrayList is not.

101. **Array** is *static structure*. It means that array is of fixed size. The memory which is allocated to array cannot be increased or decreased.

102. **For LinkedList<E>**

    i. get(int index) is O(n)

    ii. add(E element) is O(1)

    iii. add(int index, E element) is O(n)

iv. remove(int index) is O(n)
v. Iterator.remove() is O(1) <--- main benefit of LinkedList<E>
vi. ListIterator.add(E element) is O(1) <--- main benefit of LinkedList<E>

For ArrayList<E>

i. get(int index) is O(1) <--- main benefit of ArrayList<E>
ii. add(E element) is O(1) amortized, but O(n) worst-case since the array must be resized and copied
iii. add(int index, E element) is O(n - index) amortized, but O(n) worst-case (as above)
iv. remove(int index) is O(n - index) (i.e. removing last is O(1))
v. Iterator.remove() is O(n - index)
vi. ListIterator.add(E element) is O(n - index)

103. **Iterators:**
    a. When you implement Iterable (Contains method: Iterator iterator()) then those object gets support for for:each loop syntax.
    b. ConcurrentModificationException: We cannot add or remove elements to the underlying collection when we are using an iterator.
    c. Implement Iterator interface to implement your own *custom iterator*. Then implement Iterable interface to return custom iterator.
104. **Fail-fast iterators:**
    a. Fail as soon as they realize that structure of Collection has been changed since iteration has begun.
    b. Throws ConcurrentModificationException.
    c. This is not a guaranteed behavior instead it's done of "best effort basis".
    d. Iterators returned by most of collection are fail-fast including Vector, ArrayList, HashSet etc.
105. **Fail-safe iterators:**
    a. Doesn't throw any Exception if Collection is modified structurally.
    b. They work on clone of Collection instead of original collection.
    c. Iterator of CopyOnWriteArrayList, ConcurrentHashMap keyset is an example.
106. **Natural Ordering/Sorting** means the sort order which naturally applies on object e.g. lexical order for String, numeric order for Integer or Sorting employee by their ID etc.
107. **Comparable V/S Comparator:**
    a. A **comparable** object is capable of comparing itself (a.compareTo(b)) with another object. Natural sorting (object is in user control).
    b. A **comparator** object is capable of comparing two different objects (compare(a, b)). For customized sorting (object is not in users control).
108. **Static Methods** can be overloaded but not overridden. A static method belongs to the class and not objects. Static method can't refer to *this* and *super* in any way.
109. **An outer class** cannot be private or protected. However, an **inner class** can be private as well as protected.
110. **Outer classes** cannot be **static** however **inner classes** can be **static**. Both can be final & abstract though.
111. **Private Constructor:** Only native class can create the object. It can be used in the singleton design pattern. Prevent construction of objects. E.g. when a class contains only static members!

112. **Final modifier,** when applied to either a class or a method, **turns off late binding,** and thus **prevents polymorphism.**

113. **Dynamic/Late binding** means that the method implementation is determined at run-time.

114. **Copy Constructor:** constructor that takes only one parameter which is the same exact type as the class in which the copy constructor is defined. Newly created copy is independent of original object. The copy is located at a completely different address in memory than the original.
 E.g.: public Galaxy(Galaxy g) {}

115. **Constructors cannot** be **synchronized**: when the constructor is called, the object does not even exist yet.

116. An **abstract class** is also good if you want to be able to declare **non-public members**. In an **interface**, all methods **must be public**.

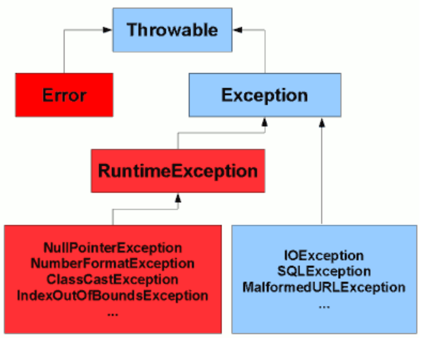
117. **Cloneable** is a *marker* interface. **CloneNotSupportedException** will be thrown on super.clone() if Cloneable is not implemented by the super class. You must override clone() method from Object class. Other ways:
    a. **Copy Constructor**
    b. **Serialization**
    c. **Apache Commons Utility**
    d. **Iterate through the getters for original and call setters for new cloned object.**

118. **Clone() is protected in Object class because**: the class implementing Cloneable and override the clone() method from Object. The overriding method can be less private i.e. in this case public only.

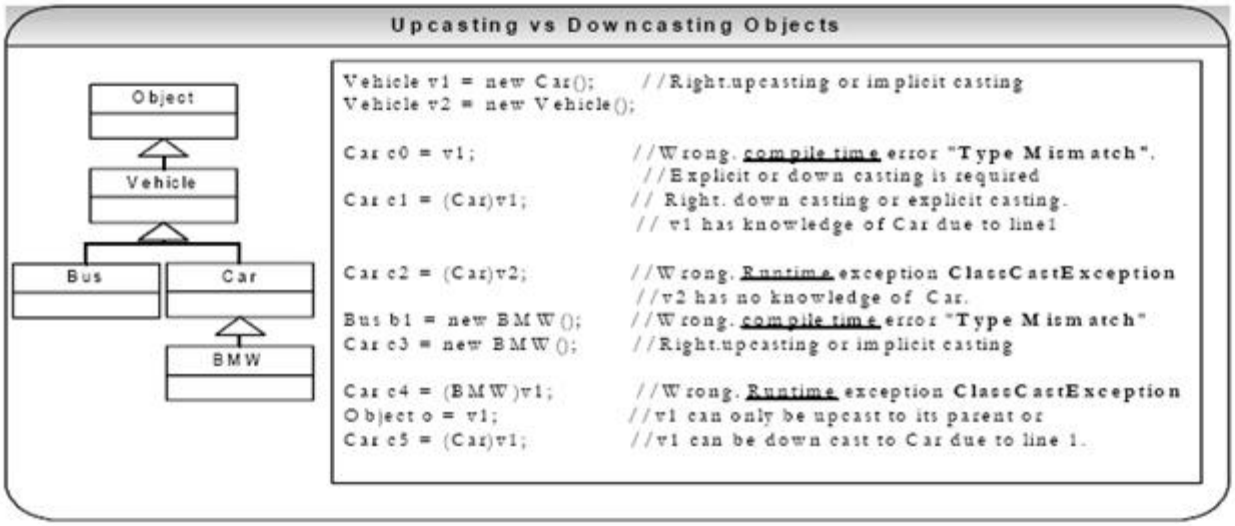
119. **Shallow v/s Deep Copy**:
    a. *Shallow*: Clone method of an object, creates a new instance of the same class and copies all the fields to the new instance and returns it.
    b. *Deep*: When the copied object contains some other object it references; are copied recursively in deep copy. Serialization can be used to achieve deep copy which also takes care of any cyclic dependencies.

120. **Checked/Unchecked Exception**: Checked Exception should be used if you know how to recover from Exception while Unchecked Exception should be used for programming errors.

Checked Exception	**Un-checked Exception**
Direct sub classes of Exception.	Direct sub classes of Runtime.
Checked at compile time.	Not checked at compile time
Method must either handle the exception or it must specify the exception using *throws* keyword.	NA
Invalid conditions in areas outside the program control. For e.g. invalid user input, network outage, absent files, database problems etc.	Represent defects in the program. E.g. Invalid arguments passed to non-private method.
FileNotFoundException, SQLException,	IllegalArgumentException, NullPointerException, IllegalStateException etc.



121. **Downcasting** is allowed and would throw RunTimeException if the code is executed.



Upcasting vs Downcasting Objects

```
Vehicle v1 = new Car();      //Right.upcasting or implicit casting
Vehicle v2 = new Vehicle();

Car c0 = v1;                 //Wrong. compile time error "Type Mismatch".
                              //Explicit or down casting is required
Car c1 = (Car)v1;            // Right. down casting or explicit casting.
                              // v1 has knowledge of Car due to line1

Car c2 = (Car)v2;            //Wrong. Runtime exception ClassCastException
                              //v2 has no knowledge of Car.
Bus b1 = new BMW();          //Wrong. compile time error "Type Mismatch"
Car c3 = new BMW();          //Right.upcasting or implicit casting

Car c4 = (BMW)v1;            //Wrong. Runtime exception ClassCastException
Object o = v1;               //v1 can only be upcast to its parent or
Car c5 = (Car)v1;            //v1 can be down cast to Car due to line 1.
```

122. **Generics:**

   a. Added to provide compile time type-safety of code and avoid ClassCastException at runtime.
   b. With Generics you don't need to cast object.
   c. Arrays don't support generics.
   d. It cannot be applied to primitives. E.g. ArrayList<int> //Illegal
   e. "?" denotes any unknown type, it can represent any Type. For e.g. Set<?> represents SetOfUnknownType and you can assign SetOfString or SetOfInteger to Set<?>.
   f. Set<Object> is setOfAnyType, it can store String, Integer but you cannot assign setOfString or setOfInteger to setOfObject. E.g.
      i. Set<Object> abc = new HashSet<String>; // Error
      ii. Set<?> abc = new HashSet<String>// Legal
   g. Set<? extends Number> will store either Number or sub type of Number like Integer, Float.
   h. T – used to denote type; E – used to denote element; K – key; V – value.

123. **Enums (Enumeration):-**

   a. Enum constants are implicitly static and final and cannot be changed once created
   b. Constructor of enum in java must be private else will result in compilation error.
   c. Enums are type-safe.
   d. Java compiler automatically generates static values() method for every enum in java.
   e. It's not possible to create instances of enums using new operator.

f. Ordinal() is used to fetch position of the enum constant in Enum.

124. **NIO (New IO) vs. IO:**

IO	NIO
Stream oriented	Buffer oriented
Read one or more bytes at a time from stream. Although, we can use BufferedReader to buffer the bytes from the stream.	Data is read directly into the buffer.
Streams are blocking.	It provides non-blocking mode.
When a thread invokes a read() or write(), that thread is blocked until there is some data to read, or the data is fully written.	Rather than remain blocked until data becomes available for reading, the thread can go on with something else.
Easy to read and parse data.	Parsing data is somewhat complicated since we have to keep track of buffer size.
Multi-threaded. Thread for each connection.	Single thread can handle multiple connections.

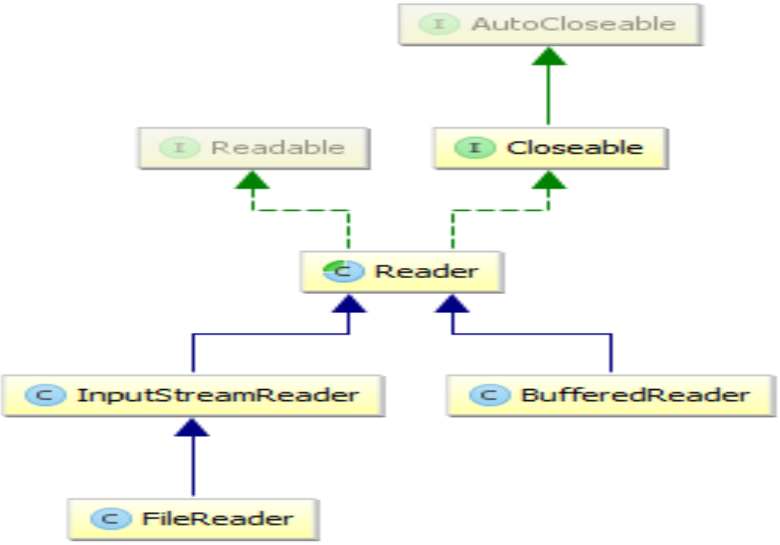
125. **Reading Files:**

a. FileReader vs. FileInputStream vs. InputStreamReader

i. FileReader class is a general tool to read in characters from a File (text files).

ii. FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream along with InputStreamReader.

iii. InputStreamReader helps read bytes. For e.g. image files.

iv. 

b. The BufferedReader class can wrap around Readers, like FileReader, InputStreamReader etc., to buffer the input and improve efficiency.

c. The efficiency difference with reading a character (or bytes) versus reading a large no. of characters in one go (or bytes).

Spring FAQs

1. **Java Servlet Lifecycle**: programs that run on a Web or Application server. Servlets are Java classes which service HTTP requests and implement the javax.servlet.Servlet interface.

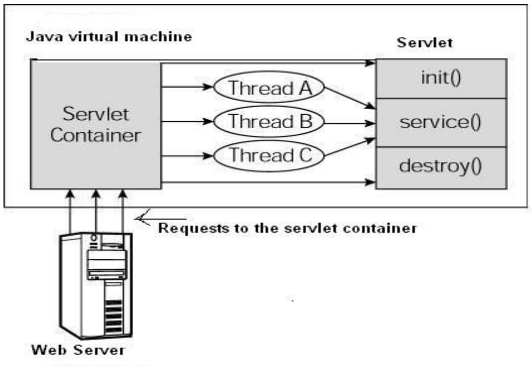


Figure: 1-thread/request. Requests are synchronous. Threads wait and sit idle until the request is processed.

   a. Init()
      i. Called only once when servlet is first created.
      ii. Usually called when user first invokes URL. Else we can specify it be loaded when the server is first started.
      iii. When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet, doPost etc.
   b. Service()
      i. Main method that performs actual task.
      ii. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client and to write the formatted response back to the client.
      iii. On every request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type and calls doGet, doPost, etc.
   c. Destroy()
      i. Called only once at the end of the life cycle of the servlet.
      ii. Let's servlet close db connections, halt background threads, cleanup etc.
      iii. After destroy(), the servlet object is marked for garbage collection.

2. **How servlets work?** *The Basics… It's by default synchronous and blocking I/O.*
   a. Web Server receives request from the Browser (Http Request)
   b. Web Server checks the request and invokes the appropriate Servlet/JSP in the Servlet Container
   c. If this is the first time the servlet is invoked its init() method gets called
   d. If not, the service() gets called.
   e. The service() method in turn delegates the processing to one of the doXXX() methods based on the request received
   f. The output of the doXXX() methods is sent back to the servlet container
   g. The container analyzes the response and formats it appropriately
   h. If the response is normal a Http Response is sent back to the client
   i. Else, an error response is sent back to the client.

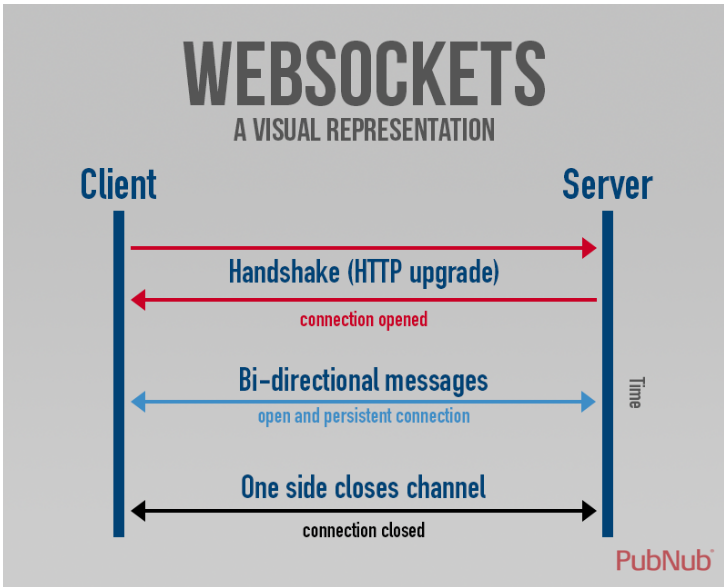
3. **Synchronous v/s Asynchronous**: By default (traditionally and before servlet 3.0 APIs) all the http requests are synchronous and blocking I/O. However, with Servlet 3.0 APIs we can make asynchronous http requests. Servlet 3.1 supports non-blocking I/O along with asynchronous calls. http://docs.oracle.com/javaee/7/tutorial/doc/servlets013.htm#BEIHICDH

RESTful

4. **RESTful Advantages/Characteristics:**
    a. The RESTful Web services are completely stateless. No client context being stored on the server between requests.
    b. Restful services provide a good caching infrastructure over HTTP GET method. It can help improve performance.
    c. Client-Server Separation: E.g. clients are not concerned with data storage. Servers are not concerned about UI.
    d. Layered architecture: A request from the client can be relayed from one server to another, improving scalability.
    e. It's light weight and simple to use. URLs are self-explanatory of the action to be performed.
5. **REST vs. SOAP**: REST is a concept that does not tie with any protocols.

SOAP	REST
Simple Object Access Protocol	REpresentational State Transfer
SOAP is a XML based messaging protocol	REST is not a protocol but an architectural style
SOAP has standard specification	REST has none
SOAP is XML based message protocol	REST does not enforces message format as XML or JSON or etc.
SOAP has specifications for stateful implementation as well	REST follows stateless model
SOAP is strongly typed, has strict specification for every part of implementation	REST gives the concept and less restrictive about the implementation.
Can be complex	Simple to understand
SOAP uses interfaces and named operations to expose business logic	REST uses (generally) URI and methods like (GET, PUT, POST, DELETE) to expose resources

6. **Idempotent**: GET, HEAD, and DELETE, PUT is idempotent, which means that if you call it more than once, it has the same result every time, whereas POST may keep doing what you ask it to do over and over again
7. **PUT** must create or update a specified resource by sending the full content of that same resource.
8. **PUT** puts a page at a specific URL. If there's already a page there, it's replaced into. If there's no page there, a new one is created.
9. **In SQL analogy**, *POST is an INSERT with an automatically generated primary key*, and *PUT is an INSERT that specifies the primary key in the INSERT statement.*
10. **PUT Example**: PUT /questions/<new_question/existing>. Used to create a resource, or overwrite it.
    POST to update: POST /questions/<existing_question>
    POST to create: POST /questions HTTP/1.1 {..request body..}

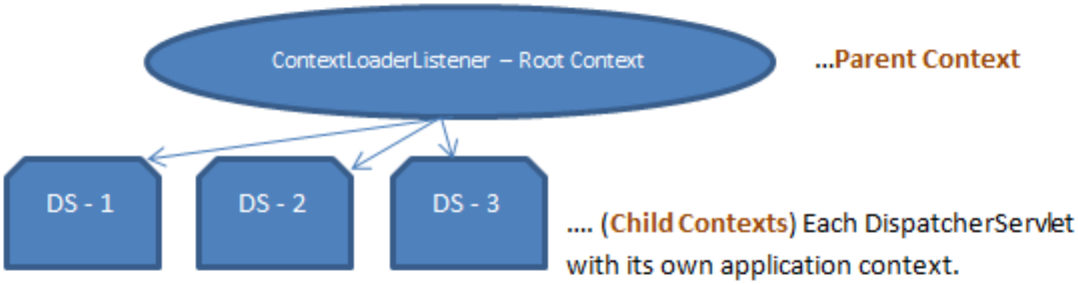
11. **Callbacks for RESTful APIs**: One option is to register callback URL. Other option is to have a persistent open connection between client and server. Third option is to use asynchronous http request (Servlets 3.1).
12. **WebSockets**:
    a. Runs over a persistent TCP connection.
    b. Through this open connection, bi-directional, full-duplex messages can be sent between the single TCP socket connection (simultaneously or back and forth).
    c. Through TCP connection, it can be used as a base for bi-directional real-time communication.
    d. Problems: Firewall, Network Topology, Security, Scalability etc.
    e. 

13. **Servlet Context:** One Servlet Context object will be created by the container for one web application. That means all servlets can have a scope to access that context object. E.g. let us assume there are 5 servlets in a web application, if u want to share same data with all servlets then we can share that data using servlet context, because every servlet can have interaction with that application environment.
14. **Servlet/Web Container:** is the component of a web server that interacts with Java servlets. A web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.
15. **Forward:**
    a. Forward is performed internally by the servlet.
    b. Browser is completely unaware that it has taken place, so its original URL remains intact.
    c. Browser reload of the resulting page will repeat the original request, with the original URL.
    d. SQL: for *SELECT* operations, use a forward.

    Redirect:
    a. It is a 2 step process, where the web application instructs the browser to fetch a 2nd URL, which differs from the original.
    b. Browser reload of the 2nd URL will not repeat the original request, but will rather fetch the second URL.
    c. It is marginally slower than a forward, since it requires two browser requests, not one objects placed in the original request scope are not available to the second request.

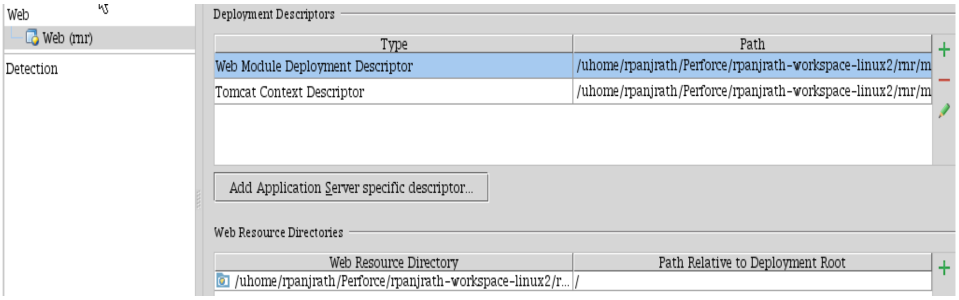
    d. SQL: for *INSERT, UPDATE, or DELETE* operations.

16. **Spring IoC container**: will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. It gets instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.
17. **BeanFactory v/s ApplicationContext**: These are two containers provided by Spring. ApplicationContext extends BeanFactory.
    a. BeanFactory provides basic IOC and DI features while ApplicationContext provides advanced features.
    b. BeanFactory doesn't provide support for internationalization.
    c. ApplicationContext can publish event to beans that are registered as listener.
18. **ApplicationContext** is an interface for providing configuration information to an application. By default spring implementations of ApplicationContext eagerly instantiate the entire singleton beans at startup.
19. **"root context"/ContextLoaderListener**


ContextLoaderListener – Root Context
...Parent Context
DS - 1
DS - 2
DS - 3
.... (Child Contexts) Each DispatcherServlet with its own application context.

    a. In terms of a web application (***WebApplicationContext***), means the main context that's loaded and used by the webapp. Typically, you'll start the root context with a ***ContextLoaderListener***.
    b. Other contexts are the child contexts which can be used to configure different set of classes/packages in application.
    c. The root context is optional. If you don't have the ContextLoaderListener defined, then you just don't have a root context. When you use a DispatcherServlet, it starts its own ApplicationContext, and it will get the beans it needs from there.
    d. Beans in a parent context are accessible throughout and by the child context, but the reverse isn't true.
    e. The *applicationContext.xml* defines the beans for the "root webapp context", i.e. the context associated with the webapp.
    f. The *spring-servlet.xml* (or whatever) defines the beans for one servlet's app context. There can be many of these in a webapp, one per spring servlet (e.g. spring1-servlet.xml for servlet spring1, spring2-servlet.xml for servlet spring2).
    g. Beans in spring-servlet.xml can reference beans in applicationContext.xml, but not vice versa.

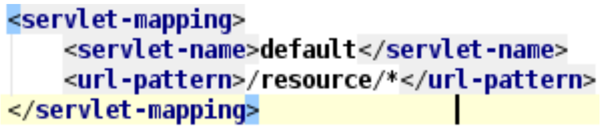
h. IntellijIDEA webapp folder not recognized properly: Please check web facet settings under Project Settings (ctrl+alt+shift+s)



Type	Path
Web Module Deployment Descriptor	/uhome/rpanjrath/Perforce/rpanjrath-workspace-linux2/mr/m
Tomcat Context Descriptor	/uhome/rpanjrath/Perforce/rpanjrath-workspace-linux2/mr/m

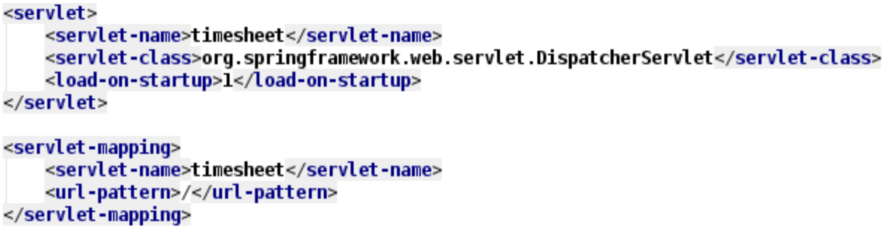
Web Resource Directory	Path Relative to Deployment Root
/uhome/rpanjrath/Perforce/rpanjrath-workspace-linux2/r...	/

20. DispatcherServlet:
    a. The DispatcherServlet first receives the request.
    b. It consults handler-mapping and invokes Controller associated with the request.
    c. The controller then processes the request by calling appropriate methods and returns the result to DispatcherServlet.
21. The @**Autowired** annotation is auto wire the bean by matching data type (*byType* is default).
22. The @**Qualifier** annotation us used to control which bean should be autowire on a field.
23. @**Autowired** (**required=false**) to disable dependency checking.
24. **web.xml**: The 'contextConfiguration' param with an empty value means that the Spring Context won't try to load a default file called <*servlet-name*>-servlet.xml.
25. **WEB-INF folder**: WEB-INF directory is a private area of the web application; any files under WEB-INF directory cannot be accessed directly from browser by specifying the URL and is accessible by the classes (e.g. servlets) within the application.
26. **Default servlet mapping**: Default servlet for all web applications that serves static resources.



```
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/resource/*</url-pattern>
</servlet-mapping>
```

27. **Static resource loader**: <mvc:resources mapping="/resource/**" location="/resource/static/"/>
28. **Controller Configuration**: By default, DispatcherServlet will look for *timesheet*-servlet.xml. Controllers, ViewResolver, Interceptors etc. must be defined in this {*servlet-name*}-servlet.xml file by convention.



```
<servlet>
    <servlet-name>timesheet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>timesheet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

29. **Multiple servlet mappings** can be defined in web.xml when required. Each servlet mapping will serve the url-patterns associated with them. The configuration for servlet mappings is not shared with other mappings in the application.
30. **JSP/HTML -> Controller mapping**: E.g. <*a href='employees'*>List employees</a> This will map to controller with RequestMapping "/employees"
31. **Autowire "byType"** (default) means, if data type of a bean is compatible with the data type of other bean property, auto wire it. If more than one exists, a fatal exception is thrown.

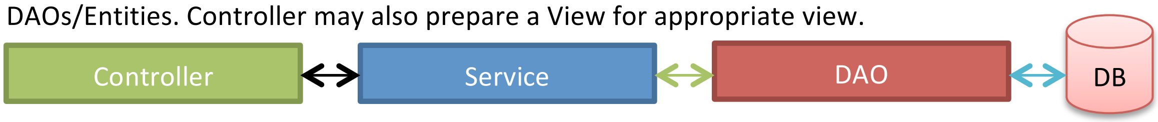
32. **Autowire "byName":** Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name and it contains a master property (i.e., it has a setMaster ()), Spring looks for a bean definition named master, and uses it to set the property.
33. Set the ***autowire-candidate*** *attribute* of the <bean/> element to false; the container makes that specific bean definition unavailable to the autowiring infrastructure
34. **Bean Scopes**:
    a. ***Singleton*** (Default - Only one shared instance of a singleton bean is managed),
    b. ***Prototype*** (creation of a new bean instance every time a request for that specific bean is made),
    c. ***Request*** (web aware spring ApplicationContext), ***Session*** and ***Global Session***.
    d. ***Custom Scopes.***
35. **Dependency Injection (DI):** Class A has a dependency to class B, if class A uses class B as a variable. If dependency injection is used then the class B is given to class A via: the constructor of the class A OR setter. It is called Inversion of Control. A class should not configure itself but should be configured from outside. http://javapapers.com/spring/dependency-injection-di-with-spring/
36. **Constructor Injection vs. Setter Injection:**
    a. Constructor-injection *enforces the order of initialization* and prevents circular dependencies (BeanCurrentlyInCreationException).
    b. With setter-injection it isn't clear in which order things need to be instantiated and when the wiring is done.
    c. Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor.
    d. It is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.
    e. Use of a @Required annotation on a setter can be used to make setters required dependencies.
    f. Constructor: large numbers of constructor arguments can get unwieldy, especially when properties are optional.
37. **DI Advantages:** Consider a service AuditService which requires AuditDAO object.
    a. **Sharing**: Since AuditDAO is injected here it's possible to share single AuditDAO (an expensive object) between multiple AuditService.
    b. **Loose coupling:** Since AuditServiceImpl is not creating instance of AuditDAO it's no more coupled with AuditDAO and work with any implementation of AuditDAO.
    c. **Testability:** Because AuditDAO is injected by DI at runtime it's easy to test audit() method by providing a mock AuditDAO class.
    d. **Flexibility:** Since AuditDAO is injected, we can always replace the bad performance DAO with an enhanced better performing DAO at any time.
38. **Bean Definition for Inner Class**: If you want to configure a bean definition for a static nested class, you have to use the binary name of the inner class. E.g. com.example.Outer$*Inner*
39. When defining a **bean** that you **create with a static factory method**, you use the class attribute to specify the *class* containing the static factory method and an attribute named *factory-method* to specify the name of the factory method itself.
40. **Bean *Aliases***: For using a bean name that is specific to that component itself. E.g.
    a. <bean id="name1" name="name2,name3,name4" class="java.lang.String"/>

    b. <alias name="name1" alias="namex1"/>

41. Specifying the target bean through the ***local*** attribute leverages the ability of the XML parser to validate XML id references within the same file. E.g. <ref local="someBean"/>
42. Specifying the target bean through the ***parent*** attribute creates a reference to a bean that is in a parent container of the current container. E.g. <ref parent="someBean"/>
43. The ***depends-on*** attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. E.g.
    a. <bean id="beanOne" class="ExampleBean" depends-on="manager"/>
44. A ***lazy-initialized*** bean tells the IOC container to create a bean instance when it is first requested, rather than at startup. E.g.
    a. <bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
    b. At the container level: <beans default-lazy-init="true">
       <! -- No beans will be pre-instantiated... -->
       </beans>
45. In Spring, **InitializingBean** and **DisposableBean** are two *marker interfaces*.
    a. For bean implemented InitializingBean, it will run afterPropertiesSet() after all bean properties have been set.
    b. For bean implemented DisposableBean, it will run destroy() after Spring container is released the bean.
46. **@PostConstruct** annotation is an alternate of initialization callback and **@PreDestroy** annotation as an alternate of destruction callback. Both are *not* related to spring and are part of JSR-250 annotation.
47. **Method-level request mappings** are examined in two separate stages:
    a. A controller is selected first by the DefaultAnnotationHandlerMapping and
    b. The actual method to invoke is narrowed down by AnnotationMethodHandlerAdapter.
48. **Consumable Media Types**: The request will be matched only if the Content-Type request header matches the specified media type. E.g.
    a. @RequestMapping(… consumes="application/json")
49. **Producible Media Types**: The request will be matched only if the Accept request header matches one of these values. E.g.
    a. @RequestMapping(… produces="application/json")
50. **RequestMapping method arguments:** @RequestParam, @RequestBody, @ResponseBody, @ModelAttribute, @PathVariable, ModelMap, Model etc.
51. **RequestMapping method return**: A String value that is interpreted as the logical view name.
52. **Request Narrow Down (params attribute):** You can narrow request matching through request parameter conditions such as params="myParam", params="!myParam", or params="myParam=myValue". This can help you narrow down between two similar requests like GET /v1/devices and GET /v1/devices?myParam1=myValue and GET /v1/device?myParam2=myValue etc.
53. **@ResponseBody**: Annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a view name).
54. **@ModelAttribute**
    a. ***Method Level***: purpose of that method is to add one or more model attributes. E.g. to fill a drop-down with pet types. All such methods are *invoked before @RequestMapping methods of the same controller*.

b. ***Method Argument:*** indicates the argument should be retrieved from the model. If not present, the argument should be instantiated first and then added to the model.

55. **@BindingResult:** http://stackoverflow.com/questions/10413886/what-is-the-use-of-bindingresult-interface-in-spring-mvc
56. The **HandlerMapping** strategy is used to map the HTTP client request to some handler controller and/or method.
57. **HandlerMapping** vs. **HandlerAdapter**: selecting objects that handle incoming requests v/s execution of objects that handle incoming requests respectively.
58. The **ViewResolver** provides a mapping between view names and actual views. The **View** interface addresses the preparation of the request and hands the request over to one of the view technologies.
59. **Flash attributes** provide a way for one request to store attributes intended for use in another. http://viralpatel.net/blogs/spring-mvc-flash-attribute-example/. RedirectAttributes can also be used as alternative.
60. <**mvc: annotation-driven**/> declares explicit support for annotation-driven MVC controllers (i.e. @RequestMapping, @Controller, although support for those is the default behavior), as well as adding support for declarative validation via **@Valid** and message body marshalling with @RequestBody/ResponseBody
61. In order to indicate that a specific class is supposed to be validated at the method level it needs to be annotated with **@Validated annotation** at type level.
62. **@EnableWebMvc** annotation is the replacement for <mvc: annotation-driven/>.
63. **@Component:** Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
64. **@Repository:** Specialization of @Component. Usually applied on DAO classes. Such a class is eligible for spring's DataAccessException.
65. **@Service**: Specialization of @Component. Users use service and don't own it.
66. **@Controller**: Specialization of @Component. It acts as a gate that directs the incoming information. It switches between going into model or view. All controllers of Spring MVC are ***singleton and stateless***.
67. **Service v/s Component**: A service is similar to a component in that it's used by foreign applications. The main difference is that component will be used locally (jar file, assembly etc.). A service will be used remotely through some remote interface.
68. **Controller v/s Service**: Controller is a part of Presentation layer and it calls services that interact with DAOs/Entities. Controller may also prepare a View for appropriate view.



69. **Service v/s DAOs**: Service is where all complex business logic is handled. Service may call DAOs for CRUD operations. DAOs handle all CRUD operations performed on Entities. Both are part of business layer.
70. **Security Checks**: All security checks like authentication, PreAuthorize etc. should be done at controller level. Service layer will handle business logic.
71. **The major building blocks of Spring Security that we've seen so far are:**

   i. **SecurityContextHolder -** to provide access to the SecurityContext.

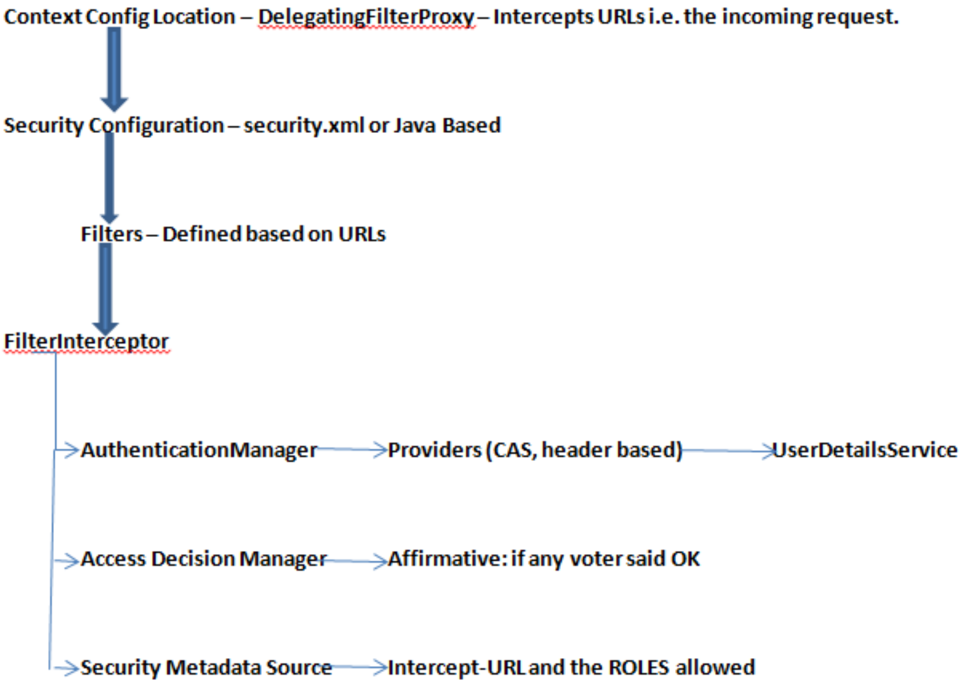
   ii. **SecurityContext -** to hold the Authentication and possibly request-specific security information. It's kept in thread-local storage.

   iii. **Authentication-** to represent the principal in a Spring Security-specific manner.

iv. **GrantedAuthority -** to reflect the application-wide permissions granted to a principal.

v. **UserDetails -** to provide the necessary information to build an Authentication object from your application's DAOs or other source of security data.

vi. **UserDetailsService -** to create a UserDetails when passed in a String-based username (or certificate ID or the like).


Context Config Location – DelegatingFilterProxy – Intercepts URLs i.e. the incoming request.
Security Configuration – security.xml or Java Based
Filters – Defined based on URLs
FilterInterceptor
AuthenticationManager
Providers (CAS, header based)
UserDetailsService
Access Decision Manager
Affirmative: if any voter said OK
Security Metadata Source
Intercept-URL and the ROLES allowed

72. **<u>Figure above shows how a typical URL is intercepted and security rules are applied.</u>**
73. **Principal:** can be used to represent any entity, such as an individual, a corporation, and a login id.
74. **A user is authenticated** when the SecurityContextHolder contains a fully populated Authentication object.
75. **Security @Pre/Post and @Secured:** secured-annotations="enabled" and pre-post-annotations="enabled" will enable PreAuthorize and its related annotations.
76. **"springSecurityFilterChain"** is an internal infrastructure bean created by the namespace to handle web security.

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

77. **Security Filter Chain:** We should define a *filterChainProxy* in web.xml using *DelegatingFilterChainProxy*. *filterChainProxy* should then be defined in application context and add filter chain beans to it. E.g.

```
<bean id="filterChainProxy"
class="org.springframework.security.web.FilterChainProxy">
  <constructor-arg>
    <list>
      <sec:filter-chain pattern="/restful/**" filters="
           securityContextPersistenceFilterWithASCFalse,
           basicAuthenticationFilter,
           exceptionTranslationFilter,
           filterSecurityInterceptor" />
      <sec:filter-chain pattern="/**" filters="
           securityContextPersistenceFilterWithASCTrue,
           formLoginFilter,
           exceptionTranslationFilter,
           filterSecurityInterceptor" />
    </list>
  </constructor-arg>
</bean>
```

78. **Bypassing the Filter Chain:** You can use the attribute *filters = "none"*
79. The **'springSecurityFilterChain'** filter needs to be configured to intercept all URLs so Spring Security can control access to them. The filter must be named this to match the default bean it retrieves from the Spring context.
80. **AccessDecisionManager**: Makes a final access control (authorization) decision.
    - i. *AffirmativeBased* decision manager will grant access if any voter said OK.
    - ii. *UnanimousBased* decision manager will grant access ONLY if all voters said OK.
    - iii. *ConsensusBased* decision manager will let the majority decide**.**
81. Spring Security has a **voter-based architecture.** The most common voter is the *RoleVoter*.
    - i. ***AuthenticatedVoter*** will search for strings IS_AUTHENTICATED_FULLY, IS_AUTHENTICATED_ANONYMOUSLY, IS_AUTHENTICATED_REMEMBERED.
    - ii. ***RoleVoter*** will check whether any string in “access” attribute starts with “ROLE_”
82. In Spring Security, the responsibility for storing the **SecurityContext** between requests falls to the **SecurityContextPersistenceFilter.**
83. **SecurityContextHolder** is where we store details of the present **security context** of the application. By default the SecurityContextHolder uses a ThreadLocal to store these details.
84. **Core Security Filters:**
    - a. **FilterSecurityInterceptor** - is responsible for handling the security of HTTP resources.
        - i. AuthenticationManager
        - ii. AccessDecisionManager
        - iii. securityMetadataSource
    - b. **ExceptionTranslationFilter** - handles exceptions thrown by the security interceptors and provides suitable and HTTP responses.
        - i. AuthenticationEntryPoint
        - ii. AccessDeniedHandler
    - c. **SecurityContextPersistenceFilter** - Populates security context across requests.
        - i. SecurityContextRepository
    - d. **UsernamePasswordAuthenticationFilter**

85. **SecurityContextPersistenceFilter** restores the context to the **SecurityContextHolder** for each request and, crucially, clears the SecurityContextHolder when the request completes
86. **AuthenticationManager** delegates the fetching of persistent user information to one or more **AuthenticationProviders**. One can specify multiple AuthenticationProviders, for e.g. *casAuthenticationProvider* and *requestHeaderAuthenticationProvider*.
87. **AuthenticationProvider** authenticates the user simply by comparing the password submitted in a **UsernamePasswordAuthenticationToken** against the one loaded by the UserDetailsService.
88. **AnonymousAuthenticationFilter**: Detects if there is no Authentication object in the SecurityContextHolder, and populates it with one if needed.
89. **Message Converters:** Defined in spring servlet xml config are used for mapping objects. For e.g. In REST APIs, JSON request can be mapped to POJO (& vice-versa) by defining MappingJsonConverter.
90. **JSON Getters/Setters:** Make sure to add getters and setters for the object you want to serialize else exception will be thrown… JsonMappingException: No serializer found for class
91. **Custom Exception Resolver:** (@link 7) we need to make spring's DispatcherServlet aware of our custom HandlerExceptionResolver. The only configuration we need is, for e.g.:
    <bean id="exceptionResolver" class="com.citrix.device.web.DeviceExceptionResolver"/>
92. **Order:** The order property is useful if you want to have the GenericExceptionHandler function in a chain if other HandlerExceptionResolvers need to be configured. This allows you to, for example, configure an AnnotationMethodHandlerExceptionResolver bean (e.g. order 0) so you can use the @ExceptionHandler annotation for custom exception handling strategies and then have the RestExceptionHandler (e.g. order 100) act as a 'catch all' default for all other Exceptions.
93. **@ExceptionHandler**-annotated methods are invoked when a @RequestMapping method on that same class throws an exception. Usually used on controllers.
94. Spring MVC 3.2 introduces the **@ControllerAdvice** component annotation that allows you define single **@ExceptionHandler** methods for all your controllers.
95. **@ExceptionHandler** catches the exception noted along with all the *sub-classes* of that exception.
96. **@ControllerAdvice** can also be a place to define global property editors and validators.
97. **Spring Custom PropertyEditors**: In order to convert the String data to your own datatype for e.g. in MVC @RequestParams DateTime dateTime, use property editors. Create your own property editor and register using **@InitBinder** in the controller.

    ```
    CustomDateTimeEditor extends PropertyEditorSupport
    @InitBinder
      private void dateBinder(WebDataBinder binder) {
        DateTimeFormatter formatter =  ISODateTimeFormat.dateTimeNoMillis().withZoneUTC();
        CustomDateTimeEditor editor = new CustomDateTimeEditor(formatter, false);
        binder.registerCustomEditor(DateTime.class, editor);
      }
    ```
98. **Custom Validators:** We may also register our own custom validators in similar way as we did for PropertyEditors.
99. **Converters v/s Formatters:** Converters are used to convert for e.g. String to DateTime whereas formatters are used to format for e.g. JODA DateTime in specific format (yyyyMMDD). E.g. Converter: *CustomDateTimeEditor* Formatter: *@DateTimeFormat.*

100. **Validators**: The Validator interface (supports & validate method) works using Errors object so that while validating, validators can report validation failures to the Errors object.

101. Spring 3 introduces **a core.convert** package that provides a general type conversion system. The system defines an **SPI** *to implement type conversion logic*, as well as an **API** to execute type conversions at runtime basically API is used to register your converters.

102. **Converter**: To create your own Converter, simply implement the interface below.

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> {
    T convert(S source);
}
```

Parameterize S as the type you are converting from, and T as the type you are converting to.

103. **ConverterFactory:** When you need to centralize the conversion logic for an entire class hierarchy, for example, when converting from String to java.lang.Enum objects, implement ConverterFactory.

```
package org.springframework.core.convert.converter;
public interface ConverterFactory<S, R> {
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

Parameterize S to be the type you are converting from and R to be the base type defining the range of classes you can convert to. Then implement getConverter(Class<T>), where T is a subclass of R.

104. **ConversionService API** must be *used to register* your own custom converter.

```
<bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="example.MyCustomConverter"/>
        </list>
    </property>
</bean>
```

105. **Formatter:** To create your own formatter just implement the interface:
public interface Formatter<T> extends Printer<T>, Parser<T> { }

106. **Annotation Driven Formatter:** Implement interface *AnnotationFormatterFactor.*

107. Spring provides **@NumberFormat and @DateTimeFormat** for numbers and JODA date-time.

108. **@Async – Asynchronous Task Execution**
   a. Used on a method for asynchronous execution.
   b. Caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring ***TaskExecutor***.
   c. Even methods that return a value can be invoked asynchronously using ***Future*** return type.
   d. From Spring 3.2 we can use qualifiers with @Async("exe1") to specify executor if there are multiple executors present in the context.
   e. We can enable @Async in two ways:
      i. XML Config:

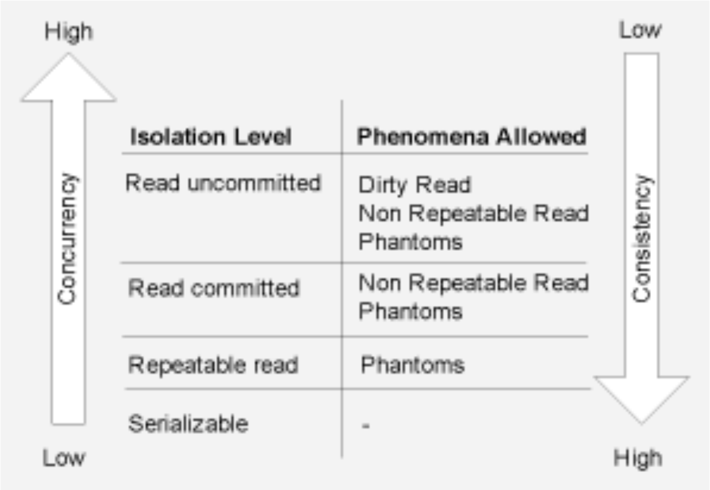
```
<task:annotation-driven executor="executorWithPoolSizeRange"
<task:executor id="executorWithPoolSizeRange" pool-size="5-25"
```

```
                    queue-capacity="100"/>
```

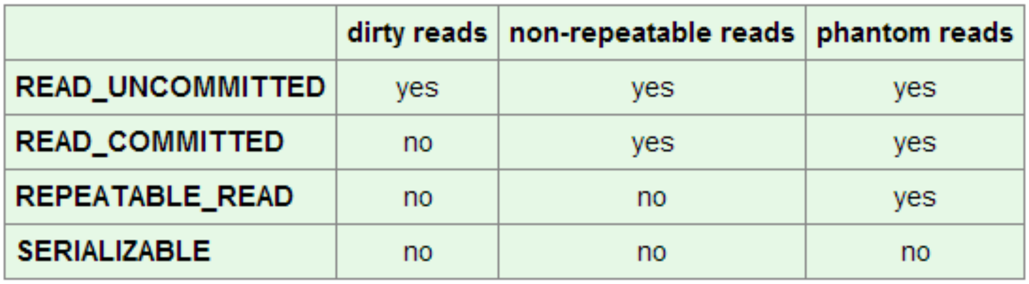
ii. Java Config: Add **@EnableAsync** to @Configuration class and implement AsyncConfigurer interface.

f. **Note:** If you invoke one method and then that method invokes the asynchronous annotated one, it will not work. The @Async annotation should be on top-level method preferably on interface defined method. This way spring would be able to intercept the annotation.

Database/Hibernate FAQs

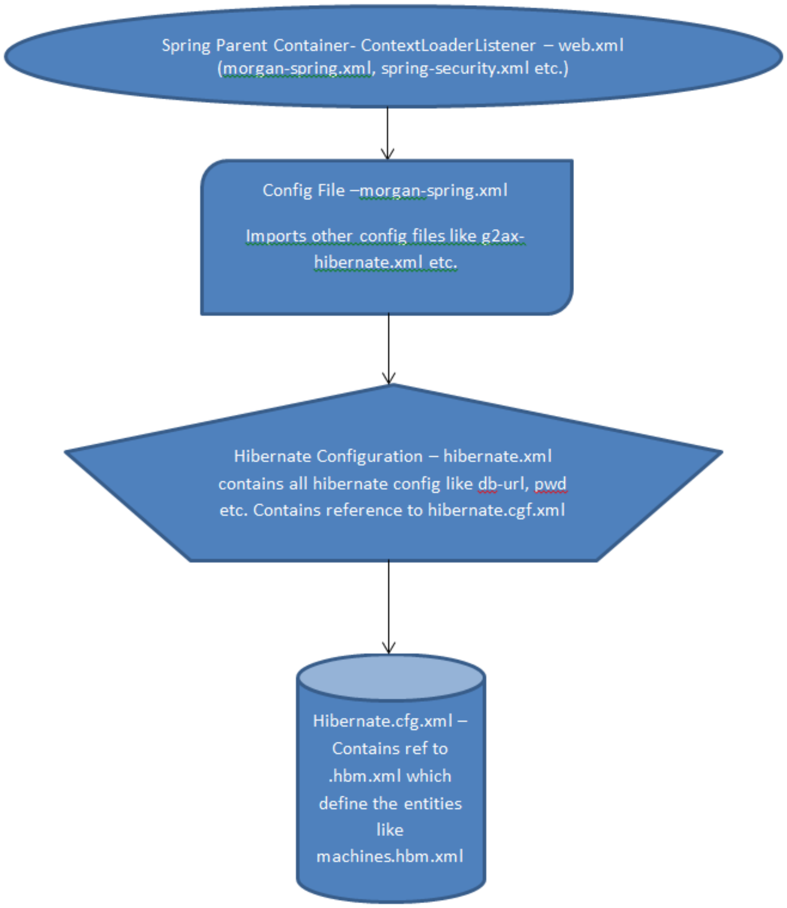
1. Hibernate instantiates your objects. So it needs to be able to instantiate them. If there isn't a **no-arg constructor**, Hibernate won't know how to instantiate it - what argument to pass.
2. The **@JoinColumn** annotation's name refers to the name of the foreign key in the target table.
3. ***"hibernate.hbm2ddl.auto***" Automatically validates or exports schema DDL to the database when the SessionFactory is created. List of possible options are: validate | update | create | create-drop.
4. Hibernate **SQL Dialect** is telling your Hibernate application which SQL language should be used to talk with your database.
   E.g. <prop key="dialect">*org.hibernate.dialect.MySQL5InnoDBDialect*</prop>
5. **Database Relationships**: refer Tutorial number 5.
6. ACID properties
7. What is transaction
8. How will u ensure two concurrent transactions are executed in one single operation
9. How spring @transactional works
10. **Default Isolation level for @Transactional annotation** is *ISOLATION_DEFAULT.* Use the default isolation level of the underlying data store. For Oracle its Read Committed. For MySQL it's Repeatable Read.
11. **.hbm.xml file** contains mapping for class (POJO) <-> table in database along with property <-> column name mapping and type of data.
12. 

Isolation Level	Phenomena Allowed
Read uncommitted	Dirty Read Non Repeatable Read Phantoms
Read committed	Non Repeatable Read Phantoms
Repeatable read	Phantoms
Serializable	-

13. 

	dirty reads	non-repeatable reads	phantom reads
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

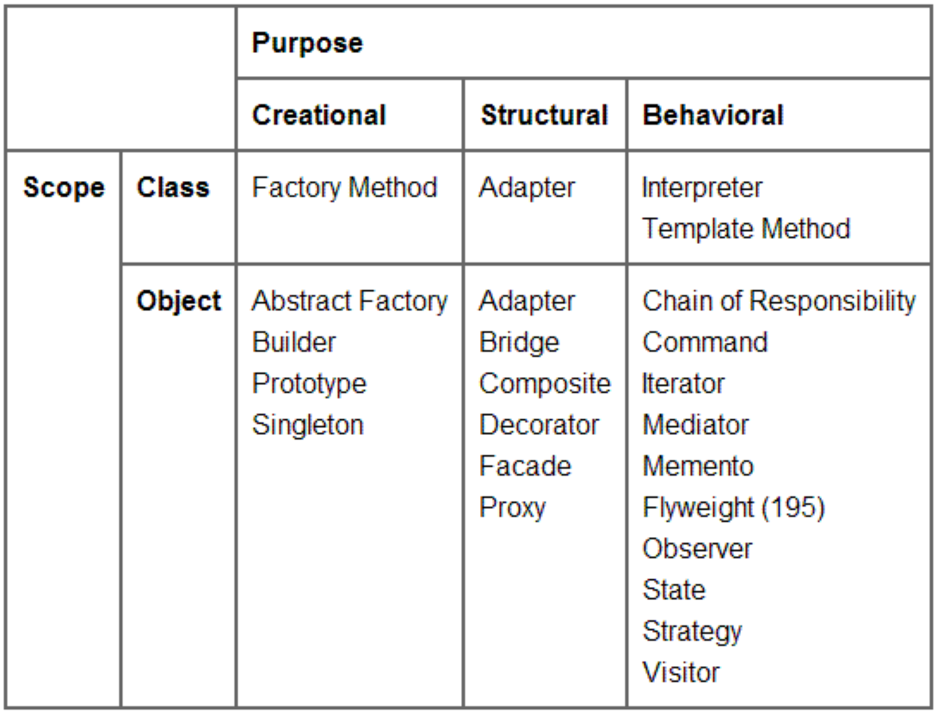
Refer link 31.


Spring Parent Container- ContextLoaderListener – web.xml
(morgan-spring.xml, spring-security.xml etc.)
Config File –morgan-spring.xml
Imports other config files like g2ax-hibernate.xml etc.
Hibernate Configuration – hibernate.xml contains all hibernate config like db-url, pwd etc. Contains reference to hibernate.cgf.xml
Hibernate.cfg.xml – Contains ref to .hbm.xml which define the entities like machines.hbm.xml

14.

15. **Propagation Levels:** Methods from distinct spring beans may be executed in the same transaction scope or actually being spanned across multiple nested transactions. Some of levels are:--
    a. Default is ***Propagation.REQUIRED.*** Means that the same transaction will be used if there is an already opened transaction in the current bean method execution context. If there is no existing transaction the spring container will create a new one.
    b. **REQUIRES_NEW** behavior means that a new physical transaction will always be created by the container. In other words the inner transaction may commit or rollback independently of the outer transaction, i.e. the outer transaction will not be affected by the inner transaction result.
    c. The **NESTED** behavior makes nested Spring transactions to use the same physical transaction but sets savepoints between nested invocations so inner transactions may also rollback independently of outer transactions.
    d. The **MANDATORY** behavior states that an existing opened transaction must already exist. If not an exception will be thrown by the container.

Design Patterns

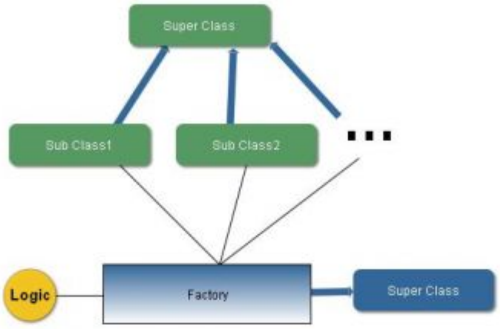


		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight (195) Observer State Strategy Visitor

Figure: Classification of Design Patterns

Creational Patterns: Define the best possible way in which an object can be instantiated.

1. **Factory Method**
   - i. If we have a super class and n sub-classes, and based on data provided, we have to return the object of one of the sub-classes, we use a factory pattern.
   - ii. Based on the arguments passed, factory method decides on which sub class to instantiate. This factory method will have the super class as its return type.
   - iii. When a class does not know which class of objects it must create.
   - iv. A class specifies its sub-classes to specify which objects to create.
   - v.



   - vi. E.g. If the sex is Male (M), it displays welcome message saying Hello Mr. <Name>. Person -> Male/Female -> Salutation Factory (returns Person type).

2. **Abstract Factory**
   i. One level of abstraction higher than factory pattern.
   ii. Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.
   iii. It isolates the concrete classes that are generated. Names of actual implementing classes aren't needed to be known at the client side.
   iv. E.g. Computers -> PC/Workstation/Server -> Different RAM, Processor etc. -> ComputerType Abstract Factory.

3. **Singleton**
   i. Some instances in the application where we have to use just one instance of a particular class.
   ii. E.g. Logger. We can't have more than one instance of Logger in the application otherwise the file in which we need to log will be created with every instance.
   iii.

```
public class Singleton {
   //initialized during class loading
   private static final Singleton INSTANCE = new Singleton();
   //to prevent creating another instance of Singleton
   private Singleton(){}
   public static Singleton getSingleton(){
      return INSTANCE;
   }
}
```

4. **Builder**
   i. Builds complex objects from simple ones step-by-step.
   ii. It separates the construction of complex objects from their representation.
   iii. The Builder pattern hides the internal details of how the product is built.
   iv. Because, each builder builds the final product step by step, we have more control on the final product.
   v. Should be used when no. of parameter required in constructor is more than manageable usually 4 or at most 5.
   vi. E.g.: MealBuilder(builds the entire meal for customers) :– Meal (Complex Object) -> Burger(Veg/Chicken) + Fries(Reg./Garlic) + Drink(Cola/Orange)

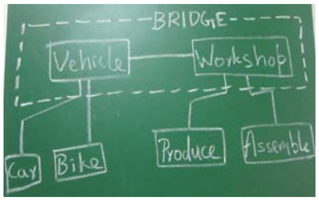
5. **Prototype**
   i. This implies cloning of an object to avoid creation.
   ii. If the cost of creating a new object is large and creation is resource intensive, we clone the object.
   iii. We use the interface Cloneable and call its method clone() to clone the object.
   iv. Drawback: Cannot use the clone as it is. You need to instantiate the clone before using it.
   v. Prototype doesn't require sub-classing, but it does require an "initialize" operation. Factory Method requires sub-classing, but doesn't require initialization.
   vi.

```
public Bike clone() {
        Bike byk=null;
```

```
    Try {
        byk = (Bike)super.clone();
    }
    Catch (CloneNotSupportedException e)
    {
        System.out.println(e.getMessage());
    }
        return byk;
}
```

Structural Patterns: Describes how objects/classes can be combined to form larger structures.

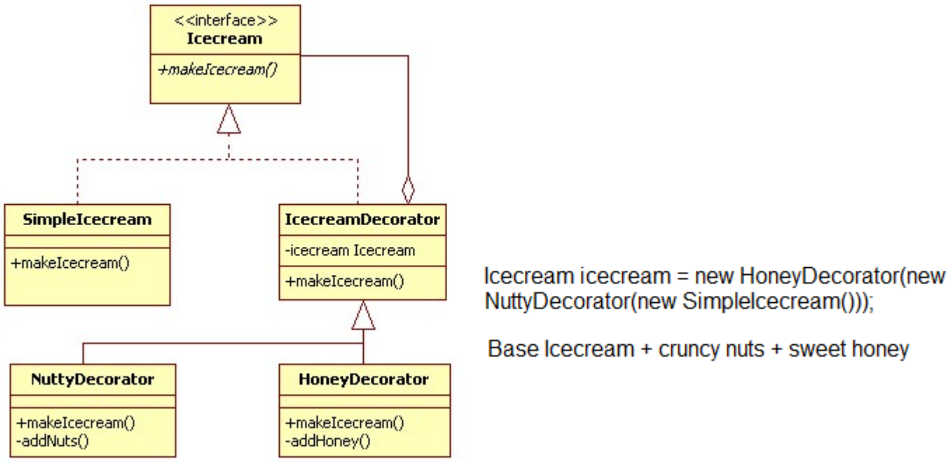
1. **Adapter**
   i. Used so that two unrelated interfaces can work together.
   ii. Something like we convert interface of one class into interface expected by the client.
   iii. Can be implemented by using inheritance method or composition method.
   iv. For e.g. we use a converter in India to convert 220V to 110V to support US devices.

2. **Bridge**
   i. Used to separate out the interface from its implementation. Doing this gives the flexibility so that both can vary independently.
   ii. For e.g., fan switch. The switch is the interface and the actual implementation is fan running when switched-on. Still, both the switch and the fan are independent of each other. Another switch can be plugged in for the fan and existing switch can be connected to light bulb.
   iii. 

   Vehicle: Abstract, Car/Bike: Fine Abstract, Workshop: Implementation
   iv. Creates two different hierarchies. One for abstraction and another for implementation
   v. Abstraction and implementation can be extended separately.
   vi. Should be used when we want to switch implementation at runtime.

3. **Composite**
   i. Allows you to create a tree like structure for simple and complex objects so they appear the same to the client.
   ii. A system consists of subsystems or components. Components can further be divided into smaller components. Further smaller components can be divided into smaller elements. This is a part-whole hierarchy.
   iii. For e.g. A car is made up of engine, tire … Engine is made up of electrical components, valves, … This hierarchy can be represented as a tree structure using composite design pattern.
   iv. Group of objects should be treated similarly as a single object.

4. **Decorator**

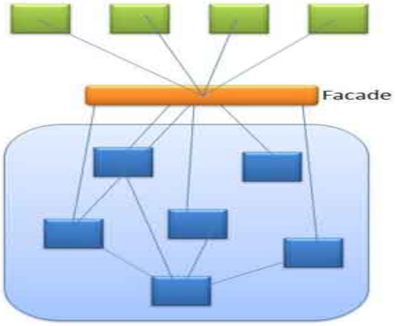
   i. The decorator pattern helps to add behavior or responsibilities to an object. This is also called "Wrapper".
   ii. Java Design Patterns suggest that Decorators should be abstract classes and the concrete implementation should be derived from them.
   iii. Implemented by constructing a wrapper around an object by extending its behavior.
   iv. 

```
Icecream icecream = new HoneyDecorator(new
NuttyDecorator(new SimpleIcecream()));
```

Base Icecream + cruncy nuts + sweet honey

   v. Code maintenance can be a problem as it provides the system with lot of similar looking small objects.

5. **Façade**

   i. Hides the complexities of the system and provides an interface to the client from where the client can access the system. The implementation is left to the vendor.
   ii. Façade also provides the implementation to be changed without affecting the client code.
   iii. Subsystem may be dependent with one another. In such case, facade can act as a coordinator and decouple the dependencies between the subsystems.
   iv. 

   v. E.g. Client => StoreKeeper (Façade) => Store => RawMaterials/FinishedGoods etc. The store keeper acts as the facade, as he hides the complexities of the system Store from Client.
   vi. Facade vs. Mediator: Subsystems are aware of the mediator. Subsystems are not aware of the existence of façade. Only façade talks to them.

6. **Flyweight**

   i. Used when there is a need to create high number of objects of almost similar nature.
   ii. Achieved by segregating object properties into intrinsic (essential) and extrinsic.
   iii. Use sharing objects to support large numbers of fine-grained objects efficiently.

iv. The object with intrinsic state is called flyweight object.

v. To create concrete objects we will have Flyweight factory. It ensures that the objects are shared and we don't end up creating duplicate objects.

vi. For e.g. consider a text editor. We will have LetterFactory that returns a concrete LetterObject for each of the letters A to Z (*intrinsic* property). This concrete object will be shared by clients. LetterFactory will ensure that we have only one shared instance. LetterObject can have other *extrinsic* properties like font, color, size etc. which can be passed through setters or constructor to create *n* number of objects with different properties.

7. **Proxy**

i. The proxy pattern is used when you need to represent a complex object with a simpler one.

ii. Proxy design pattern is also known as surrogate design pattern.

iii. *Adapter* provides a different interface from the real object and enables the client to use it to interact with the real object. Proxy provides same interface as in the real object.

iv. *Decorator* design pattern *adds behavior* at runtime to the real object. But, *Proxy* does not change the behavior instead it *controls the behavior*.

v. E.g. Sending an email with image attachment to million customers would be resource intensive. Instead we could send email with link to servlet which loads image i.e. placeholder/proxy of the image. Image will be loaded at run time when customer opens email.

vi. E.g. ATM acts as a proxy for Bank and its related operations.

Behavioral Patterns: are those which are concerned with interactions between the objects.

1. **Chain of Responsibility**

i. It decouples the sender of the request to the receiver. Distributes responsibilities.

ii. Sender will not know which object in the chain will serve its request.

iii. There might be a problem when the request is never sent to the object which can handle it!

iv. Based on the request data sent, the receiver is picked. This is called "data-driven".

v. The responsibility of handling the request data is given to any of the members of the "chain". If the 1st link of the chain can't handle the responsibility, request data is passed to the next level in the chain, i.e. to the next link.

vi. There can be such a scenario when none of the objects in the chain can handle the request. In this case, the chain will discard the request.

vii. E.g. Java Exception Hierarchy, multiple catch blocks, coin sorting machine, ATM money dispenser etc.

2. **Command**

i. Used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

ii. Map a command to an object/module which can handle the request.

iii. Unlike COR, it doesn't send the request to all the objects instead to one specific object.

iv. Command pattern helps to decouple the invoker and the receiver. Receiver is the one which knows how to perform an action.

v. Four terms always associated with the command pattern are command, receiver, invoker and client.

vi. E.g. You (Client) => Switch (Invoker) => Up/Down (Command) => Light/Fan (Receiver).

3. **Interpreter**

i. Defines a grammatical representation for a language and an interpreter to interpret the grammar.

ii. Java itself is an interpreted language.

iii. E.g. The “musical notes” is an “Interpreted Language”. The musicians read the notes, interpret them according to “Sa, Re, Ga, Ma…” and play the instruments, what we get in output is musical sound waves.

iv. If you are using interpreter pattern, you need checks for grammatical mistakes etc.

v. This is not a very common pattern.

4. **Iterator**

i. Iterator should be implemented as an interface. This allows the user to implement it anyway its easier for him/her to return data.

ii. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

iii. Take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object.

iv. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

v. Provides a uniform interface for traversing different collection.

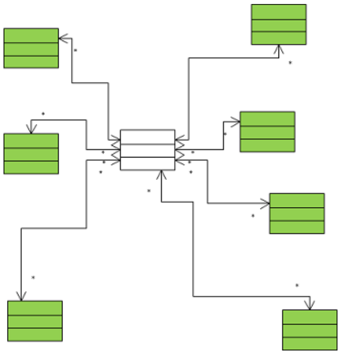
vi. E.g. Java Iterator over Collections.

5. **Mediator**

i. The mediator pattern deals with the complexity which comes in the coding when number of classes increase.

ii. It promotes loose-coupling of classes such that only one class (Mediator) has the knowledge of all the classes, rest of the classes have their responsibilities and only interact with the Mediator.

iii.



iv. E.g. airplanes interacting with the control tower and not among themselves.

v. E.g. stock exchange acts like a mediator and the traders did not need to know other traders and services provided by them to make a deal.

vi. This brings standardization, which might be cumbersome. Also, there might be a slight loss in efficiency.

6. **Memento**

7. **Observer**

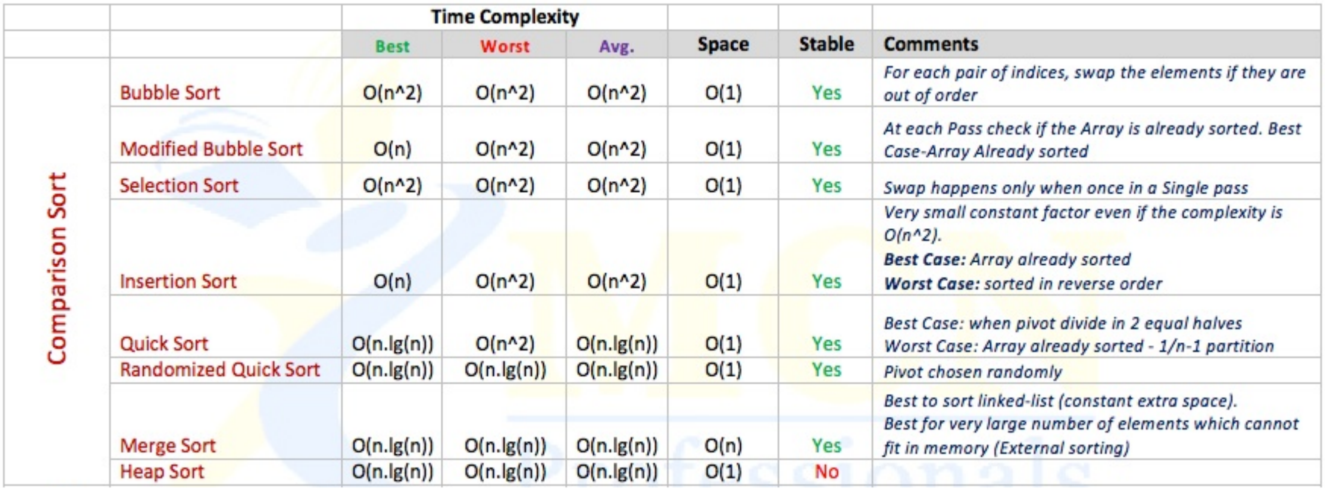
8. **State**

9. **Strategy**

10. **Template**

11. **Visitor**

Sorting Algorithms



		Time Complexity					
		Best	Worst	Avg.	Space	Stable	Comments
Comparison Sort	Bubble Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*For each pair of indices, swap the elements if they are out of order*
	Modified Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*At each Pass check if the Array is already sorted. Best Case-Array Already sorted*
	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*Swap happens only when once in a Single pass*
	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*Very small constant factor even if the complexity is O(n^2).* ***Best Case:** Array already sorted* ***Worst Case:** sorted in reverse order*
	Quick Sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	Yes	*Best Case: when pivot divide in 2 equal halves* *Worst Case: Array already sorted - 1/n-1 partition*
	Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	*Pivot chosen randomly*
	Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	*Best to sort linked-list (constant extra space).* *Best for very large number of elements which cannot fit in memory (External sorting)*
	Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	

Sorting Algorithm Properties:

1. Adaptive - performance adapts to the initial order of elements.
2. Stable - insertion sort retains relative order of the same elements.
3. In-place - requires constant amount of additional space.
4. Online - new elements can be added during the sort.

1. Bubble Sort

a. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one).

b. After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

c. http://upload.wikimedia.org/wikipedia/commons/c/c8/Bubble-sort-example-300px.gif

2. Selection Sort

a. The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array.

b. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

c. https://upload.wikimedia.org/wikipedia/commons/9/94/Selection-Sort-Animation.gif

3. Insertion Sort

a. Insertion sort is a comparison sort algorithm, which works similar to the way we arrange the cards in a hand. We take one element at a time, starts compare from one end and them place them in between the card lesser and greater than it.

b. March up the array, checking each element. If larger (than what's in previous position checked), leave it If smaller then march back down, shifting larger elements up until encounter a smaller element. Insert there.

c. http://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

4. **Quick Sort**

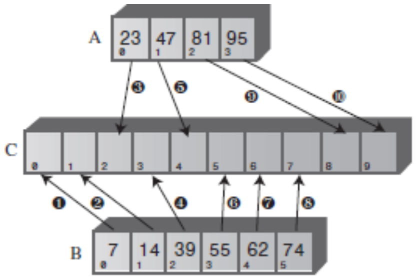
   a. Divide and conquer strategy.

   b. Choose a pivot value. We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.

   c. Partition. Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.

   d. Sort both parts. Apply quicksort algorithm recursively to the left and the right parts.

5. **Merge Sort**

   a. 

   b.
```
public static void merge( int[] arrayA, int sizeA, int[] arrayB, int sizeB, int[] arrayC ) {
        int aDex=0, bDex=0, cDex=0;
        while(aDex < sizeA && bDex < sizeB)                 // neither array empty
                if( arrayA[aDex] < arrayB[bDex] )
                        arrayC[cDex++] = arrayA[aDex++];
                else
                        arrayC[cDex++] = arrayB[bDex++];
        while(aDex < sizeA)                                 // arrayB is empty,
                arrayC[cDex++] = arrayA[aDex++];            // but arrayA isn't
        while(bDex < sizeB) // arrayA is empty,
                arrayC[cDex++] = arrayB[bDex++];            // but arrayB isn't
}                                                           // end merge()
```

6. **Shell Sort**

   a.

Data Structures

Basics

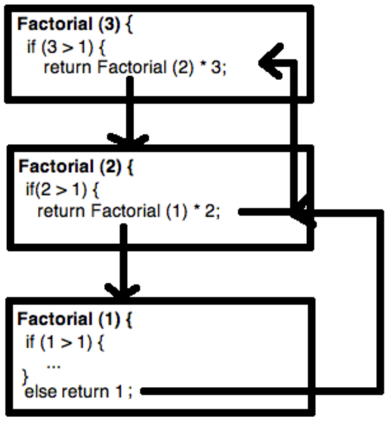
i. **Recursion:**

a. It calls itself.

b. When it calls itself, it does so to solve a smaller problem.

c. There’s some version of the problem that is simple enough that the routine can solve it, and return (***Base Case***), without calling itself.

d.



```
Factorial (3) {
 if (3 > 1) {
   return Factorial (2) * 3;
```

```
Factorial (2) {
 if(2 > 1) {
   return Factorial (1) * 2;
```

```
Factorial (1) {
 if (1 > 1) {
   ...
}
else return 1 ;
```

e. Arguments to the method and the address to which the method should return are pushed onto an internal stack so that the method can access the argument values and know where to return.

ii. **Topics:**

- General-purpose data structures: arrays, linked lists, trees, hash tables.
- Specialized data structures: stacks, queues, priority queues, graphs.
- Sorting: insertion sort, Shellsort, quicksort, mergesort, heapsort.
- Graphs: adjacency matrix, adjacency list.
- External storage: sequential storage, indexed files, B-trees, hashing.

1. **Stacks - ADT**

a. It follows ***first in last out*** strategy.

b. It can be used for problems like *string reversal*, *checking delimiters*, etc. Push for opening delimiter and pop for closing delimiter. Compare when you pop to see if opening delimiter corresponds to closing one.

c. public **Stack**(int s) // constructor

```
{
        maxSize = s; // set array size
        stackArray = new long[maxSize]; // create array
        top = -1; // no items yet
}
```

d. public void **push**(long j) // put item on top of stack

```
{
        stackArray[++top] = j; // increment top, insert item
}
```

e. public long **pop**() // take item from top of stack

```
{
        return stackArray[top--]; // access item, decrement top
}
```

f. public long **peek**() // peek at top of stack

```
{
        return stackArray[top];
}
```

g. public boolean **isEmpty**() // true if stack is empty

```
{
        return (top == -1);
}
```

h. public boolean **isFull**() // true if stack is full

```
{
        return (top == maxSize-1);
}
```

2. **Queue - ADT**

a. It follows ***first in first out*** strategy.

b. public **Queue**(int s) // constructor

```
{
        maxSize = s;
        queArray = new long[maxSize];
        front = 0;
        rear = -1;
        nItems = 0;
}
```

c. public void **insert**(long j) // put item at rear of queue

```
{
        if(rear == maxSize-1)     // deal with wraparound
                rear = -1;
        queArray[++rear] = j;     // increment rear and insert
        nItems++;                 // one more item
}
```

d. public long **remove**() // take item from front of queue

```
{
        long temp = queArray[front++]; // get value and increment front
        if(front == maxSize)      // deal with wraparound
                front = 0;
        nItems--;                 // one less item
        return temp;
}
```

e. public long **peekFront**() // peek at front of queue

```
{
        return queArray[front];
}
```

f. public boolean **isEmpty**() // true if queue is empty

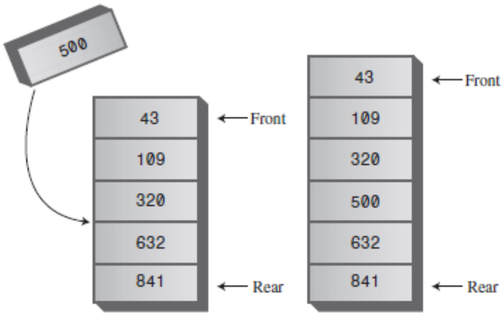
```
{
        return (nItems==0);
}
```

g. public boolean **isFull**() // true if queue is full

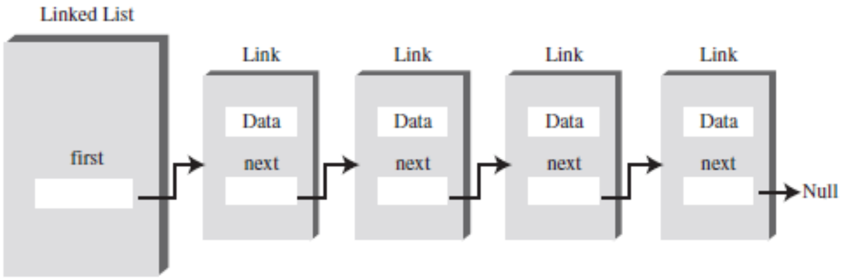
```
{
        return (nItems==maxSize);
}
```

h. public int **size**() // number of items in queue

```
{
        return nItems;
}
```

3. **Priority Queue**
    a. A specializer implementation of queue.
    b. Items are ordered in the queue based on the key value. Lowest value can be in front (ascending) or rear (descending).
    c. 

New item inserted in priority queue
    d. Insertion runs in O(N) time, while deletion takes O(1) time.
    e. Priority Queue can be implemented using Heaps. Insertion and Deletion (*of root node*) are the order of height of the tree, i.e., O(log N).

4. **LinkedLists**
    a. 

    b. public class **Link** {

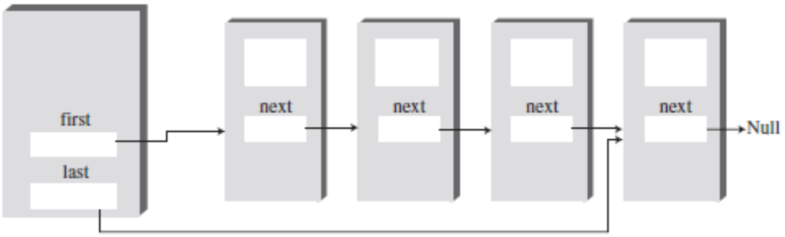
```
        public int data;
        public Link next;
```

```
    public Link (int data) {
        this.data = data;
    }
}
```

c. public class **LinkList** {

```
    Link firstLink;
    public void isEmpty {
        return (firstLink == null);
    }
    public void insertAtBeginning(int data) {
        Link newLink = new Link(data);
        newLink.next = firstLink;
        firstLink = newLink;
    }
    public Link delete(int key) {
        Link currentLink = firstLink;
        Link previousLink = firstLink;
        while(currentLink.data != key) {
            if (currentLink.next == null) {   // didn't find break n return null
                return null;
            } else {
                previousLink = currentLink;
                currentLink = currentLink.next;
            }
        } // exits while when link is found.
        If (currentLink == firstLink) {
            firstLink = currentLink.next;
        } else {
            previousLink.next = currentLink.next;
        }
        return currentLink;
    }
}
```

5. **<u>Double-Ended LinkedList</u>**

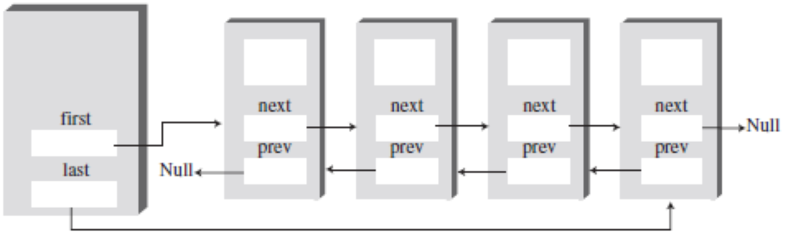
a. 

b. It has one additional feature: a reference to the last link.

c. The reference to the last link permits you to insert a new link directly at the end of the list.

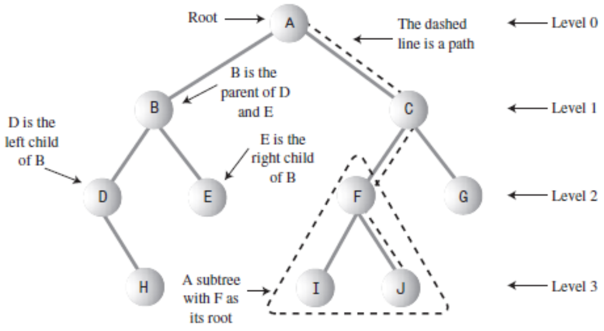
d. It’s conceptually different from doubly linked list.

6. **Doubly LinkedList**

a. 

b.
```
Class Link {
            int data;
            Link next;
            Link previous;
}
Class DoublyLinkList {
            Link first;
            Link last;
            Public DoublyLinkList() {
                        first = null;
                        last = null;
            }
}
```

7. **Trees**

a. 
Root
A
The dashed line is a path
Level 0
B is the parent of D and E
B
C
Level 1
D is the left child of B
E is the right child of B
D
E
F
G
Level 2
H
A subtree with F as its root
I
J
Level 3

b. A node in a tree can have more than 2 child nodes.

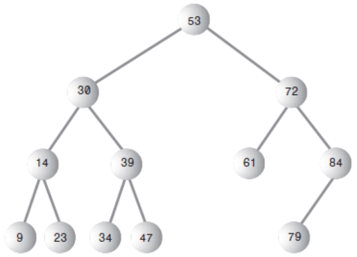
8. **Binary Trees**

a. Every node in the tree can have at most two child nodes.

b. A node’s left child must have value less then itself (parent).

c. A node’s right child must have value greater than or equal to itself (parent).

d.



e.

```
Class Node {
        int data; // Could also be an object reference like Person p.
        Node leftChild;
        Node rightChild;
}
```

f. **Finding a node**: O(log n)

```
public Node find (int key) {
        Node current = root;
        while (current.data != key) {
                if (key < current.data) {                    // go left
                        current = current.leftChild;
                } else {                                     // go right
                        current = current.rightChild;
                }
                If (current == null) {
                        return null;                         // not found
                }
        }
        return current;
}
```

g. **Inserting a node** is similarly to finding a node and as soon as you encounter null, you know the place to insert the node.

```
while(true) // (exits internally)
{
        parent = current;
        if(id < current.iData) // go left?
        {
                current = current.leftChild;
                if(current == null) // if end of the line,
                { // insert on left
                        parent.leftChild = newNode;
                        return;
                }
        }               // end if go left
        else {.....}    // go to right
}
```

h. **Deleting a node**

   i.

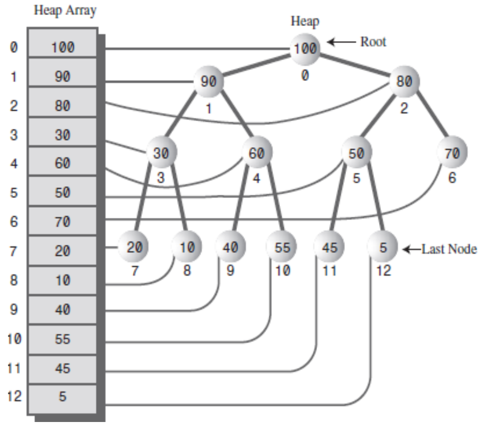
i. **Inorder Traversal:**

   i. An inorder traversal of a binary search tree will cause all the nodes to be visited in ascending order, based on their key values.

   ii. private void inOrder(node localRoot) {

```
private void inOrder(node localRoot) {
            if(localRoot != null){
                        inOrder(localRoot.leftChild);
                        System.out.print(localRoot.iData + " ");
                        inOrder(localRoot.rightChild);
            }
}
```

9. **Heaps**

a. 

Heap implemented using array.

b. **Insertion**:

   i. The node is inserted at the end of the array used to create heap.

   ii. Then it's compared with its parent (index - 1)/2 until it's not smaller than its parent (max heap) or it has become the root.

c. **Removal**:

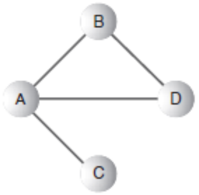
   i. The root is always removed. It gives us the current max (max heap) or current min (min heap).

   ii. The hole created by the root is replaced by the element at end of the heap array.

   iii. Then the node is trickled down and compared with both left (2 * index + 1) and right (left + 1) child.

   iv. It's swapped with a greater child (max heap) at every step.

10. **Graphs**

a. 

a) Connected Graph

b.

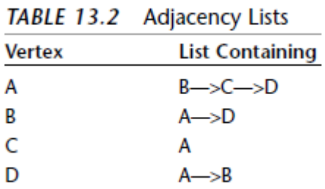


TABLE 13.2 Adjacency Lists

Vertex	List Containing
A	B—>C—>D
B	A—>D
C	A
D	A—>B

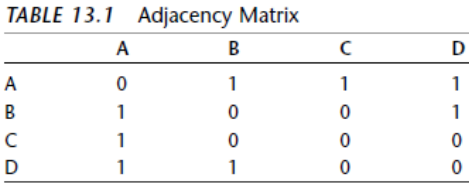


TABLE 13.1 Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

c.

```
class Vertex
{
        public char label; // label (e.g. ‘A’)
        public boolean wasVisited;
        public Vertex(char lab) // constructor
        {
                label = lab;
                wasVisited = false;
        }
}
```

d.

```
class Graph {
        private final int MAX_VERTS = 20;
        private Vertex vertexList[]; // array of vertices
        private int adjMat[][]; // adjacency matrix
        private int nVerts; // current number of vertices

        public Graph() {
                vertexList = new Vertex[MAX_VERTS];
                // adjacency matrix
                adjMat = new int[MAX_VERTS][MAX_VERTS];
                nVerts = 0;
                for(int j=0; j<MAX_VERTS; j++) // set adjacency
                        for(int k=0; k<MAX_VERTS; k++) // matrix to 0
                                adjMat[j][k] = 0;
        } // end constructor

        public void addVertex(char lab) {    // argument is label
                vertexList[nVerts++] = new Vertex(lab);
        }

        public void addEdge(int start, int end) {
                adjMat[start][end] = 1;
                adjMat[end][start] = 1;
        }

        public void displayVertex(int v) {
                System.out.print(vertexList[v].label);
        }
} // end class Graph
```

e. **Depth First Search (DFS)** Carried out using a STACK.

i. Rule 1: If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

ii. Rule 2: If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

iii. Rule 3: If you can't follow Rule 1 or Rule 2, you're done.

iv. Get an unvisited vertex adjacent to v

```
public int getAdjUnvisitedVertex(int v) {
        for(int j=0; j<nVerts; j++) {
                if(adjMat[v][j] == 1 && vertexList[j].wasVisited == false)
                        return j; // return first such vertex
        }
        return -1; // no such vertices
} // end getAdjUnvisitedVertex()
```

v.

```
public void dfs(){ // begin at vertex 0
        vertexList[0].wasVisited = true;  // mark it
        displayVertex(0);                 // display it
        theStack.push(0);                 // push it
        while( !theStack.isEmpty() ) {    // until stack empty, get an unvisited vertex
                                          //adjacent to stack top
                int v = getAdjUnvisitedVertex( theStack.peek() );
                if(v == -1) // if no such vertex,
                        theStack.pop(); // pop a new one
                else {                    // if it exists,
                        vertexList[v].wasVisited = true; // mark it
                        displayVertex(v); // display it
                        theStack.push(v); // push it
                }
                } // end while
}
```

f. **Breadth First Search** (**BFS**) Carried out using a QUEUE.

i. Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.

ii. Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.

iii. Rule 3: If you can't carry out Rule 2 because the queue is empty, you're done.

iv.

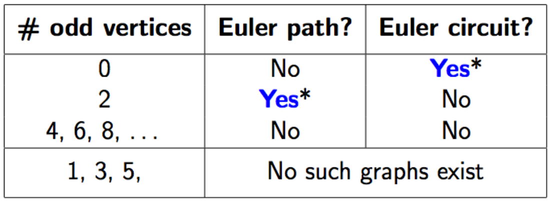
```
public int getAdjUnvisitedVertex(int v) {
        for(int j=0; j<nVerts; j++)
                if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
                        return j;
        return -1;
}
```

v.

```
public void bfs() {           // breadth-first search, begin at vertex 0
        vertexList[0].wasVisited = true;            // mark it
        displayVertex(0);                       // display it
        theQueue.insert(0);                     // insert at tail
        int v2;
        while( !theQueue.isEmpty()) {           // until queue empty
            int v1 = theQueue.remove();         // remove vertex at head
            // until it has no unvisited neighbors
            while( (v2=getAdjUnvisitedVertex(v1)) != -1 ) {   // get one
                vertexList[v2].wasVisited = true;             // mark it
                displayVertex(v2);                            // display it
                theQueue.insert(v2);                          // insert it
            } // end while
        } // end while(queue not empty)
        // queue is empty, so we're done
} // end bfs()
```

g. **Euler's Path/Circuit –** Eularian path is a path in graph that visits every edge exactly once. Eularian circuit is a Eularian path, which starts and ends on the same vertex.

Condition to check if a graph has Euler path/circuit: -



# odd vertices	Euler path?	Euler circuit?
0	No	Yes*
2	Yes*	No
4, 6, 8, . . .	No	No
1, 3, 5,	No such graphs exist	

*Graph should be connected.

Find Eulers path (Fleury's algorithm):

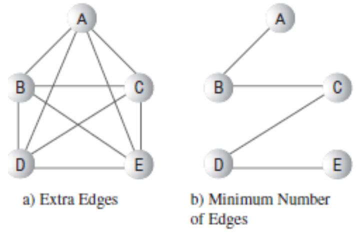
i. Make sure the graph has either 0 or 2 odd vertices.
ii. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
iii. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, always choose the non-bridge.
iv. Stop when you run out of edges.

Problem Example: --

Given a set of names, sort them in the following manner the next word should start with the last letter of the previous word. For e.g. "abc" - > "cde" -> "efg" -> "glm"

11. **Minimum spanning trees**

a.



a) Extra Edges

b) Minimum Number of Edges

b. No. of edges E in a minimum spanning tree is always one less than the number of vertices V: *E = V – 1*.
c. It can be either implemented using DFS or BFS. Algorithm almost identical to searches except that we have to print the edges travelled.

Tutorials/Useful Sites/FAQs

1. REST API: http://briansjavablog.blogspot.com/2012/08/rest-services-with-spring.html
2. Java Patterns: http://www.allapplabs.com/java_design_patterns/creational_patterns.htm
3. Java Patterns: http://javapapers.com/design-patterns/introduction-to-design-patterns/
4. Java Patterns: http://www.oodesign.com/
5. Spring 3.2: http://static.springsource.org/spring/docs/3.2.0.RELEASE/spring-framework-reference/html/index.html
6. Spring 3.1 MVC ambiguous mapping: http://stackoverflow.com/questions/8909482/spring-mvc-3-ambiguous-mapping-found
7. http://www.techopedia.com/it-dictionary/tags/data-management
8. http://viralpatel.net/blogs/
9. Spring-Exception-Handling:-- http://steveliles.github.com/configuring_global_exception_handling_in_spring_mvc.html
10. http://www.javaworld.com/community/node/8657
11. http://www.stormpath.com/blog/spring-mvc-rest-exception-handling-best-practices-part-2
12. String Questions: http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html
13. Comparator v/s Comparable: http://www.digizol.com/2008/07/java-sorting-comparator-vs-comparable.html
14. Java Heap Memory: http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html
15. Java Pass by reference v/s value: http://www.programmerinterview.com/index.php/java-questions/does-java-pass-by-reference-or-by-value/
16. Spring Custom DateTime Editor: http://joda-interest.219941.n2.nabble.com/Joda-Spring-td2237700.html
17. Init Binder: http://blog.jamesrossiter.co.uk/2013/03/26/use-a-spring-initbinder-to-resolve-type-mismatch-and-bind-exceptions-in-post-from-spring-framework-mvc-forms-to-controller-actions/
18. Spring-security-Filter-Chain: http://static.springsource.org/spring-security/site/docs/3.2.x/reference/security-filter-chain.html
19. Spring Security By Example: http://blog.solidcraft.eu/2011/03/spring-security-by-example-set-up-and.html
20. Annotation-Driven: http://stackoverflow.com/questions/3977973/whats-the-difference-between-mvcannotation-driven-and-contextannotation
21. Substring Memory Leak: http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html
22. Why String is Immutable: http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html
23. Autoboxing: http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html
24. Equals and hash: http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html
25. How hashing works; http://www.javamex.com/tutorials/collections/hashing_intro.shtml
26. Advanced Multithreading: http://inheritingjava.blogspot.com/p/multithreading-advanced-concepts.html
27. Thread Interview Qs: http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html

28. Synchronization: http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html
29. Forward v/s Redirect: http://javapapers.com/jsp/difference-between-forward-and-sendredirect/
30. API Façade Patterns http://www.slideshare.net/apigee/api-facade-patterns-composition
31. Isolation Levels: http://www.byteslounge.com/tutorials/spring-transaction-isolation-tutorial
32. Abstraction: http://javapapers.com/core-java/java-abstraction/