

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| **1) ArrayList is not synchronized.** | **Vector is synchronized.** |
| **2) ArrayList increments 50% of current array size if number of element exceeds from its capacity.** | **Vector increments 100% means doubles the array size if total number of element exceeds than its capacity.** |
| **3) ArrayList is not a legacy class, it is introduced in JDK 1.2.** | **Vector is a legacy class.** |
| **4) ArrayList is fast because it is non-synchronized.** | **Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.** |
| **5) ArrayList uses Iterator interface to traverse the elements.** | **Vector uses Enumeration interface to traverse the elements. But it can use Iterator also.** |

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| **1) HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code.** | **Hashtable is synchronized. It is thread-safe and can be shared with many threads.** |
| **2) HashMap allows one null key and multiple null values.** | **Hashtable doesn't allow any null key or value.** |
| **3) HashMap is a new class introduced in JDK 1.2.** | **Hashtable is a legacy class.** |
| **4) HashMap is fast.** | **Hashtable is slow.** |
| **5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);** | **Hashtable is internally synchronized and can't be unsynchronized.** |
| **6) HashMap is traversed by Iterator.** | **Hashtable is traversed by Enumerator and Iterator.** |
| **7) Iterator in HashMap is fail-fast.** | **Enumerator in Hashtable is not fail-fast.** |
| **8) HashMap inherits AbstractMap class.** | **Hashtable inherits Dictionary class.** |

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| **1) Abstract class can have abstract and non-abstractmethods.** | **Interface can have only abstract methods. Since Java 8, it can have default and static methods also.** |
| **2) Abstract class doesn't support multiple inheritance.** | **Interface supports multiple inheritance.** |
| **3) Abstract class can have final, non-final, static and non-static variables.** | **Interface has only static and final variables.** |
| **4) Abstract class can provide the implementation of interface.** | **Interface can't provide the implementation of abstract class.** |
| **5) The abstract keyword is used to declare abstract class.** | **The interface keyword is used to declare interface.** |
| **6) An abstract classcan extend another Java class and implement multiple Java interfaces.** | **An interface can extend another Java interface only.** |
| **7) An abstract classcan be extended using keyword ?extends?.** | **An interface classcan be implemented using keyword ?implements?.** |
| **8) A Javaabstract classcan have class members like private, protected, etc.** | **Members of a Java interface are public by default.** |
| **9)Example: public abstract class Shape{ public abstract void draw(); }** | **Example: public interface Drawable{ void draw(); }** |
| **10. You cannot create the object from abstract class but runtime environment can create an object for you.** | **10. Object cannot created at all.** |

**ABSTRACT CLASS SCENARIO:** Suppose there a requirement for which client has to give implementation for 2 methods and the remaining 3 method implementations have to be provided by the developer. The **Client method implementation** could be provided in an abstract class and only **method declaration** of developer’s methods can be kept in that abstract class. The implementations could be provided in child class.

**Difference between constructor and setter injection**

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

|  |  |  |
| --- | --- | --- |
| **No.** | **SOAP** | **REST** |
| **1)** | **SOAP is a protocol.** | **REST is an architectural style.** |
| **2)** | **SOAP stands for Simple Object Access Protocol.** | **REST stands for REpresentational State Transfer.** |
| **3)** | **SOAP can't use REST because it is a protocol.** | **REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.** |
| **4)** | **SOAP uses services interfaces to expose the business logic.** | **REST uses URI to expose business logic.** |
| **5)** | **JAX-WS is the java API for SOAP web services.** | **JAX-RS is the java API for RESTful web services.** |
| **6)** | **SOAP defines standards to be strictly followed.** | **REST does not define too much standards like SOAP.** |
| **7)** | **SOAP requires more bandwidth and resource than REST.** | **REST requires less bandwidth and resource than SOAP.** |
| **8)** | **SOAP defines its own security.** | **RESTful web services inherits security measures from the underlying transport.** |
| **9)** | **SOAP permits XML data format only.** | **REST permits different data format such as Plain text, HTML, XML, JSON etc.** |
| **10)** | **SOAP is less preferred than REST.** | **REST more preferred than SOAP.** |

**ANNOTATIONS**

If you annotate a method with **@ResponseBody**, spring will try to convert its return value and write it to the http response automatically.

Now what the annotation means is that the returned value of the method will constitute the body of the HTTP response. Of course, an HTTP response can't contain Java objects. So this list of accounts is transformed to a format suitable for REST applications, typically JSON or XML.

If you annotate a methods parameter with **@RequestBody**, spring will try to convert the content of the incoming request body to your parameter object on the fly. **HTTP** Message converters convert the **HTTP** request body into domain object based on **Accept** header present in the request.

**@RequestParam** annotated parameters get linked to specific Servlet request parameters. Parameter values are converted to the declared method argument type. This annotation indicates that a method parameter should be bound to a web request parameter.

The **@PathVariable** annotation is used to extract the value of the URI template variables and assign their value to a method variable.

We have added the ***“jackson-databind”*** as our dependency, Spring invokes inbuilt **MappingJackson2HttpMessageConverter** to convert the response into JSON.

**<context:component-scan>** will let the Spring Container to search for all the annotation under the package **“com.javainteriviewpoint”.**

**<mvc:annotation-driven/>** annotation will activate the **@Controller, @RequestMapping, @Valid** etc annotations.

**dataSource** : This bean holds all the database related configurations such as driverClassName, url, username, password.

**transactionManager**: We are using the JpaTransactionManager for managing the transactions for our application, we will be passing the **entityManagerFactoryBean** reference to it.

|  |  |  |
| --- | --- | --- |
| **No.** | **Class Variable** | **Instance Variable** |
| **1)** | **Class variables are declared with keyword static.** | **Instance variables are declared without static keyword.** |
| **2)** | **Class variables are common to all instances of a class. These variables are shared between the objects of a class.** | **Instance variables are not shared between the objects of a class. Each instance will have their own copy of instance variables.** |
| **3)** | **As class variables are common to all objects of a class, changes made to these variables through one object will reflect in another.** | **As each object will have its own copy of instance variables, changes made to these variables through one object will not reflect in another object.** |
| **4)** | **Class variables can be accessed using either class name or object reference.** | **Instance variables can be accessed only through object reference.** |

**SERIALIZATION**

Serialization is usually used When the need arises to send your data over network or stored in files. By data I mean objects and not text.

Now the problem is your Network infrastructure and your Hard disk are hardware components that understand bits and bytes but not JAVA objects.

Serialization is the translation of your Java object's values/states to bytes to send it over network or save it.

This is analogous to how your voice is transmitted over PSTN telephone lines.

Serialization is simply turning an existing object into a byte array. This byte array represents the class of the object, the version of the object, and the internal state of the object. This byte array can then be used between JVM's running the same code to transmit/read the object.

Serializable is a **marker interface** like Cloneable and Remote.

**Why would we want to do this?**

There are several reasons:

**Communication**: If you have two machines that are running the same code, and they need to communicate, an easy way is for one machine to build an object with information that it would like to transmit, and then serialize that object to the other machine. It's not the best method for communication, but it gets the job done.

**Persistence**: If you want to store the state of a particular operation in a database, it can be easily serialized to a byte array, and stored in the database for later retrieval.

**Deep Copy**: If you need an exact replica of an Object, and don't want to go to the trouble of writing your own specialized clone() class, simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.

**Caching**: Really just an application of the above, but sometimes an object takes 10 minutes to build, but would only take 10 seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed.

**Cross JVM Synchronization**: Serialization works across different JVMs that may be running on different architectures.

**Serialization ID – Lets JVM know which class has been serialized**

**JMeter(Load Testing Tool) –** Load is increased by increasing no. of users that hit a particular service simultaneously.

Easily identifies memory leak

Suppose, a query is directly hit in DB, it takes less time than when the same query is hit through JAVA code. JMeter can tell exactly which method/query is taking how much time so that tuning of that method can be done in code level.

**Performance Tuning**

Configuration level changes: Example – How much additional RAM is required to handle the incoming requests, etc.

**Deployment**

PUTTY: Start and stop server

WINSCP: Deployment of WAR

Process: Login through PUTTY and stop server first.

Login to WinSCP and go to the required path.

Take backup of deployed WAR from WinSCP.

Build WAR using “mvn clean install” from command prompt

Paste the new WAR in the path in WinSCP.

Go back to PUTTY and start the server again.

**STACK AND HEAP MEMORY**

package com.journaldev.test;

public class Memory {

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

int i=1; // Line 2

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

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

mem.foo(obj); // Line 5

} // Line 9

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

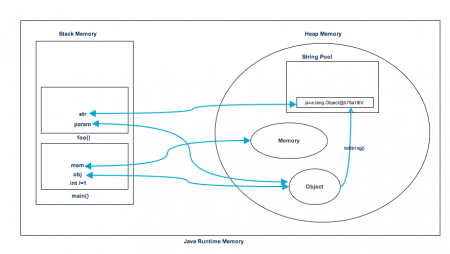
String str = param.toString(); //// Line 7

System.out.println(str);

} // Line 8

}

Below image shows the Stack and Heap memory with reference to above program and how they are being used to store primitive, Objects and reference variables.



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

* As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in line 3, it’s created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
* Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
* A string is created in line 7, it goes in the [String Pool](https://www.journaldev.com/797/what-is-java-string-pool) in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

**PROGRAM COUNTER REGISTER (PC REGISTER)**

A program counter is a **register** in a **computer** processor that contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 1.

**INTERPRETER AND JIT COMPILER**

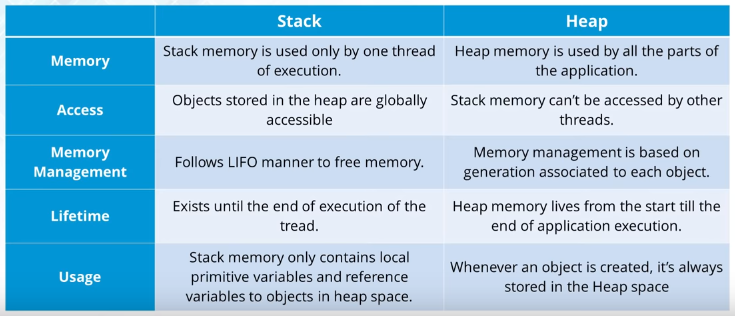
**1. Interpreter**

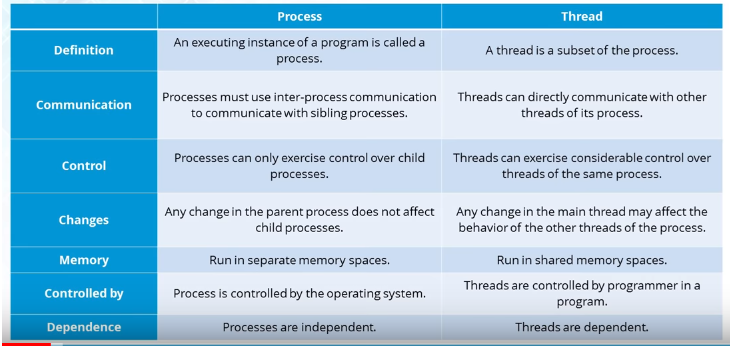
It is responsible to read byte code and interpret into machine code (native code) and execute that machine code line by line. The problem with interpret is it interprets every time even some method invoked multiple times which effects performance of the system. To overcome this problem SUN people introduced JIT compilers in 1.1 V.

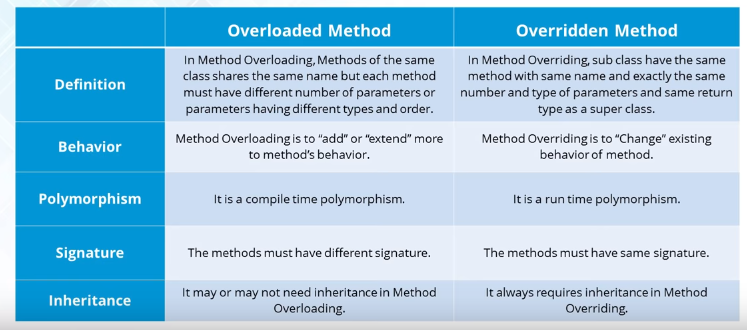
**2. JIT Compiler**

The JIT compiler has been introduced to compensate for the disadvantages of the interpreter. The main purpose of JIT compiler is to improve the performance. Internally JIT compiler maintains a separate count for every method. Whenever JVM across any method call, first that method will be interpreted normally by the interpreter and JIT compiler increments the corresponding count variable.

This process will be continued for every method once if any method count reaches thread hold value then JIT compiler identifies that method is a repeatedly used method (Hotspot) immediately JIT compiler compiles that method and generates corresponding native code. Next time JVM come across that method call then JVM directly uses native code and executes it instead of interpreting once again, so that performance of the system will be improved. Threshold is varied from JVM to JVM. Some advanced JIT compilers will recompile generated native code if count reaches threshold value second time so that more optimized code will be generated.



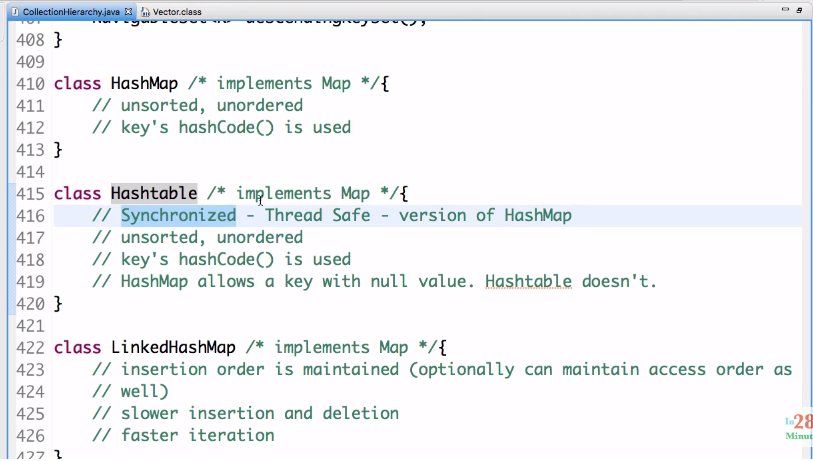


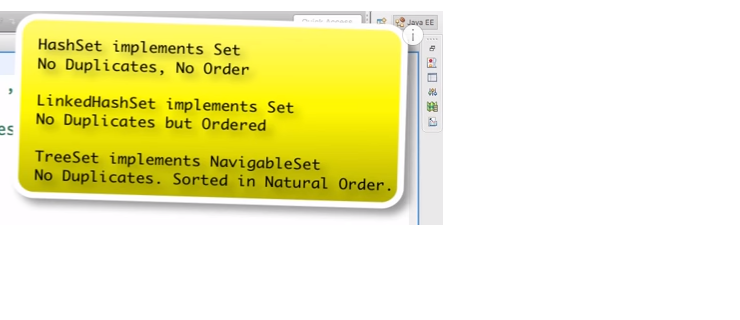


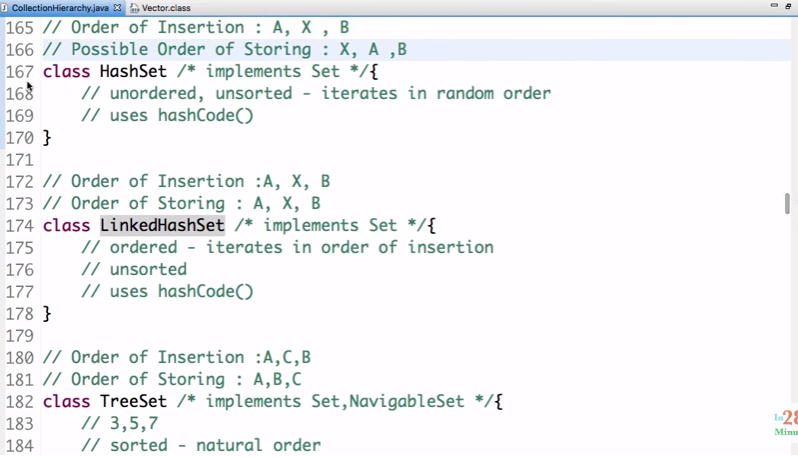
**QUEUE METHODS**

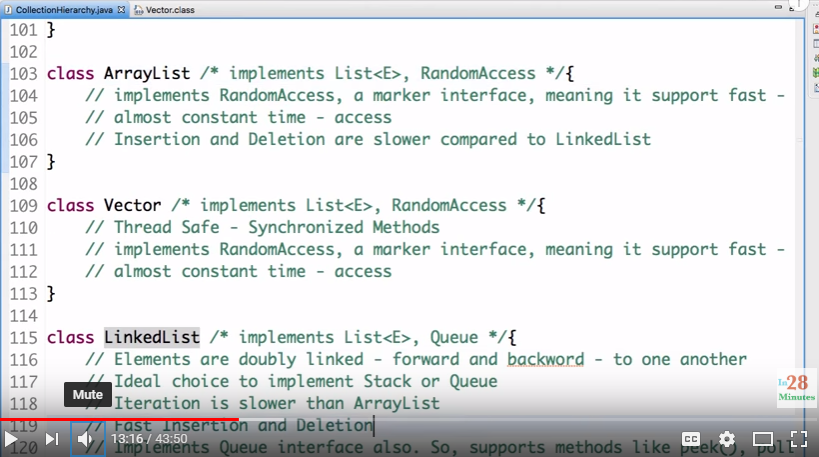
* The peek() method retrieves the value of the first element of the queue**without removing**it from the queue. For each invocation of the method we always get the same value and its execution  
  does not affect the size of the queue. **If the queue is empty the peek() method returns null.**
* The element() method behaves like peek(), so it again retrieves the value of the first element **without removing it.**Unlike peek ), however, **if the list is empty element() throws a NoSuchElementException**
* The poll() method retrieves the value of the first element of the queue **by removing it from the queue.**. At each invocation it removes the first element of the list and if the list is already empty **it returns null but does not throw any exception**
* The remove() method behaves as the poll() method, so it **removes the first element**of the list and **if the list is empty it throws a NoSuchElementException**

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









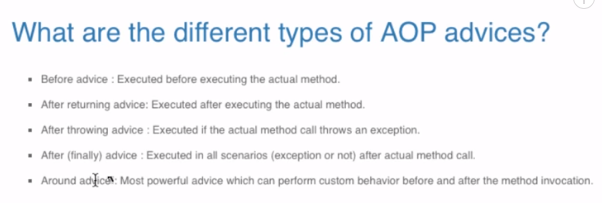
**SPRING ADVANTAGES**

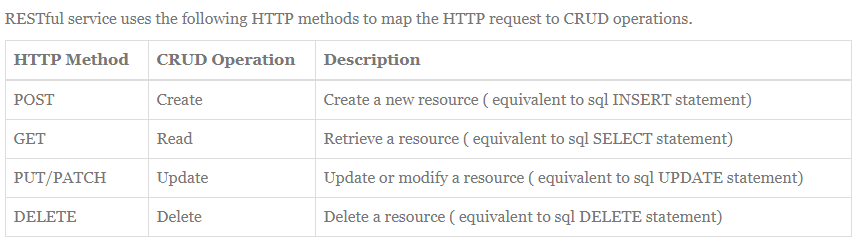
* **Dependency injection**
* **Integration with other frameworks like Hibernate**
* **Flexibility – Spring has inbuilt AOP Framework but doesn’t force you to use it**

**AOP:** Used to execute cross-cutting concerns – logging, declaration, security, transactions, etc.

PointCut – It is an expression that is used to identify the methods that are to be intercepted.

Aspect – What is the functionality/concern that we want to implement i.e; logging code





**MULTITHREADING** - in28Minutes - <https://www.youtube.com/watch?v=AfVbJDr-8ic&t=1298s>

**COLLECTIONS** - in28Minutes - <https://www.youtube.com/watch?v=3hgYHXDVh-o>

**JVM ARCHITECTURE** - <https://www.youtube.com/watch?v=dncpVFP1JeQ>

**DATA STRUCTURES** – mycodeschool (all 42 videos) – 1st link - <https://www.youtube.com/watch?v=92S4zgXN17o&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P>

<http://careerdrill.com/blog/coding-interview/choosing-the-right-data-structure-to-solve-problems/>

**SPRING MVC with RestFUL WEBSERVICES and HIBERNATE -**

<http://www.javainterviewpoint.com/spring-restful-web-services-crud-example/>