**---------------------------------------------------------------------------------------------------------------------**

**---->How to avoid deadlock**

* **don't use multiple threads (like Swing does, for example, by mandating that everything is done in the EDT)**
* **don't hold several locks at once. If you do, always acquire the locks in the same order**
* **don't execute foreign code while holding a lock**
* **use interruptible locks**

**————————————————————————————————————————-**

**when git has independent project and local has indempendent**

**to pull chenges that are not in your local**

**git pull origin branchname --allow-unrelated-histories**

**\**

**to pull without commit**

**git stash**

**git pull**

**git stash pop**

**To apply a stash and remove it from the stash list, run:**

**git stash pop stash@{n}**

**To apply a stash and keep it in the stash cache, run:**

**git stash apply stash@{n}**

**When switching branches, proceed even if the index or the working tree differs from HEAD. This is used to throw away local changes.**

**git checkout -f branch**

**t visualize all branch**

**gitk --all**

**//USE REVERT TO ADD NEW COMMIT IN END WHEN PUSH IS DONE**

**//USE RESET IF NOT PUSHED AND WANT TO GO BCK IN GIT CHAIN**

**git log $ git reflog**

**# This will detach your HEAD, that is, leave you with no branch checked out:**

**git checkout 0d1d7fc32**

**//REVERT UNPUSHED COMMITS**

**# This will destroy any local modifications.**

**# Don't do it if you have uncommitted work you want to keep.**

**git reset --hard 0d1d7fc32**

**# Alternatively, if there's work to keep:**

**git stash**

**git reset --hard 0d1d7fc32**

**git stash pop**

**# This saves the modifications, then reapplies that patch after resetting.**

**# You could get merge conflicts, if you've modified things which were**

**# changed since the commit you reset to.**

**//REVERT PUSHED CHANGES**

**Undo published commits with new commits**

**On the other hand, if you've published the work, you probably don't want to reset the branch, since that's effectively rewriting history. In that case, you could indeed revert the commits. With Git, revert has a very specific meaning: create a commit with the reverse patch to cancel it out. This way you don't rewrite any history.**

**# This will create three separate revert commits:**

**git revert a867b4af 25eee4ca 0766c053**

**# It also takes ranges. This will revert the last two commits:**

**git revert HEAD~2..HEAD**

**#Similarly, you can revert a range of commits using commit hashes:**

**git revert a867b4af..0766c053**

**# Reverting a merge commit**

**git revert -m 1 <merge\_commit\_sha>**

**# To get just one, you could use `rebase -i` to squash them afterwards**

**# Or, you could do it manually (be sure to do this at top level of the repo)**

**# get your index and work tree into the desired state, without changing HEAD:**

**git checkout 0d1d7fc32 .**

**# Then commit. Be sure and write a good message describing what you just did**

**git commit**

**//Reverting Working Copy to Most Recent Commit**

**To revert to a previous commit, ignoring any changes:**

**git reset --hard HEAD**

**where HEAD is the last commit in your current branch**

**Reverting The Working Copy to an Older Commit**

**To revert to a commit that's older than the most recent commit:**

**# Resets index to former commit; replace '56e05fced' with your commit code**

**git reset 56e05fced**

**# Moves pointer back to previous HEAD**

**git reset --soft HEAD@{1}**

**git commit -m "Revert to 56e05fced"**

**# Updates working copy to reflect the new commit**

**git reset --hard**

**git reset --soft HEAD~1**

**--soft indicates that the uncommitted files should be retained as working files opposed to**

**--hard which would discard them.**

**HEAD~1 is the last commit. If you want to rollback 3 commits you could use HEAD~3.**

**If you want to rollback to a specific revision number, you could also do that using its SHA hash.**

**git stash**

**It pushes changes to a stack. When you want to pull them back use**

**git stash apply**

**You can even pull individual items out. To completely blow away the stash:**

**git stash clear**

**------------------------------------------------------------------------------------------**

**--->How ANnotation work in java**

**The first main distinction between kinds of annotation is whether they're used at compile time and then discarded (like @Override) or placed in the compiled class file and available at runtime (like Spring's @Component). This is determined by the** [**@Retention**](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Retention.html) **policy of the annotation. If you're writing your own annotation, you'd need to decide whether the annotation is helpful at runtime (for autoconfiguration, perhaps) or only at compile time (for checking or code generation).**

**When compiling code with annotations, the compiler sees the annotation just like it sees other modifiers on source elements, like access modifiers (public/private) or final. When it encounters an annotation, it runs an annotation processor, which is like a plug-in class that says it's interested a specific annotation. The annotation processor generally uses the Reflection API to inspect the elements being compiled and may simply run checks on them, modify them, or generate new code to be compiled. @Override is an example of the first; it uses the Reflection API to make sure it can find a match for the method signature in one of the superclasses and uses the Messager to cause a compile error if it can't.**

**There are a number of tutorials available on writing annotation processors;** [**here's a useful one**](http://travisdazell.blogspot.com/2012/10/writing-annotation-based-processor-in.html)**. Look through the methods on** [**the Processor interface**](http://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html) **for how the compiler invokes an annotation processor; the main operation takes place in the process method, which gets called every time the compiler sees an element that has a matching annotation.**

**Refer TestCustomAnnotation.java**

**----------------------------------------------------------------------------------------**

**--->How can you handle null pointer when iterator encounters a null key while iteration over a hashmap. assume my developer has inserted a null key by mistake**

**y answer: Just we need to check null!=entry.getKey(). he was not satisfied and said before that how will you handle.**

**How should I answer that question. When I was back.**

**I tried this**

**public class Main { public static void main(String[] args) { Map<String,String> map = new HashMap<String,String>();**

**map.put(null, "null");**

**map.put("null",null);**

**map.put("test", "test");**

**Iterator<Entry<String, String>> it = map.entrySet().iterator();**

**while(it.hasNext())**

**{**

**System.out.println(it.next().getKey());**

**} } }**

**output:**

**null test null**

**ap.remove(null)**

**From JavaDocs:**

**Returns the value to which this map previously associated the key, or null if the map contained no mapping for the key.**

**If this map permits null values, then a return value of null does not necessarily indicate that the map contained no mapping for the key; it's also possible that the map explicitly mapped the key to null.**

**The map will not contain a mapping for the specified key once the call returns.**

**Call it before iteration, and it will remove explicit null keys**

**If myMap doesn't contain a key that matches c, then myMap.get(c) will return null. In that case, when the JVM unboxes what it expects to be a java.lang.Boolean object into a boolean primitive to execute the condition, it founds a null object and therefore throws a java.lang.NullPointerException.**

**The following block is equivalent to what you have in your example and should make it easier to understand why you would have a NullPointerException:**

**if (((Boolean) myMap.get(c)).booleanValue())**

**I would re-write your original condition as:**

**if ( myMap.containsKey(c) )**

**--------------------------------------------------------------------------------------**

**------>HOW to find active session**

**Here’s a simple “HttpSessionListener” example to keep track the total number of active sessions in a web application. If you want to keep monitor your session’s create and remove behavior, then consider this listener.**

## Java Source

**package com.mkyong;**

**import javax.servlet.http.HttpSessionEvent;**

**import javax.servlet.http.HttpSessionListener;**

**public class SessionCounterListener implements HttpSessionListener {**

**private static int totalActiveSessions;**

**public static int getTotalActiveSession(){**

**return totalActiveSessions;**

**}**

**@Override**

**public void sessionCreated(HttpSessionEvent arg0) {**

**totalActiveSessions++;**

**System.out.println("sessionCreated - add one session into counter");**

**}**

**@Override**

**public void sessionDestroyed(HttpSessionEvent arg0) {**

**totalActiveSessions--;**

**System.out.println("sessionDestroyed - deduct one session from counter");**

**}**

**}**

**Copy**

## web.xml

**<web-app ...>**

**<listener>**

**<listener-class>com.mkyong.SessionCounterListener</listener-class>**

**</listener>**

**</web-app>**

**Copy**

## How it work?

**– If a new session is created , e.g “request.getSession();” , the listener’s sessionCreated() will be executed.**

**– If a session is destroyed, e.g session’s timeout or “session.invalidate()”, the listener’s sessionDestroyed() will be executed.**

**HttpSession session = request.getSession(); //sessionCreated() is executed**

**session.setAttribute("url", "mkyong.com");**

**session.invalidate(); //sessionDestroyed() is executed**

**-----------------------------------------------------------------------------------------------------------**

**-------> Why public private protected variable cant be declared inside method**

**{**

**private int sum; ILLEGAL MODIFIER FOR PARAMETER SUM; ONLY FINAL IS PERMITTED**

**private int i; ILLEGAL MODIFIER FOR PARAMETER I; ONLY FINAL IS PERMITTED**

**sum = 0;**

**i = 0;**

**while (i >= 10)**

**{**

**sum = sum + i;**

**i++;**

**}**

**}**

**The modifiers private, protected, and public cannot be used on variables inside of a method. This is because you can only have local variables inside of methods.**

**Java is simply telling you that the only modifier allowed at that time is the final keyword.**

**\*/**

**--------------------------------------------------------------------------------------**

**--------->Why static variable cant be a local variable**

**/\* declaring variable as final inside method is fine , but static inside**

**\* method mean you are trying to declare**

**\* local variable as class level memebr which is conflict.**

**\* Also class level members are given meemory at class loading hence if locallly defined static it will not get initialized and**

**\* throws error (compilation)**

**\* \*--------------------------------------------------------------------------------------**

**------->Why wait , notify in object class but sleep in thread class**

**For better understanding why wait() and notify() method belongs to Object class, I'll give you a real life example: Suppose a gas station has a single toilet, the key for which is kept at the service desk. The toilet is a shared resource for passing motorists. To use this shared resource the prospective user must acquire a key to the lock on the toilet. The user goes to the service desk and acquires the key, opens the door, locks it from the inside and uses the facilities.**

**Meanwhile, if a second prospective user arrives at the gas station he finds the toilet locked and therefore unavailable to him. He goes to the service desk but the key is not there because it is in the hands of the current user. When the current user finishes, he unlocks the door and returns the key to the service desk. He does not bother about waiting customers. The service desk gives the key to the waiting customer. If more than one prospective user turns up while the toilet is locked, they must form a queue waiting for the key to the lock. Each thread has no idea who is in the toilet.**

**Obviously in applying this analogy to Java, a Java thread is a user and the toilet is a block of code which the thread wishes to execute. Java provides a way to lock the code for a thread which is currently executing it using the synchronized keyword, and making other threads that wish to use it wait until the first thread is finished. These other threads are placed in the waiting state. Java is NOT AS FAIR as the service station because there is no queue for waiting threads. Any one of the waiting threads may get the monitor next, regardless of the order they asked for it. The only guarantee is that all threads will get to use the monitored code sooner or later.**

**Finally the answer to your question: the lock could be the key object or the service desk. None of which is a Thread.**

**However, these are the objects that currently decide whether the toilet is locked or open. These are the objects that are in a position to notify that the bathroom is open (“notify”) or ask people to wait when it is locked wait.**

If wait() and notify() were on the Thread instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call thread2.notify() it would have to somehow find out that thread2 was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.

In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the *object* that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is thread2 which is waiting for the resource – they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.

In Java, we then use lock objects as synchronization, mutex, and communication points between threads. We synchronize on a lock object to get mutex access to an important code block and to synchronize memory. We wait on an object if we are waiting for some condition to change – some resource to become available. We notify on an object if we want to awaken sleeping threads.

// locks should be final objects so the object instance we are synchronizing on, // never changes private final Object lock = new Object(); ... // ensure that the thread has a mutex lock on some key code synchronized (lock) { ... // i need to wait for other threads to finish with some resource // this releases the lock and waits on the associated monitor lock.wait(); ... // i need to signal another thread that some state has changed and they can // awake and continue to run lock.notify(); }

---------------------------------------------------------------------------------------------------------

**------>How will you convert a list into map using streams in java 8? The list is a list of employees, the employee object contains id and name?**

**public static Map<Integer,String> listToHashmapJava8(List<Student> students)**

**{**

**Map<Integer, String> studentsMap = students.stream().collect(Collectors.toMap(Student :: getId, Student :: getName));**

**return studentsMap;**

**}**

**What are Customized Excpetions explain with a real world example?**

**Explain Producer Consumer Example?**

**Explain Internal Working of Hashmap?**

**----->What is a thread local variable in Java? (**[**answer**](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html)**)**

Thread-local variables are variables confined to a thread, its like thread's own copy which is not shared between multiple threads. Java provides a ThreadLocal class to support thread-local variables. It's one of the many ways to achieve thread-safety. Though be careful while using thread local variable in managed environment e.g. with web servers where worker thread out lives any application variable. Any thread local variable which is not removed once its work is done can potentially cause a memory leak in Java application.

**Thread Local is an interesting and useful concept, yet most of the Java developers are not aware of how to use that. In this post, I’ll explain what is Thread Local and when to use it, with an example code.**

**Since it’ll be little tough to understand this concept at first, I’ll keep the explanation as simple as possible (corollary: you shouldn’t use this code as it is in a production environment. Grasp the concept and improve upon it!)**

**Let’s begin.**

## What is Thread Local?

**Thread Local can be considered as a scope of access, like a *request scope* or *session scope*. It’s a *thread scope.* You can set any object in Thread Local and this object will be *global* and *local* to the specific thread which is accessing this object. Global and local!!? Let me explain:**

* **Values stored in Thread Local are *global* to the thread, meaning that they can be accessed from anywhere inside that thread. If a thread calls methods from several classes, then all the methods can see the Thread Local variable set by other methods (because they are executing in same thread). The value need not be passed explicitly. It’s like how you use global variables.**
* **Values stored in Thread Local are *local* to the thread, meaning that each thread will have it’s own Thread Local variable. One thread can not access/modify other thread’s Thread Local variables.**

**Well, that’s the concept of Thread Local. I hope you understood it (if not, please leave a comment).**

## When to Use Thread Local?

**We saw what is thread local in the above section. Now let’s talk about the use cases. i.e. when you’ll be needing something like Thread Local.**

**I can point out one use case where I used thread local. Consider you have a Servlet which calls some business methods. You have a requirement to generate a unique transaction id for each and every request this servlet process and you need to pass this transaction id to the business methods, for logging purpose. One solution would be passing this transaction id as a parameter to all the business methods. But this is not a good solution as the code is redundant and unnecessary.**

**To solve that, you can use Thread Local. You can generate a transaction id (either in servlet or better in a filter) and set it in the Thread Local. After this, what ever the business method, that this servlet calls, can access the transaction id from the thread local.**

**This servlet might be servicing more that one request at a time. Since each request is processed in separate thread, the transaction id will be unique to each thread (local) and will be accessible from all over the thread’s execution (global).**

**Got it!?**

## How to Use Thread Local?

**Java provides an** [**ThreadLocal**](http://download.oracle.com/javase/6/docs/api/java/lang/ThreadLocal.html) **object using which you can set/get *thread scoped* variables. Below is a code example demonstrating what I’d explained above.**

**Lets first have the Context.java file which will hold the transactionId field.**

**package com.veerasundar;public class Context {private String transactionId = null; /\* getters and setters here \*/}**

**Now create the MyThreadLocal.java file which will act as a container to hold our context object.**

**package com.veerasundar;/\*\* \* this class acts as a container to our thread local variables. \* @author vsundar \* \*/public class MyThreadLocal {public static final ThreadLocal userThreadLocal = new ThreadLocal();public static void set(Context user) {userThreadLocal.set(user);}public static void unset() {userThreadLocal.remove();}public static Context get() {return userThreadLocal.get();}}**

**In the above code, you are creating a ThreadLocal object as a static field which can be used by rest of the code to set/get thread local variables.**

**Let’s create our main class file which will generate and set the transaction ID in thread local and then call the business method.**

**package com.veerasundar;**

**public class ThreadLocalDemo extends Thread**

**{public static void main(String args[])**

**{Thread threadOne = new ThreadLocalDemo();**

**threadOne.start();**

**Thread threadTwo = new ThreadLocalDemo();threadTwo.start();}@Overridepublic void run() {// sample code to simulate transaction idContext context = new Context();**

**context.setTransactionId(getName());// set the context object in thread local to access it somewhere else**

**MyThreadLocal.set(context);/\***

**note that we are not explicitly passing the transaction id \*/new BusinessService().businessMethod();**

**MyThreadLocal.unset();}}**

**Finally, here’s the code for the BusinessService.java which will read from thread local and use the value.**

**package com.veerasundar;public class BusinessService {public void businessMethod() {// get the context from thread localContext context = MyThreadLocal.get();System.out.println(context.getTransactionId());}}**

**When you run the ThreadLocalDemo file, you’ll get the below output:**

**Thread-0Thread-1**

**As you might see, even though we are not explicitly passing the transaction id, the value can be accessed from the business method and printed on the console. Adding to it, the transaction ID differs for each thread (0 and 1).**

**Well, that’s it. I hope I’d explained it in a simple possible way. Please let me know what do you think about this article in comments. Do leave a comment if you want to add anything to this topic**

**----------------------------------------------------------------------------------------------------------------**

**Producer consumer problem**

# Producer Consumer Problem with Wait and Notify - Thread Example

**Producer Consumer Problem is a classical concurrency problem and in fact it is one of the concurrency design pattern. In last article we have seen solving** [**Producer Consumer problem in Java using blocking Queue**](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html)**but one of my reader emailed me and requested code example and explanation of solving Producer Consumer problem in Java with** [**wait and notify method**](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) **as well, Since its often asked as one of the top** [**coding question in Java**](http://java67.blogspot.sg/2012/08/10-java-coding-interview-questions-and.html)**. In this Java tutorial, I have put the code example of wait notify version of earlier producer consumer concurrency design pattern. You can see this is much longer code with explicit handling blocking conditions like when shared queue is full and when queue is empty. Since we have replaced** [**BlockingQueue**](http://javarevisited.blogspot.sg/2012/12/blocking-queue-in-java-example-ArrayBlockingQueue-LinkedBlockingQueue.html) **with Vector we need to implement blocking using** [**wait and notify**](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html) **and that's why we have introduced produce(int i) and consume() method. If you see I have kept consumer thread little slow by allowing it to sleep for 50 Milli second to give an opportunity to producer to fill the queue, which helps to understand that Producer thread is also waiting when Queue is full.**

## Java program to solve Producer Consumer Problem in Java

How to solve Producer Consumer Problem in Java with Example**Here is complete Java program to solve producer consumer problem in Java programming language. In this program we have used wait and notify method from java.lang.Object class instead of using BlockingQueue for flow control.**

**import java.util.Vector;**

**import java.util.logging.Level;**

**import java.util.logging.Logger;**

**/\*\***

**\* Java program to solve Producer Consumer problem using wait and notify**

**\* method in Java. Producer Consumer is also a popular concurrency design pattern.**

**\***

**\* @author Javin Paul**

**\*/**

**public class ProducerConsumerSolution {**

**public static void main(String args[]) {**

**Vector sharedQueue = new Vector();**

**int size = 4;**

**Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");**

**Thread consThread = new Thread(new Consumer(sharedQueue, size), "Consumer");**

**prodThread.start();**

**consThread.start();**

**}**

**}**

**class Producer implements Runnable {**

**private final Vector sharedQueue;**

**private final int SIZE;**

**public Producer(Vector sharedQueue, int size) {**

**this.sharedQueue = sharedQueue;**

**this.SIZE = size;**

**}**

**@Override**

**public void run() {**

**for (int i = 0; i < 7; i++) {**

**System.out.println("Produced: " + i);**

**try {**

**produce(i);**

**} catch (InterruptedException ex) {**

**Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);**

**}**

**}**

**}**

**private void produce(int i) throws InterruptedException {**

***//wait if queue is full***

**while (sharedQueue.size() == SIZE) {**

**synchronized (sharedQueue) {**

**System.out.println("Queue is full " + Thread.currentThread().getName()**

**+ " is waiting , size: " + sharedQueue.size());**

**sharedQueue.wait();**

**}**

**}**

***//producing element and notify consumers***

**synchronized (sharedQueue) {**

**sharedQueue.add(i);**

**sharedQueue.notifyAll();**

**}**

**}**

**}**

**class Consumer implements Runnable {**

**private final Vector sharedQueue;**

**private final int SIZE;**

**public Consumer(Vector sharedQueue, int size) {**

**this.sharedQueue = sharedQueue;**

**this.SIZE = size;**

**}**

**@Override**

**public void run() {**

**while (true) {**

**try {**

**System.out.println("Consumed: " + consume());**

**Thread.sleep(50);**

**} catch (InterruptedException ex) {**

**Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);**

**}**

**}**

**}**

**private int consume() throws InterruptedException {**

***//wait if queue is empty***

**while (sharedQueue.isEmpty()) {**

**synchronized (sharedQueue) {**

**System.out.println("Queue is empty " + Thread.currentThread().getName()**

**+ " is waiting , size: " + sharedQueue.size());**

**sharedQueue.wait();**

**}**

**}**

***//Otherwise consume element and notify waiting producer***

**synchronized (sharedQueue) {**

**sharedQueue.notifyAll();**

**return (Integer) sharedQueue.remove(0);**

**}**

**}**

**}**

**Output:**

**Produced: 0**

**Queue is empty Consumer is waiting , size: 0**

**Produced: 1**

**Consumed: 0**

**Produced: 2**

**Produced: 3**

**Produced: 4**

**Produced: 5**

**Queue is full Producer is waiting , size: 4**

**Consumed: 1**

**Produced: 6**

**Queue is full Producer is waiting , size: 4**

**Consumed: 2**

**Consumed: 3**

**Consumed: 4**

**Consumed: 5**

**Consumed: 6**

**Queue is empty Consumer is waiting , size: 0**

**That’s all on How to solve producer consumer problem in Java using wait and notify method. I still think that using BlockingQueueto implement producer consumer design pattern is much better because of its simplicity and concise code. At the same time this problem is an excellent exercise to understand concept of wait and notify method in Java.**

**------------------------------------------------------------------------------------------------**

**---------------->Can volatile make a non-atomic operation to atomic?(volatile vs Atomic)**

**---->Volatile and Atomic are two different concepts. Volatile ensures, that a certain, expected (memory) state is true across different threads, while Atomics ensure that operation on variables are performed atomically.**

**Take the following example of two threads in Java:**

**Thread A:**

**value = 1; done = true;**

**Thread B:**

**if (done) System.out.println(value);**

**Starting with value = 0 and done = false the rule of threading tells us, that it is undefined whether or not Thread B will print value. Furthermore *value* is undefined at that point as well! To explain this you need to know a bit about Java memory management (which can be complex), in short: Threads may create local copies of variables, and the JVM can reorder code to optimize it, therefore there is no guarantee that the above code is run in exactly that order. Setting done to true and thensetting value to 1 could be a possible outcome of the JIT optimizations.**

**volatile only ensures, that at the moment of access of such a variable, the new value will be immediately visible to all other threads and the order of execution ensures, that the code is at the state you would expect it to be. So in case of the code above, defining done as *volatile* will ensure that whenever Thread B checks the variable, it is either false, or true, and if it is true, then value has been set to 1 as well.**

**As a side-effect of *volatile*, the value of such a variable is set thread-wide atomically (at a very minor cost of execution speed). This is however only important on 32-bit systems that i.E. use long (64-bit) variables (or similar), in most other cases setting/reading a variable is atomic anyways. But there is an important difference between an atomic access and an atomic operation. Volatile only ensures that the access is atomically, while Atomics ensure that the *operation* is atomically.**

**Take the following example:**

**i = i + 1;**

**No matter how you define i, a different Thread reading the value just when the above line is executed might get i, or i + 1, because the *operation* is not atomically. If the other thread sets i to a different value, in worst case i could be set back to whatever it was before by thread A, because it was just in the middle of calculating i + 1 based on the old value, and then set i again to that old value + 1. Explanation:**

**Assume i = 0 Thread A reads i, calculates i+1, which is 1 Thread B sets i to 1000 and returns Thread A now sets i to the result of the operation, which is i = 1**

**Atomics like AtomicInteger ensure, that such operations happen atomically. So the above issue cannot happen, i would either be 1000 or 1001 once both threads are finished.**

**/\***

**The effect of the volatile keyword is approximately that each individual read or write operation on that variable is atomic.**

**Notably, however, an operation that requires more than one read/write -- such as i++, which is equivalent to i = i + 1, which does one read and one write -- is *not* atomic, since another thread may write to i between the read and the write.**

**The Atomic classes, like AtomicInteger and AtomicReference, provide a wider variety of operations atomically, specifically including increment for AtomicInteger**

**/\***

**------->About volatile**

**the volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from** [**main memory**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)**. So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in an int or a boolean variable then you can declare them as volatile variable.**

**From Java 5 along with major changes like Autoboxing, Enum, Generics and Variable arguments , Java introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made from one thread to another also as "happens-before" which solves the problem of memory writes that happen in one thread can "leak through" and be seen by another thread.**

**The Java volatile keyword cannot be used with method or class and it can only be used with a variable. Java volatile keyword also guarantees visibility and ordering, after Java 5 write to any volatile variable happens before any read into the volatile variable. By the way use of volatile keyword also prevents compiler or JVM from the reordering of code or moving away them from synchronization barrier.**

## The Volatile variable Example in Java

**To Understand example of volatile keyword in java let’s go back to** [**Singleton pattern in Java**](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) **and see** [**double checked locking in Singleton**](http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html) **with Volatile and without the volatile keyword in java.**

**/\*\***

**\* Java program to demonstrate where to use Volatile keyword in Java.**

**\* In this example Singleton Instance is declared as volatile variable to ensure**

**\* every thread see updated value for \_instance.**

**\***

**\* @author Javin Paul**

**\*/**

**public class Singleton{**

**private static volatile Singleton \_instance; //volatile variable**

**public static Singleton getInstance(){**

**if(\_instance == null){**

**synchronized(Singleton.class){**

**if(\_instance == null)**

**\_instance = new Singleton();**

**}**

**}**

**return \_instance;**

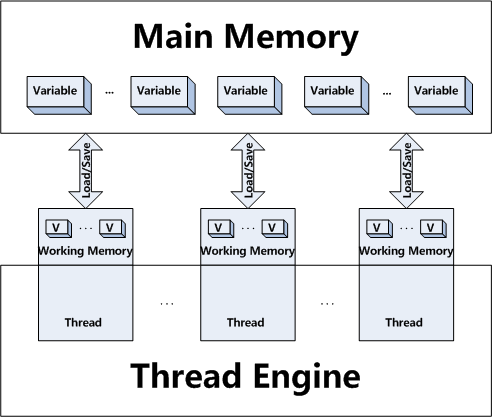
**}**

**If you look at the code carefully you will be able to figure out:**

**1) We are only creating instance one time**

**2) We are creating instance lazily at the time of the first request comes.**

**If we do not make the \_instance variable volatile than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of \_instance as not null and they will believe its still** [**null**](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html)**.**

****

**Why? because reader threads are not doing any locking and until writer thread comes out of synchronized block, memory will not be synchronized and value of \_instance will not be updated in main memory. With Volatile keyword in Java, this is handled by Java himself and such updates will be visible by all reader threads.**

**So in Summary apart from** [**synchronized keyword in Java**](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html)**, volatile keyword is also used to communicate the content of memory between threads.**

### Let’s see another example of volatile keyword in Java

**most of the time while writing game we use a variable bExit to check whether user has pressed exit button or not, value of this variable is updated in** [**event thread**](http://javarevisited.blogspot.sg/2011/09/invokeandwait-invokelater-swing-example.html) **and checked in game thread, So if we don't use volatile keyword with this variable, Game Thread might miss update from event handler thread if it's not synchronized in Java already. volatile keyword in java guarantees that value of the volatile variable will always be read from main memory and "*happens-before"* relationship in Java Memory model will ensure that content of memory will be communicated to different threads.**

**private boolean bExit;**

**while(!bExit) {**

**checkUserPosition();**

**updateUserPosition();**

**}**

**In this code example, One Thread (Game Thread) can cache the value of "bExit" instead of getting it from** [**main memory**](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html) **every time and if in between any other thread (Event handler Thread) changes the value; it would not be visible to this thread. Making boolean variable "bExit" as volatile in java ensures this will not happen.**

**Also, If you have not read already then I also suggest you read the topic about volatile variable from** [**Java Concurrency in Practice**](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) **book by Brian Goetz, one of the must read to truly understand this complex concept.**

## When to use Volatile variable in Java

**One of the most important thing in learning of volatile keyword is understanding when to use volatile variable in Java. Many** [**programmer**](http://javarevisited.blogspot.sg/2011/06/top-programming-interview-questions.html) **knows what is volatile variable and how does it work but they never really used volatile for any practical purpose. Here are couple of example to demonstrate when to use Volatile keyword in Java:**

**1) You can use Volatile variable if you want to read and write long and** [**double**](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) **variable atomically. long and double both are** [**64 bit**](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) **data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.**

**2) A volatile variable can be used as an alternative way of achieving** [**synchronization in Java**](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) **in some cases, like Visibility. with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.**

**3) volatile variable can be used to inform the compiler that a particular field is subject to be accessed by multiple threads, which will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment. Without volatile variable compiler can re-order the code, free to cache value of volatile variable instead of always reading from main memory. like following example without volatile variable may result in an [infinite loop](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html)**

**private boolean isActive = thread;**

**public void printMessage(){**

**while(isActive){**

**System.out.println("Thread is Active");**

**}**

**}**

**without the *volatile modifier*, it's not guaranteed that one** [**Thread**](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) **sees the updated value of isActive from other thread. The compiler is also free to cache value of isActive instead of reading it from main memory in every iteration. By making isActive a volatile variable you avoid these issue.**

**4) Another place where a volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in** [**Why should you use Enum as Singleton**](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) **that double checked locking was broken in Java 1.4 environment.**

### Important points on Volatile keyword in Java

**1. The volatile keyword in Java is only application to a variable and using volatile keyword with class and method is illegal.**

**2. volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.**

**3. In Java reads and writes are** [**atomic**](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) **for all variables declared using Java volatile keyword (including long and double variables).**

**4. Using the volatile keyword in Java on variables reduces the risk of memory consistency errors because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.**

**5. From Java 5 changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable in Java, it sees not just the** [**latest change to the volatile variable**](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html) **but also the side effects of the code that led up the change.**

**6. Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without the use of volatile keyword in Java.**

**7. An access to a volatile variable in Java never has a chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any** [**lock**](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html)**.**

**8. Java volatile variable that is an object reference may be null.**

**9. Java volatile keyword doesn't mean atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.**

**10. If a variable is not shared between multiple threads, you don't need to use volatile keyword with that variable.**

## Difference between synchronized and volatile keyword in Java

**What is the difference between volatile and synchronized is another popular** [**core Java question**](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html)**asked on multi-threading and concurrency interviews. Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.**

**1. The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.**

**2. Synchronized obtains and releases the lock on monitor’s Java volatile keyword doesn't require that.**

**3. Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the** [**volatile keyword**](http://java67.blogspot.com/2012/11/difference-between-transient-vs-volatile-modifier-variable-java.html) **in Java.**

**4. Synchronized method affects performance more than a volatile keyword in Java.**

**5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.**

**6. You can not synchronize on the null object but your volatile variable in Java could be null.**

**7. From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire**

**In short, volatile keyword in Java is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with use of synchronization in Java. If you like to know more about volatile I would also suggest going thorough FAQ on Java Memory Model here which explains happens-before operations quite well.**

**--------->Which one would be easy to write? synchronization code for 10 threads or 2 threads?**

**In terms of writing code, both will be of same complexity because synchronization code is independent of a number of threads. Choice of synchronization though depends upon a number of threads because the number of thread present more contention, so you go for advanced synchronization technique e.g. lock stripping, which requires more complex code and expertise**

**------->Can we make array volatile in Java?**

------>The volatile is a modifier in Java which only applies to member variables, both [instance and class variables](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html) and both [primitive and reference type](http://javarevisited.blogspot.sg/2015/09/difference-between-primitive-and-reference-variable-java.html). It provides the happens-before guarantee which ensures that a write to a volatile variable will happen before any reading. This ensures that any modification to volatile object or primitive type will be visible to all threads i.e. it provides the visibility guarantee.

The [volatile modifier](http://www.java67.com/2012/08/what-is-volatile-variable-in-java-when.html) also provides ordering guarantee because the compiler cannot re-order any code or operation which involves volatile variables (primitive and objects), but what is perhaps more important to know and remember is that volatile variable doesn't provide **atomicity** (except for write to the volatile double variable) and **mutual exclusion**, which is also the main difference between volatile and [synchronized](http://www.java67.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) keyword.

There are certain restrictions with volatile keyword e.g. you cannot make a member variable both [final](http://javarevisited.blogspot.sg/2016/09/21-java-final-modifier-keyword-interview-questions-answers.html) and volatile at the same time, but you can make a [static variable](http://www.java67.com/2016/05/difference-between-static-and-nonstatic-member-variable-in-java.html) volatile in Java.

If you want to learn more about the volatile variable in Java, I suggest reading [Java Concurrency in Practice](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20), which provides more thorough and complete introduction and application of volatile modifier in Java.

**Can we make array volatile in Java?**

Now, coming back to the original question, *can we make an array volatile in Java?* The answer is, Yes, you can make an array (both primitive and reference type array e.g. an [int array](http://www.java67.com/2015/07/array-concepts-interview-questions-answers-java.html) and [String array](http://www.java67.com/2012/09/java-program-to-convert-string-arraylist-to-string-array.html)) volatile in Java but only changes to reference pointing to an array will be visible to all threads, not the whole array. What this means is that suppose you have a reference variable called primes as shown below:

protected volatile int[] primes = new int[10];

then if you assign a new array to primes variable, change will be visible to all threads, but changes to individual indices will not be covered under volatile guarantee i.e.

primes = new int[20];

will follow the **"happens-before"** rule and cause memory barrier refresh, but following code will not do so

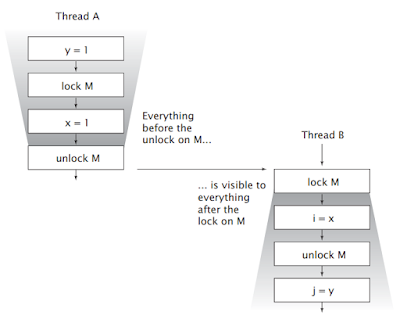
primes[0] = 10;

primes[1] = 20;

primes[2] = 30;

primes[3] = 40;

This means, if multiple threads are changing individual array elements e.g. storing updates, there won’t be any happens-before guarantee provided by the [volatile modifier](http://javarevisited.blogspot.com/2012/03/difference-between-transient-and.html) for such modification. So, if your use-case is to provide memory visibility guarantee for individual array elements than volatile is not the right choice. You must rely on other synchronization and a thread-safety mechanism to cover this case e.g. [synchronized](http://www.java67.com/2012/08/5-thread-interview-questions-answers-in.html) keyword, [atomic variables](http://javarevisited.blogspot.com//2011/07/java-multi-threading-interview.html), or [ReentrantLock](http://javarevisited.blogspot.sg/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html).



On a similar note, sometimes instead of an array, Interviewer put the collection i.e. they will ask *can you make a collection variable volatile in Java or not* e.g. an [ArrayList](http://www.java67.com/2015/06/20-java-arraylist-interview-questions.html) or [HashMap](http://javarevisited.blogspot.sg/2014/11/how-to-loop-hashmap-or-hashtable-in-jsp-example.html). The answer is same, of course, you can make a reference variable pointing to a Collection volatile in Java, but the happens-before guarantee will only be provided if the value of that reference variable is changed e.g. you assign a new collection to it.

Any modification done on actual collection object e.g. adding or removing elements from ArrayList will not invoke happens-before guarantee or memory barrier refresh

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

------------>why should wait always call in loop

-->You need not only to loop it but check your condition in the loop. Java does not guarantee that your thread will be woken up only by a notify()/notifyAll() call or the right notify()/notifyAll() call at all. Because of this property the loop-less version might work on your development environment and fail on the production environment unexpectedly.

For example, you are waiting for something:

synchronized (theObjectYouAreWaitingOn) { while (!carryOn) { theObjectYouAreWaitingOn.wait(); } }

An evil thread comes along and:

theObjectYouAreWaitingOn.notifyAll();

If the evil thread does not/can not mess with the carryOn you just continue to wait for the proper client.

**Edit:** Added some more samples. The wait can be interrupted. It throws InterruptedException and you might need to wrap the wait in a try-catch. Depending on your business needs, you can exit or suppress the exception and continue waiting.

===============================================================

---------------------->what is Recusrsion---

# Recursion

**What is Recursion?**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are [Towers of Hanoi (TOH)](http://quiz.geeksforgeeks.org/c-program-for-tower-of-hanoi/), [Inorder/Preorder/Postorder Tree Traversals](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/), [DFS of Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), etc.

**What is base condition in recursion?**

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

int fact(int n)

{

if (n < = 1) // base case

return 1;

else

return n\*fact(n-1);

}

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

**How a particular problem is solved using recursion?**

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

**Why Stack Overflow error occurs in recursion?**

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

int fact(int n)

{

// wrong base case (it may cause

// stack overflow).

if (n == 100)

return 1;

else

return n\*fact(n-1);

}

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

**What is the difference between direct and indirect recursion?**

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in Table 1.

**// An example of direct recursion**

void directRecFun()

{

// Some code....

directRecFun();

// Some code...

}

**// An example of indirect recursion**

void indirectRecFun1()

{

// Some code...

indirectRecFun2();

// Some code...

}

void indirectRecFun2()

{

// Some code...

indirectRecFun1();

// Some code...

}

**What is difference between tailed and non-tailed recursion?**

A recursive function is tail recursive when recursive call is the last thing executed by the function. Please refer [tail recursion article](https://www.geeksforgeeks.org/tail-recursion/) for details.

**How memory is allocated to different function calls in recursion?**

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Let us take the example how recursion works by taking a simple function.

* CPP
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

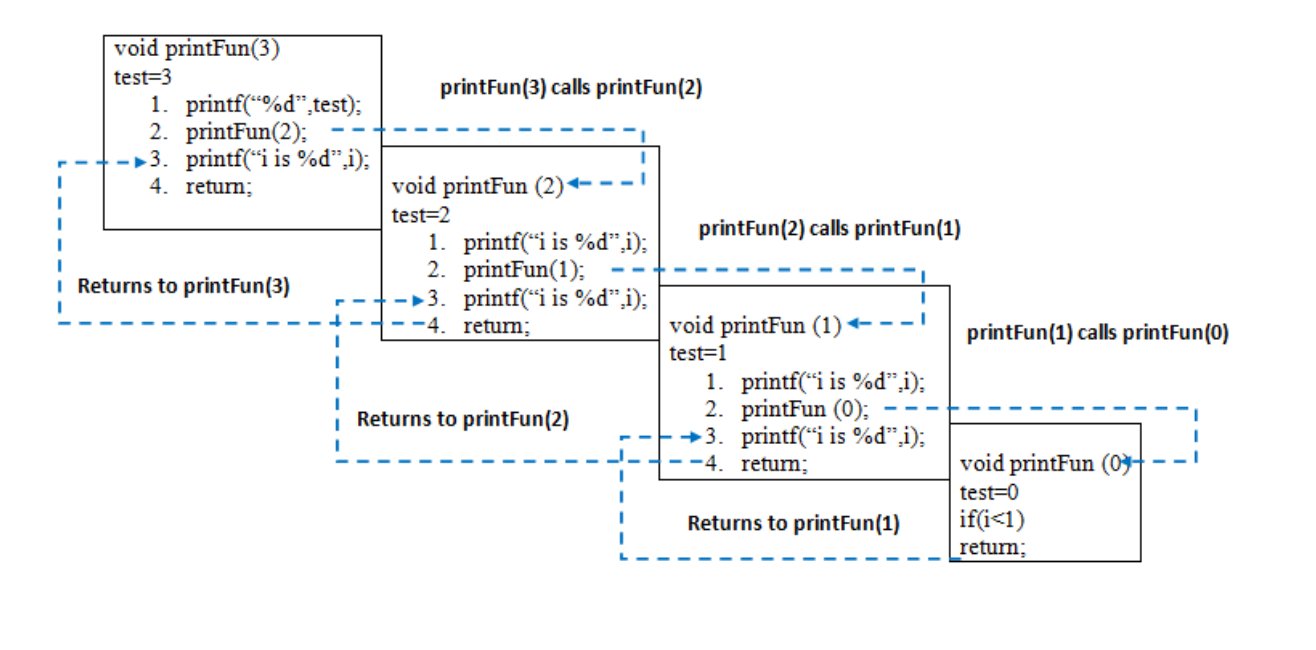
brightness\_4

|  |
| --- |
| // A Java program to demonstrate working of  // recursion  class GFG{  static void printFun(int test)  {  if (test < 1)  return;  else  {  System.out.printf("%d ",test);  printFun(test-1); // statement 2  System.out.printf("%d ",test);  return;  }  }    public static void main(String[] args)  {  int test = 3;  printFun(test);  }  }    // This code is contributed by  // Smitha Dinesh Semwal |

Output :

3 2 1 1 2 3

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



**What are the disadvantages of recursive programming over iterative programming?**

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

**What are the advantages of recursive programming over iterative programming?**

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, [Tower of Hanoi](https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/), etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer [Inorder Tree Traversal without Recursion](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/), [Iterative Tower of Hanoi](https://www.geeksforgeeks.org/iterative-tower-of-hanoi/).

----------------------------------------------------------------------------------------------------------

Manhattan

1. Reverse the linked list.

2. Find loop in linked list.

3. Find the hight of tree.

Easy Question

1. Swap to variables with /without using third variable.

2. Sort the given numbers with less time & space complexity.

Try to write programs without using builtin methods.

If they don't have any restrictions then go head .

& brush up usages of builtin methods &theory concepts..

-----------------------------------------------------------------------------------------------

JOIN:  
A join is used for displaying columns with the same or different names from different tables. The output displayed will have all the columns shown individually. That is, the columns will be aligned next to each other.  
  
UNION:  
The UNION set operator is used for combining data from two tables which have columns with the same datatype. When a UNION is performed the data from both tables will be collected in a single column having the same datatype.  
  
For example:  
  
See the two tables shown below:  
  
Table t1  
Articleno article price manufacturer\_id  
1 hammer 3 $ 1  
2 screwdriver 5 $ 2  
  
Table t2  
manufacturer\_id manufacturer  
1 ABC Gmbh  
2 DEF Co KG  
Now for performing a JOIN type the query is shown below.  
  
SELECT articleno, article, manufacturer  
FROM t1 JOIN t2 ON (t1.manufacturer\_id =  
t2.manufacturer\_id);  
  
articelno article manufacturer  
1 hammer ABC GmbH  
2 screwdriver DEF Co KG  
That is a join.  
  
UNION means that you have to tables or resultset with the same amount and type of columns and you add this to tables/resultsets together. Look at this example:  
  
Table year2006  
Articleno article price manufacturer\_id  
1 hammer 3 $ 1  
2 screwdriver 5 $ 2  
  
Table year2007  
Articleno article price manufacturer\_id  
1 hammer 6 $ 3  
2 screwdriver 7 $ 4  
  
SELECT articleno, article, price, manufactruer\_id  
FROM year2006  
UNION  
SELECT articleno, article, price, manufacturer\_id  
FROM year2007  
  
articleno article price manufacturer\_id  
1 hammer 3 $ 1  
2 screwdriver 5 $ 2  
1 hammer 6 $ 3  
2 screwdriver 7 $ 4//////////////////////////////////////////////////////////////////////////////

Spring boot actuator

Adding a custom health check is easy. Just create a new Java class, extend it from the AbstractHealthIndicator and implement the doHealthCheck method. The method gets a builder passed with some useful methods. Call builder.up() if your health is OK or builder.down() if it is not. What you do to check the health is completely up to you. Maybe you want to ping some server or check some files.  
  
@Component  
public class CustomHealthCheck extends AbstractHealthIndicator {  
 @Override  
 protected void doHealthCheck(Health.Builder bldr) throws Exception {  
 // TODO implement some check  
 boolean running = true;  
 if (running) {  
 bldr.up();  
 } else {  
 bldr.down();  
 }  
 }  
}  
This is enough to activate the new health check (make sure @ComponentScan is on your application). Restart your application and locate your browser to the /health endpoint and you will see the newly added health check.  
  
{  
 "status":"UP",  
 "CustomHealthCheck": {  
 "status":"UP"  
 },  
 "diskSpace": {  
 "status":"UP",  
 "free":56443746,  
 "threshold":1345660  
 }  
}

////////////////////////////////////////////////////////////////////////

GIT

-> gotot project directorydo git init

-> add files git add . if permission denied then git add --ignore-errors .

->git add .gitignore

->set up git account on bash

git config --global user.email "you@example.com"

git config --global user.name "Your Name"

->enter the commit message git commit -m “” or

Git commit opens vim then enter msg and write :wq

->git remote add origin [git@github.com](mailto:git@github.com):username/new\_repo to add remote git

<https://github.com/rish93/HotelApi>

-> to update remote url git remote set-url origin https://github.com/rish93/HotelApi

->git push -u origin master

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**SPRING SECURITY ROLE AND USER vs AUTO CONFIGURATION**

**Spring Boot Security Auto-Configuration**  
Last modified: February 23, 2018  
  
by baeldung SecuritySpring Spring Boot  
I just announced the new Spring 5 modules in REST With Spring:  
>> CHECK OUT THE COURSE  
  
1. Introduction  
In this article, we’ll have a look at Spring Boot’s opinionated approach to security.  
  
Simply put, we’re going to focus on the default security configuration and how we can disable or customize it if we need to.  
  
2. Default Security Setup  
In order to add security to our Spring Boot application, we need to add the security starter dependency:  
  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
This will include the SecurityAutoConfiguration class – containing the initial/default security configuration.  
  
Notice how we didn’t specify the version here, with the assumption that the project is already using Boot as the parent.  
  
Simply put, by default, the application will get Basic Authentication enabled. There are some predefined properties, such as:  
  
1  
2  
security.user.name=user  
security.basic.enabled=true  
If we start the application, we’ll notice that the default password is randomly generated and printed in the console log:  
  
1  
Using default security password: c8be15de-4488-4490-9dc6-fab3f91435c6  
For more defaults see the Spring Boot Common Application Properties at the security properties chapter.  
  
3. Disabling the Auto-Configuration  
To discard the security auto-configuration and add our own configuration, we need to exclude the SecurityAutoConfiguration class.  
  
  
@SpringBootApplication(exclude = { SecurityAutoConfiguration.class })  
public class SpringBootSecurityApplication {  
   
 public static void main(String[] args) {  
 SpringApplication.run(SpringBootSecurityApplication.class, args);  
 }  
}  
Or by adding some configuration into the application.properties file:  
  
1  
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration  
There are also some particular cases in which this setup isn’t quite enough.  
  
For example, almost each Spring Boot application is started with Actuator in the classpath. This causes problems because another auto-configuration class needs the one we’ve just excluded, so the application will fail to start.  
  
In order to fix this issue, we need to exclude that class; and, specific to the Actuator situation, we need to exclude ManagementWebSecurityAutoConfiguration.  
  
3.1. Disabling vs. Surpassing Security Auto-Configuration  
There’s a significant difference between disabling autoconfiguration and surpassing it.  
  
By disabling it, it’s just like adding the Spring Security dependency and the whole setup from scratch. This can be useful in several cases:  
  
Integrating application security with a custom security provider  
Migrating a legacy Spring application with already existing security setup – to Spring Boot  
But, most of the time we won’t need to fully disable the security auto-configuration.  
  
The way Spring Boot is configured permits surpassing the autoconfigured security by adding in our new/custom configuration classes. This is typically easier, as we’re just customizing an existing security setup to fulfill our needs.  
  
4. Configuring Spring Boot Security  
If we’ve chosen the path of disabling security auto-configuration, we naturally need to provide our own configuration.  
  
As we’ve discussed before, this is the default security configuration; we can customize it by modifying the property file.  
  
We can, for example, override the default password by adding our own:  
  
1  
security.user.password=password  
If we want a more flexible configuration, with multiple users and roles for example – you now need to make use of a full @Configuration class:  
  
  
@Configuration  
@EnableWebSecurity  
public class BasicConfiguration extends WebSecurityConfigurerAdapter {  
   
 @Override  
 protected void configure(AuthenticationManagerBuilder auth)  
 throws Exception {  
 auth  
 .inMemoryAuthentication()  
 .withUser("user")  
 .password("password")  
 .roles("USER")  
 .and()  
 .withUser("admin")  
 .password("admin")  
 .roles("USER", "ADMIN");  
 }  
   
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .anyRequest()  
 .authenticated()  
 .and()  
 .httpBasic();  
 }  
}  
The @EnableWebSecurity annotation is crucial if we disable the default security configuration.  
  
If missing, the application will fail to start. The annotation is only optional if we’re just overriding the default behavior using a WebSecurityConfigurerAdapter.  
  
Now, we should verify that our security configuration applies correctly by with a couple of quick live tests:  
  
  
@RunWith(SpringRunner.class)  
@SpringBootTest(webEnvironment = RANDOM\_PORT)  
public class BasicConfigurationIntegrationTest {  
   
 TestRestTemplate restTemplate;  
 URL base;  
 @LocalServerPort int port;  
   
 @Before  
 public void setUp() throws MalformedURLException {  
 restTemplate = new TestRestTemplate("user", "password");  
 base = new URL("http://localhost:" + port);  
 }  
   
 @Test  
 public void whenLoggedUserRequestsHomePage\_ThenSuccess()  
 throws IllegalStateException, IOException {  
 ResponseEntity<String> response   
 = restTemplate.getForEntity(base.toString(), String.class);  
   
 assertEquals(HttpStatus.OK, response.getStatusCode());  
 assertTrue(response  
 .getBody()  
 .contains("Baeldung"));  
 }  
   
 @Test  
 public void whenUserWithWrongCredentials\_thenUnauthorizedPage()   
 throws Exception {  
   
 restTemplate = new TestRestTemplate("user", "wrongpassword");  
 ResponseEntity<String> response   
 = restTemplate.getForEntity(base.toString(), String.class);  
   
 assertEquals(HttpStatus.UNAUTHORIZED, response.getStatusCode());  
 assertTrue(response  
 .getBody()  
 .contains("Unauthorized"));  
 }  
}  
The idea is that behind Spring Boot Security is, in fact, Spring Security, so any security configuration that can be done with this one, or any integration this one supports can be also implemented into Spring Boot.  
  
5. Spring Boot OAuth2 Auto-Configuration  
Spring Boot has a dedicated auto-configuration support for OAuth2.  
  
Before we get to that, let’s add the Maven dependency to start setting up our application:  
<dependency>  
 <groupId>org.springframework.security.oauth</groupId>  
 <artifactId>spring-security-oauth2</artifactId>  
</dependency>  
This dependency includes a set of classes that are capable of triggering the auto-configuration mechanism defined in OAuth2AutoConfiguration class.  
  
Now, we have multiple choices to continue, depending on the scope of our application.  
  
5.1. OAuth2 Authorization Server Auto-Configuration  
If we want our application to be an OAuth2 provider, we can use @EnableAuthorizationServer.  
  
On startup, we’ll notice in the logs that the auto-configuration classes will generate a client id and a client-secret for our authorization server and of course a random password for basic authentication.  
Using default security password: a81cb256-f243-40c0-a585-81ce1b952a98  
security.oauth2.client.client-id = 39d2835b-1f87-4a77-9798-e2975f36972e  
security.oauth2.client.client-secret = f1463f8b-0791-46fe-9269-521b86c55b71  
These credentials can be used to obtain an access token:  
  
1  
2  
3  
curl -X POST -u 39d2835b-1f87-4a77-9798-e2975f36972e:f1463f8b-0791-46fe-9269-521b86c55b71 \  
 -d grant\_type=client\_credentials -d username=user -d password=a81cb256-f243-40c0-a585-81ce1b952a98 \  
 -d scope=write http://localhost:8080/oauth/token  
5.2. Other Spring Boot OAuth2 Auto-Configuration Settings  
There are some other use cases covered by Spring Boot OAuth2 like:  
  
Resource Server – @EnableResourceServer  
Client Application – @EnableOAuth2Sso or @EnableOAuth2Client  
If we need our application to be one of the types above we just have to add some configuration to application properties.  
  
All OAuth2 specific properties can be found at Spring Boot Common Application Properties.  
  
6. Conclusion  
In this article, we focused on the default security configuration provided by Spring Boot. We saw how the security auto-configuration mechanism can be disabled or overridden and how a new security configuration can be applied.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**-----> Spring Boot and H2 in memory database - Why, What and How?**  
  
  
  
For example, with Oracle or mySQL databases, you would need to  
  
Install the Database  
Setup a Schema  
Setup the tables  
Populate the data  
Connect the application to the database by setting up a data source and a lot of other code  
Scenario 1 - Let’s consider a situation where you would want to do a quick POC. Using a traditional database involves a lot of overhead.  
  
Scenario 2 - Consider your unit tests  
  
You don’t want them to fail when some data/schema in the database changes  
You would want to be able to run them in parallel - multiple developers might be running the tests in parallel.  
In these kind of scenarios, an in memory database provides an ideal solution.  
  
An in memory database is created when an application starts up and destroyed when the application is stopped.  
  
Advantages  
  
Zero project setup or infrastructure  
Zero Configuration  
Zero Maintainance  
Easy to use for Learning, POCs and Unit Tests  
Spring Boot provides Simple Configuration to switch between a real database and an in memory database like H2  
H2  
H2 is one of the popular in memory databases. Spring Boot has very good integration for H2.  
  
From https://en.wikipedia.org/wiki/H2\_(DBMS)  
  
H2 is a relational database management system written in Java. It can be embedded in Java applications or run in the client-server mode.  
  
H2 supports a sub set of the SQL standard.  
  
H2 also provides a web console to maintain the database.  
  
Spring Boot and H2  
You need very little configuration to connect Spring Boot application with H2.  
  
In most situations, just adding the H2 runtime jar into dependencies should be sufficient.  
  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
</dependency>  
Setting up a Spring Boot Project with H2  
Spring Initializr http://start.spring.io/ is great tool to bootstrap your Spring Boot projects.  
  
Image  
  
As shown in the image above, following steps have to be done  
  
Launch Spring Initializr and choose the following  
Choose com.in28minutes.springboot.rest.example as Group  
Choose spring-boot-2-jpa-with-hibernate-and-h2 as Artifact  
Choose following dependencies  
Web  
JPA  
H2  
DevTools  
Click Generate Project.  
Import the project into Eclipse. File -> Import -> Existing Maven Project.  
Ensure that H2 is selected in the dependencies.  
  
Create a simple Student Entity with a primary key id.  
  
@Entity  
public class Student {  
 @Id  
 @GeneratedValue  
 private Long id;  
 private String name;  
 private String passportNumber;  
H2 - A Few Tips  
An in-memory database is live only during the time of execution of the application. It is an efficient way to learn a framework.  
This is not how you want your real world applications to behave.  
We explain how to connect to a database of your choice in the answer to the question “How do we connect to a external database?”.  
Spring Boot and H2 Magic  
H2 provides a web interface called H2 Console to see the data. Let’s enable h2 console in the application.properties.  
  
/src/main/resources/application.properties  
  
# Enabling H2 Console  
spring.h2.console.enabled=true  
When you start the application up now, you would see a lot of magic unfold!  
  
When you reload the application, you can launch up H2 Console at http://localhost:8080/h2-console.  
  
Image  
  
Tip - Make sure that you use jdbc:h2:mem:testdb as JDBC URL.  
  
You will see that a new table called ‘student’ is created in H2 Console.  
  
How did the Student table get created?  
  
Its because of Spring Boot Auto Configuration. We will talk about this in the next section.  
  
You can also populate some data into student table by adding a file called data.sql  
  
/src/main/resources/data.sql  
  
insert into student  
values(10001,'Ranga', 'E1234567');  
  
insert into student  
values(10002,'Ravi', 'A1234568');  
When you reload the application, you can launch up H2 Console to see that the student table is populated with the data from `data.sql’  
  
http://localhost:8080/h2-console.  
How did all the magic happen? Let’s look at it question by question in the next section.  
  
Frequently asked questions about Spring Boot, JPA, Hibernate and H2  
Q : How does H2 and Spring Boot combination work?  
First and most important thing - Spring Boot is intelligent.  
  
If you are talking to an in memory db, by default, it looks at the entities and creates the database and the tables.  
  
However, if you connect to a mysql database, Spring Boot knows that its a permanent database. By default, it expects you to set up the database, set up the tables and it uses the connection that you established.  
  
Q : How did the Spring Boot Application connect to the database H2?  
Its down to Spring Boot Auto Configuration!  
  
First thing you would need to understand is Spring Boot Auto Configuration.  
  
Here’s a good read  
  
http://www.springboottutorial.com/spring-boot-auto-configuration  
As far as H2 is concerned, as soon as Spring Boot sees H2 in the class path, it auto configures a data source similar to what you see below:  
  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=  
  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
It knows that you are using an inmemory database H2 and it uses the default url if you don’t provide one.  
  
Q : Where is the database connection info specified? How does it know to automatically connect to H2?  
Thats Spring Boot Autoconfiguration magic.  
  
From https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html  
  
Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, If HSQLDB is on your classpath, and you have not manually configured any database connection beans, then Spring Boot will auto-configure an in-memory database.  
  
Recommended Reading  
  
http://www.springboottutorial.com/spring-boot-auto-configuration  
Q : What happens if H2 is not in the classpath?  
You get this error  
  
Cannot determine embedded database driver class for database type NONE  
Add H2 to the pom.xml and Restart your server  
  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
</dependency>  
Q : Why is the data lost between restart?  
H2 is an in memory database. Its not a persisted database.  
  
H2 is a great tool for learning because you need zero setup.  
  
Error : Table is not created automatically in h2 embedded db or I’m unable to see the tables  
Usually, the table’s are created but the url used in H2 GUI Console is wrong.  
  
In the browser, change the database url to jdbc:h2:mem:testdb (Shown in the screen below).  
  
  
  
You should be good to go!  
  
Error : H2 Console is not Launched up?  
Try enabling it in the application.properties  
  
spring.h2.console.enabled=true  
Q : How did the insert query from data.sql run at application startup?  
That’s part of the Spring Boot startup routine. Any queries in data.sql are run at application startup. You can read more here.  
  
https://docs.spring.io/spring-boot/docs/current/reference/html/howto-database-initialization.html  
Running H2 as a persisted database with Spring Boot  
While we dont recommend this , it interesting to note that H2 has a persisted database mode  
  
With this configuration, the data is not lost even after spring boot restart and computer restart.  
You would find H2 being very rarely used in this way. If you are really interested in a persistent database, we recommend exploring MySQL, Oracle or some other relational database.  
  
application.properties  
  
spring.datasource.name=yourdbname  
spring.datasource.driverClassName=org.h2.Driver  
   
spring.datasource.initialize=false  
spring.datasource.url=jdbc:h2:file:~/yourdbname;DB\_CLOSE\_ON\_EXIT=FALSE;IFEXISTS=TRUE;DB\_CLOSE\_DELAY=-1;  
   
spring.jpa.hibernate.ddl-auto = update  
Using H2 for unit tests  
The standard properties file that Spring Boot picks up automatically when running an application is called application.properties and resides in the src/main/resources folder.  
  
If we want to use different properties for tests, then we can override the properties file in the main folder by placing another file with the same name in src/test/resources.  
  
The application.properties file in src/test/resources folder should contain the standard key-value pairs necessary for configuring a in memory connection.  
  
First add the dependencies for your database driver (mysql in the example below) and make the dependency for h2 test scoped.  
  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>  
  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>test</scope>  
</dependency>  
Use the mysql database for your real code  
  
src\main\resources\application.properties  
  
spring.jpa.hibernate.ddl-auto=none  
spring.datasource.url=jdbc:mysql://localhost:3306/person\_example  
spring.datasource.username=personuser  
spring.datasource.password=YOUR\_PASSWORD  
Use in memory database for your unit tests  
  
src\test\resources\application.properties  
  
spring.datasource.driver-class-name=org.h2.Driver  
spring.datasource.url=jdbc:h2:mem:testdb;DB\_CLOSE\_DELAY=-1  
spring.datasource.username=sa  
spring.datasource.password=sa

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

------------>JSON WEB TOKEN AUTHENTICATION FOR WEBSERVICE

Securing Restful APIs With JWTs  
JSON Web Tokens, commonly known as JWTs, are tokens that are used to authenticate users on applications. This technology has gained popularity over the past few years because it enables backends to accept requests simply by validating the contents of these JWTs. That is, applications that use JWTS no longer have to hold cookies or other session data about their users. This characteristic facilitates scalability while keeping applications secure.  
  
During the authentication process, when a user successfully logs in using their credentials, a JSON Web Token is returned and must be saved locally (typically in local storage). Whenever the user wants to access a protected route or resource (an endpoint), the user agent must send the JWT, usually in the Authorizationheader using the Bearer schema, along with the request.  
  
When a backend server receives a request with a JWT, the first thing to do is to validate the token. This consists of a series of steps, and if any of these fails then the request must be rejected. The following list shows the validation steps needed:  
  
Check that the JWT is well formed.  
Check the signature.  
Validate the standard claims.  
Check the Client permissions (scopes).  
We won't get into the nitty-gritty details about JWTS in this article but, if needed, this resource can provide more about information about JWTS and this resource about JWT validation.  
  
The Restful Spring Boot API Overview  
The RESTful Spring Boot API that we are going to secure is a task list manager. The task list is kept globally, which means that all users will see and interact with the same list. To clone and run this application, let's issue the following commands:  
  
**# clone the starter project  
git clone https://github.com/auth0-blog/spring-boot-auth.git  
cd spring-boot-auth  
# run the unsecured RESTful API  
gradle bootRun  
If everything works as expected, our RESTful Spring Boot API will be up and running. To test it, we can use a tool like Postman or curl to issue a request to the available endpoints:  
  
# issue a GET request to see the (empty) list of tasks  
curl http://localhost:8080/tasks**  
**# issue a POST request to create a new task  
curl -H "Content-Type: application/json" -X POST -d '{  
 "description": "Buy some milk(shake)"  
}' http://localhost:8080/tasks  
# issue a PUT request to update the recently created task  
curl -H "Content-Type: application/json" -X PUT -d '{  
 "description": "Buy some milk"  
}' http://localhost:8080/tasks/1  
# issue a DELETE request to remove the existing task  
curl -X DELETE http://localhost:8080/tasks/1**  
All the endpoints used in the commands above are defined in the TaskController class, which belongs to the com.auth0.samples.authapi.task package. Besides this class, this package contains two other classes:  
  
Task: the entity model that represents tasks in the application.  
TaskRepository: the class responsible for handling the persistence of Tasks.  
The persistence layer of our application is backed by an in-memory database called HSQLDB. We would typically use a production-ready database like PostgreSQL or MySQL on real applications, but for this tutorial, this in-memory database will be enough.  
  
Enabling User Registration on Spring Boot APIs  
Now that we took a look at the endpoints that our RESTful Spring Boot API exposes, we are going to start securing it. The first step is to allow new users to register themselves. The classes that we will create in this feature will belong to a new package called com.auth0.samples.authapi.user. Let's create this package and add a new entity class called ApplicationUser to it:  
  
package com.auth0.samples.authapi.user;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
@Entity  
public class ApplicationUser {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private long id;  
 private String username;  
 private String password;  
 public long getId() {  
 return id;  
 }  
 public String getUsername() {  
 return username;  
 }  
 public void setUsername(String username) {  
 this.username = username;  
 }  
 public String getPassword() {  
 return password;  
 }  
 public void setPassword(String password) {  
 this.password = password;  
 }  
}  
This entity class contains three properties:  
  
The id that works as the primary identifier of a user instance in the application.  
The username that will be used by users to identify themselves.  
And the password to check the user identity.  
To manage the persistence layer of this entity, we will create an interface called ApplicationUserRepository. This interface will be an extension of JpaRepository—which gives us access to some common methods like save—and will be created in the same package of the ApplicationUser class:  
  
package com.auth0.samples.authapi.user;  
import org.springframework.data.jpa.repository.JpaRepository;  
public interface ApplicationUserRepository extends JpaRepository<ApplicationUser, Long> {  
 ApplicationUser findByUsername(String username);  
}  
We have also added a method called findByUsername to this interface. This method will be used when we implement the authentication feature.  
  
The endpoint that enables new users to register will be handled by a new @Controller class. We will call this controller UserController and add it to the same package as the ApplicationUser class:  
  
package com.auth0.samples.authapi.user;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
@RestController  
@RequestMapping("/users")  
public class UserController {  
 private ApplicationUserRepository applicationUserRepository;  
 private BCryptPasswordEncoder bCryptPasswordEncoder;  
 public UserController(ApplicationUserRepository applicationUserRepository,  
 BCryptPasswordEncoder bCryptPasswordEncoder) {  
 this.applicationUserRepository = applicationUserRepository;  
 this.bCryptPasswordEncoder = bCryptPasswordEncoder;  
 }  
 @PostMapping("/sign-up")  
 public void signUp(@RequestBody ApplicationUser user) {  
 user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));  
 applicationUserRepository.save(user);  
 }  
}  
The implementation of the endpoint is quite simple. All it does is encrypt the password of the new user (holding it as plain text wouldn't be a good idea) and then save it to the database. The encryption process is handled by an instance of BCryptPasswordEncoder, which is a class that belongs to the Spring Security framework.  
  
Right now we have two gaps in our application:  
  
We didn't include the Spring Security framework as a dependency to our project.  
There is no default instance of BCryptPasswordEncoder that can be injected in the UserController class.  
The first problem we solve by adding the Spring Security framework dependency to the ./build.gradle file:  
  
...  
dependencies {  
 ...  
 compile("org.springframework.boot:spring-boot-starter-security")  
}  
The second problem, the missing BCryptPasswordEncoder instance, we solve by implementing a method that generates an instance of BCryptPasswordEncoder. This method must be annotated with @Bean and we will add it in the Application class:  
  
package com.auth0.samples.authapi;  
// ... other imports  
import org.springframework.context.annotation.Bean;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
@SpringBootApplication  
public class Application {  
 @Bean  
 public BCryptPasswordEncoder bCryptPasswordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
 // ... main method definition  
}  
This ends the user registration feature, but we still lack support for user authentication and authorization. Let's tackle these features next.  
  
User Authentication and Authorization on Spring Boot  
To support both authentication and authorization in our application, we are going to:  
  
Implement an authentication filter to issue JWTs to users sending credentials.  
Implement an authorization filter to validate requests containing JWTs.  
Create a custom implementation of UserDetailsService to help Spring Security loading user-specific data in the framework.  
And extend the WebSecurityConfigurerAdapter class to customize the security framework to our needs.  
Before proceeding to the development of these filters and classes, let's create a new package called com.auth0.samples.authapi.security. This package will hold all the code related to our app's security.  
  
The Authentication Filter  
The first element that we are going to create is the class responsible for the authentication process. We are going to call this class JWTAuthenticationFilter, and we will implement it with the following code:  
  
package com.auth0.samples.authapi.security;  
import com.auth0.samples.authapi.user.ApplicationUser;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.SignatureAlgorithm;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.AuthenticationException;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
import javax.servlet.FilterChain;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.Date;  
import static com.auth0.samples.authapi.security.SecurityConstants.EXPIRATION\_TIME;  
import static com.auth0.samples.authapi.security.SecurityConstants.HEADER\_STRING;  
import static com.auth0.samples.authapi.security.SecurityConstants.SECRET;  
import static com.auth0.samples.authapi.security.SecurityConstants.TOKEN\_PREFIX;  
public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {  
 private AuthenticationManager authenticationManager;  
 public JWTAuthenticationFilter(AuthenticationManager authenticationManager) {  
 this.authenticationManager = authenticationManager;  
 }  
 @Override  
 public Authentication attemptAuthentication(HttpServletRequest req,  
 HttpServletResponse res) throws AuthenticationException {  
 try {  
 ApplicationUser creds = new ObjectMapper()  
 .readValue(req.getInputStream(), ApplicationUser.class);  
 return authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 creds.getUsername(),  
 creds.getPassword(),  
 new ArrayList<>())  
 );  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 @Override  
 protected void successfulAuthentication(HttpServletRequest req,  
 HttpServletResponse res,  
 FilterChain chain,  
 Authentication auth) throws IOException, ServletException {  
 String token = Jwts.builder()  
 .setSubject(((User) auth.getPrincipal()).getUsername())  
 .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))  
 .signWith(SignatureAlgorithm.HS512, SECRET)  
 .compact();  
 res.addHeader(HEADER\_STRING, TOKEN\_PREFIX + token);  
 }  
}  
Note that the authentication filter that we created extends the UsernamePasswordAuthenticationFilter class. When we add a new filter to Spring Security, we can explicitly define where in the filter chain we want that filter, or we can let the framework figure it out by itself. By extending the filter provided within the security framework, Spring can automatically identify the best place to put it in the security chain.  
  
Our custom authentication filter overwrites two methods of the base class:  
  
attemptAuthentication: where we parse the user's credentials and issue them to the AuthenticationManager.  
successfulAuthentication: which is the method called when a user successfully logs in. We use this method to generate a JWT for this user.  
Our IDE will probably complain about the code in this class for two reasons. First, because the code imports four constants from a class that we haven't created yet, SecurityConstants. Second, because this class generates JWTs with the help of a class called Jwts, which belongs to a library that we haven't added as a dependency to our project.  
  
Let's solve the missing dependency first. In the ./build.gradle file, let's add the following line of code:  
  
...  
dependencies {  
 ...  
 compile("io.jsonwebtoken:jjwt:0.7.0")  
}  
This will add the Java JWT: JSON Web Token for Java and Android library to our project and will solve the issue of the missing classes. Now we have to create the SecurityConstants class:  
  
package com.auth0.samples.authapi.security;  
public class SecurityConstants {  
 public static final String SECRET = "SecretKeyToGenJWTs";  
 public static final long EXPIRATION\_TIME = 864\_000\_000; // 10 days  
 public static final String TOKEN\_PREFIX = "Bearer ";  
 public static final String HEADER\_STRING = "Authorization";  
 public static final String SIGN\_UP\_URL = "/users/sign-up";  
}  
This class contains all four constants referenced by the JWTAuthenticationFilter class, alongside a SIGN\_UP\_URL constant that will be used later.  
  
The Authorization Filter  
As we have implemented the filter responsible for authenticating users, we now need to implement the filter responsible for user authorization. We create this filter as a new class, called JWTAuthorizationFilter, in the com.auth0.samples.authapi.security package:  
  
package com.auth0.samples.authapi.security;  
import io.jsonwebtoken.Jwts;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;  
import javax.servlet.FilterChain;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.io.IOException;  
import java.util.ArrayList;  
import static com.auth0.samples.authapi.security.SecurityConstants.HEADER\_STRING;  
import static com.auth0.samples.authapi.security.SecurityConstants.SECRET;  
import static com.auth0.samples.authapi.security.SecurityConstants.TOKEN\_PREFIX;  
public class JWTAuthorizationFilter extends BasicAuthenticationFilter {  
 public JWTAuthorizationFilter(AuthenticationManager authManager) {  
 super(authManager);  
 }  
 @Override  
 protected void doFilterInternal(HttpServletRequest req,  
 HttpServletResponse res,  
 FilterChain chain) throws IOException, ServletException {  
 String header = req.getHeader(HEADER\_STRING);  
 if (header == null || !header.startsWith(TOKEN\_PREFIX)) {  
 chain.doFilter(req, res);  
 return;  
 }  
 UsernamePasswordAuthenticationToken authentication = getAuthentication(req);  
 SecurityContextHolder.getContext().setAuthentication(authentication);  
 chain.doFilter(req, res);  
 }  
 private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {  
 String token = request.getHeader(HEADER\_STRING);  
 if (token != null) {  
 // parse the token.  
 String user = Jwts.parser()  
 .setSigningKey(SECRET)  
 .parseClaimsJws(token.replace(TOKEN\_PREFIX, ""))  
 .getBody()  
 .getSubject();  
 if (user != null) {  
 return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());  
 }  
 return null;  
 }  
 return null;  
 }  
}  
We have extended the BasicAuthenticationFilter to make Spring replace it in the filter chain with our custom implementation. The most important part of the filter that we've implemented is the private getAuthentication method. This method reads the JWT from the Authorization header, and then uses Jwtsto validate the token. If everything is in place, we set the user in the SecurityContext and allow the request to move on.  
  
Integrating the Security Filters on Spring Boot  
Now that we have both security filters properly created, we have to configure them on the Spring Security filter chain. To do that, we are going to create a new class called WebSecurity in the com.auth0.samples.authapi.securitypackage:  
  
package com.auth0.samples.authapi.security;  
import org.springframework.http.HttpMethod;  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import static com.auth0.samples.authapi.security.SecurityConstants.SIGN\_UP\_URL;  
@EnableWebSecurity  
public class WebSecurity extends WebSecurityConfigurerAdapter {  
 private UserDetailsService userDetailsService;  
 private BCryptPasswordEncoder bCryptPasswordEncoder;  
 public WebSecurity(UserDetailsService userDetailsService, BCryptPasswordEncoder bCryptPasswordEncoder) {  
 this.userDetailsService = userDetailsService;  
 this.bCryptPasswordEncoder = bCryptPasswordEncoder;  
 }  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable().authorizeRequests()  
 .antMatchers(HttpMethod.POST, SIGN\_UP\_URL).permitAll()  
 .anyRequest().authenticated()  
 .and()  
 .addFilter(new JWTAuthenticationFilter(authenticationManager()))  
 .addFilter(new JWTAuthorizationFilter(authenticationManager()));  
 }  
 @Override  
 public void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);  
 }  
}  
We have annotated this class with @EnableWebSecurity and made it extend WebSecurityConfigurerAdapter to take advantage of the default web security configuration provided by Spring Security. This allows us to fine-tune the framework to our needs by overwriting two methods:  
  
configure(HttpSecurity http): a method where we can define which resources are public and which are secured. In our case, we set the SIGN\_UP\_URL endpoint as being public and everything else as being secured. We also configure custom security filter in the Spring Security filter chain.  
configure(AuthenticationManagerBuilder auth): a method where we defined a custom implementation of UserDetailsService to load user-specific data in the security framework. We have also used this method to set the encrypt method used by our application (BCryptPasswordEncoder).  
Spring Security doesn't come with a concrete implementation of UserDetailsService that we could use out of the box with our in-memory database. Therefore, we create a new class called UserDetailsServiceImpl in the com.auth0.samples.authapi.user package to provide one:  
  
package com.auth0.samples.authapi.user;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.stereotype.Service;  
import static java.util.Collections.emptyList;  
@Service  
public class UserDetailsServiceImpl implements UserDetailsService {  
 private ApplicationUserRepository applicationUserRepository;  
 public UserDetailsServiceImpl(ApplicationUserRepository applicationUserRepository) {  
 this.applicationUserRepository = applicationUserRepository;  
 }  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 ApplicationUser applicationUser = applicationUserRepository.findByUsername(username);  
 if (applicationUser == null) {  
 throw new UsernameNotFoundException(username);  
 }  
 return new User(applicationUser.getUsername(), applicationUser.getPassword(), emptyList());  
 }  
}  
The only method that we had to implement is loadUserByUsername. When a user tries to authenticate, this method receives the username, searches the database for a record containing it, and (if found) returns an instance of User. The properties of this instance (username and password) are then checked against the credentials passed by the user in the login request. This last process is executed outside this class, by the Spring Security framework.  
  
We can now rest assured that our endpoints won't be publicly exposed and that we can support authentication and authorization with JWTS on Spring Boot properly. To check everything, let's run our application (through the IDE or through gradle bootRun) and issue the following requests:  
  
**# issues a GET request to retrieve tasks with no JWT  
# HTTP 403 Forbidden status is expected  
curl http://localhost:8080/tasks  
# registers a new user  
curl -H "Content-Type: application/json" -X POST -d '{  
 "username": "admin",  
 "password": "password"  
}' http://localhost:8080/users/sign-up  
# logs into the application (JWT is generated)  
curl -i -H "Content-Type: application/json" -X POST -d '{  
 "username": "admin",  
 "password": "password"  
}' http://localhost:8080/login  
# issue a POST request, passing the JWT, to create a task  
# remember to replace xxx.yyy.zzz with the JWT retrieved above  
curl -H "Content-Type: application/json" \  
-H "Authorization: Bearer xxx.yyy.zzz" \  
-X POST -d '{  
 "description": "Buy watermelon"  
}' http://localhost:8080/tasks  
# issue a new GET request, passing the JWT  
# remember to replace xxx.yyy.zzz with the JWT retrieved above  
curl -H "Authorization: Bearer xxx.yyy.zzz" http://localhost:8080/tasks  
Conclusion**  
Securing RESTful Spring Boot API with JWTs is not a hard task. This article showed that by creating a couple of classes and extending a few others provided by Spring Security, we can protect our endpoints from unknown users, enable users to register themselves and authenticate existing users based on JWTs.  
  
Of course, for it to be a production-ready application we would need a few more features, like password retrieval, but, hopefully, this article demystified the most sensible parts of dealing with JWTs to authorize requests on Spring Boot applications.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

-------------------> UNIT TEST FOR WEB SERVICE

Unit Testing APIs is an important part of [API testing](https://www.blazemeter.com/api-testing?utm_source=blog&utm_medium=BM_blog&utm_campaign=spring-boot-rest-api-unit-testing-with-junit) because Unit Testing ensures that API components will function properly. In this article, we will learn how to cover Spring Boot REST APIs with JUnit. Spring Boot is an open-source framework for application creation, and where we create our APIs.

There are many different variations and techniques to Unit Test APIs. I prefer the following combination: [Spring Boot](https://projects.spring.io/spring-boot/), [JUnit](http://junit.org/junit4/), [MockMvc](https://docs.spring.io/spring-security/site/docs/current/reference/html/test-mockmvc.html), and [Mockito](http://site.mockito.org/), because they are all open-source and support Java, which is my preferred language.

To start, we have to have [IntelliJ IDEA](https://www.jetbrains.com/idea/), as an IDE for development, and [JDK8](https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html), for using Java for development. These are my personal preferences but [Eclipse](https://eclipse.org/), [NetBeans](https://netbeans.org/), or even a simple text editor could also be used.

Now, let’s setup a project. You can also participate - the source code is located [here](https://github.com/LosBandolero/blazedemo). We will be testing controllers and repository classes. In short, we have 4 controllers (ArrivalController, DepartureController, UsersController, FlightsController) and 4 repositories (ArrivalRepository, DepartureRepository, UsersRepository, FlightsRepository). We will write tests per controller (testing the size of the JSON Object, the status of the call to the endpoint and an assertion on one item from the JSON object) and tests per repository (inserting two new items in a table and making sure the return object is equal).

## Step 1 - Create an API Testing Project

1. [Install](https://www.jetbrains.com/idea/#chooseYourEdition) IntelliJ IDEA.

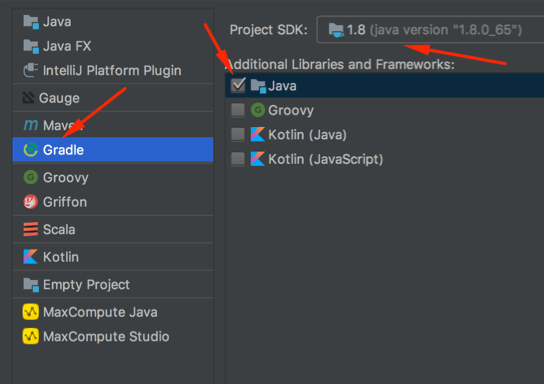
2. Make sure you have [JDK installed](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) (at least version 1.8.XXX).

Now we will create a new project.

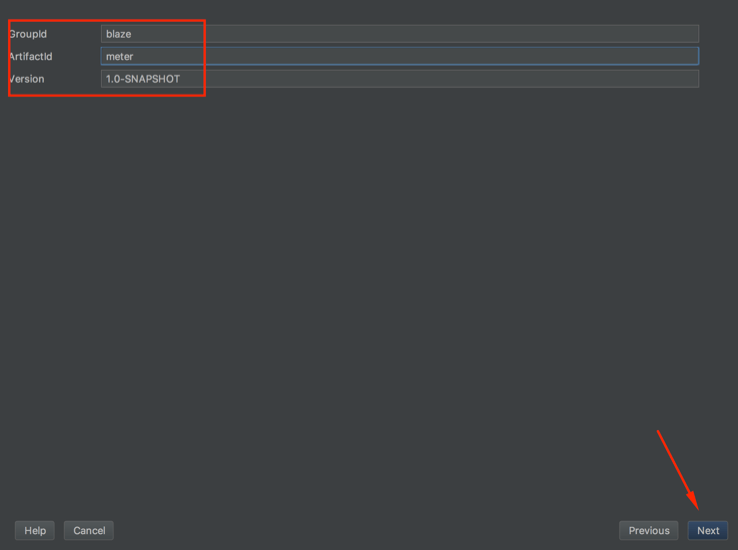
3. Open IntelliJ and click “Create New Project.”



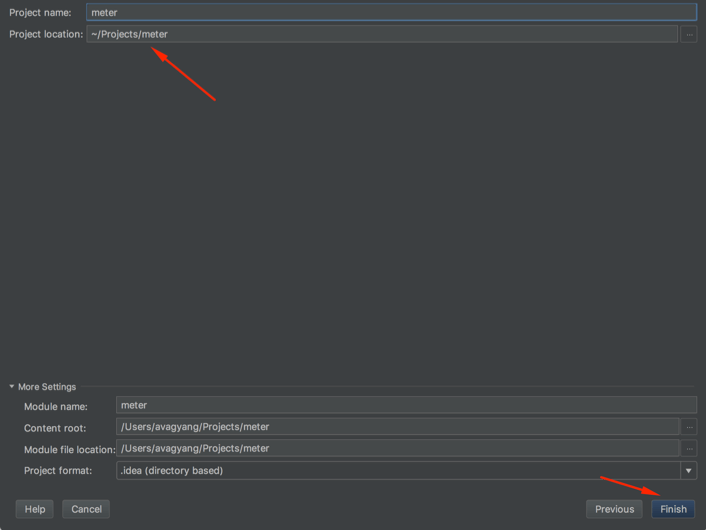
4. Select Gradle, Java, and the JDK version.



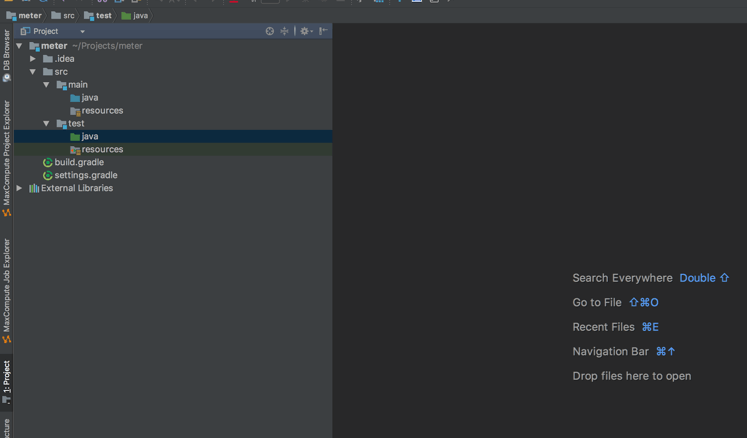
5. Name your project.



6. Choose the project destination.



If you did everything correctly, you should now see this window with an empty Java project:



## Step 2 - Add Dependencies

Now that we have a project, we need to setup the dependencies. You can use these dependencies since they are public.

To do that, double click on your build.gradle file and add the following gradle configuration file:

group 'blazemeter'

version '1.0-SNAPSHOT'

buildscript {

repositories {

jcenter()

mavenCentral()

maven { url "http://repo.spring.io/libs-snapshot" }

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.2.RELEASE")

}

}

apply plugin: 'java'

apply plugin: 'idea'

apply plugin: 'io.spring.dependency-management'

apply plugin: 'org.springframework.boot'

sourceSets {

main.java.srcDir "src/main/java"

main.resources.srcDir "src/main/resources"

test.java.srcDir "src/test/java"

test.resources.srcDir "src/test/resources"

}

jar {

baseName = 'blaze-demo-api'

version = '1.0'

}

bootRepackage {

mainClass = 'com.demo.BlazeMeterApi'

}

dependencyManagement {

imports {

mavenBom 'io.spring.platform:platform-bom:Brussels-SR2'

}

}

repositories {

mavenCentral()

jcenter()

maven { url "http://repo.spring.io/libs-snapshot" }

}

sourceCompatibility = 1.8

targetCompatibility = 1.8

dependencies {

compile group: 'org.springframework', name: 'spring-core'

compile group: 'org.springframework.boot', name: 'spring-boot-starter-jdbc'

compile group: 'org.springframework.boot', name: 'spring-boot-starter-web'

compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'

compile group: 'org.springframework.boot', name: 'spring-boot-starter-security'

compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-jpa'

compile group: 'org.springframework.security.oauth', name: 'spring-security-oauth2'

compile group: 'com.fasterxml.jackson.datatype', name: 'jackson-datatype-hibernate4'

compile group: 'mysql', name: 'mysql-connector-java'

compile group: 'io.rest-assured', name: 'rest-assured', version: '3.0.3'

compile group: 'io.rest-assured', name: 'json-schema-validator', version: '3.0.3'

testCompile group: 'org.springframework.boot', name: 'spring-boot-starter-test'

testCompile group: 'org.springframework.security', name: 'spring-security-test'

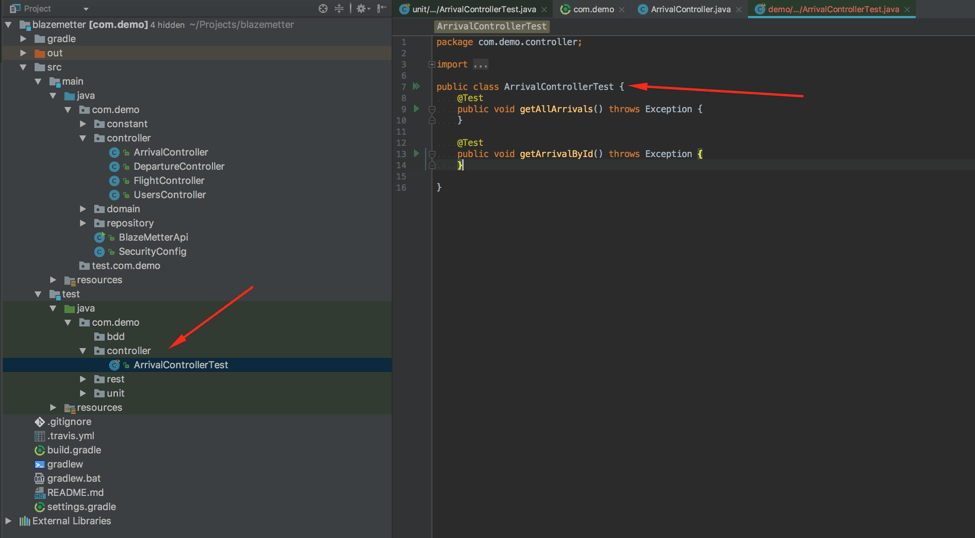
testCompile group: 'junit', name: 'junit'

testCompile group: 'org.hsqldb', name: 'hsqldb'

}

## Step 3 - Write Your Unit Test via JUnit

In IntelliJ IDEA, go to the class that you want to test. Hit Cmd + Shift + T and a popup will appear. In the popup, select “Create New Test...”. Then, IntelliJ IDEA will create a file for writing the test in. The file will be created in the default place. In our case, if we are going to cover the class ArrivalController, it will create a test class with the path test/java/com/demo/controller. You can see this in the following screenshot:



I personally prefer to group tests (as you can see in the same picture - there are 3 folders: bdd, rest, unit) according to the test types: REST, UNIT, BDD, etc. For that reason, I create the test classes by myself. For this example, ArrivalControllerTest is located at the test/java/com/demo/unit/controller path.

Here is the test class itself:

package com.demo.unit.controller;

import com.demo.controller.ArrivalController;

import com.demo.domain.Arrival;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.test.context.junit4.SpringRunner;

import org.springframework.test.web.servlet.MockMvc;

import java.util.List;

import static com.demo.constant.Paths.ARRIVAL;

import static com.demo.constant.Paths.VERSION;

import static java.util.Collections.singletonList;

import static org.hamcrest.collection.IsCollectionWithSize.hasSize;

import static org.hamcrest.core.Is.is;

import static org.mockito.BDDMockito.given;

import static org.springframework.http.MediaType.APPLICATION\_JSON;

import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.user;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)

@WebMvcTest(ArrivalController.class)

public class ArrivalControllerTest {

@Autowired

private MockMvc mvc;

@MockBean

private ArrivalController arrivalController;

@Test

public void getArrivals() throws Exception {

Arrival arrival = new Arrival();

arrival.setCity("Yerevan");

List<Arrival> allArrivals = singletonList(arrival);

given(arrivalController.getAllArrivals()).willReturn(allArrivals);

mvc.perform(get(VERSION + ARRIVAL + "all")

.with(user("blaze").password("Q1w2e3r4"))

.contentType(APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("$", hasSize(1)))

.andExpect(jsonPath("$[0].city", is(arrival.getCity())));

}

@Test

public void getArrivalsById() throws Exception {

Arrival arrival = new Arrival();

arrival.setCity("Yerevan");

given(arrivalController.getArrivalById(arrival.getId())).willReturn(arrival);

mvc.perform(get(VERSION + ARRIVAL + arrival.getId())

.with(user("blaze").password("Q1w2e3r4"))

.contentType(APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("city", is(arrival.getCity())));

}

}

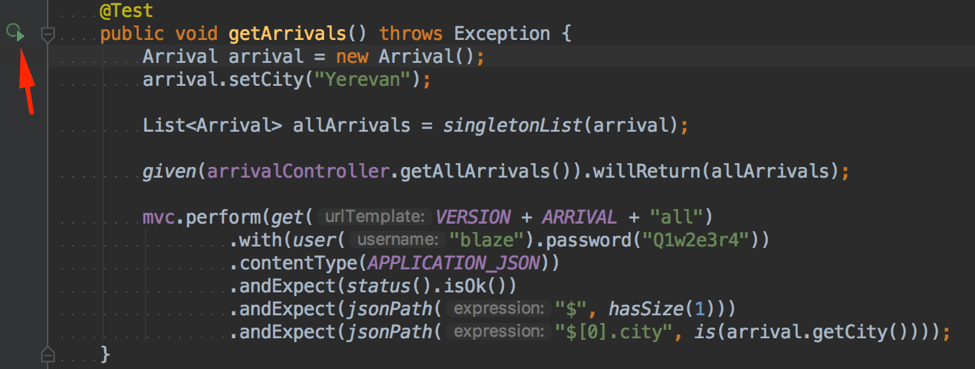
In this test class, we have two test methods, getArrivals() and getArrivalsById(). The reason we have two is that we have two methods in the controller itself, so we want to test them both.

The getArrivals() method does the following (code snippet above):

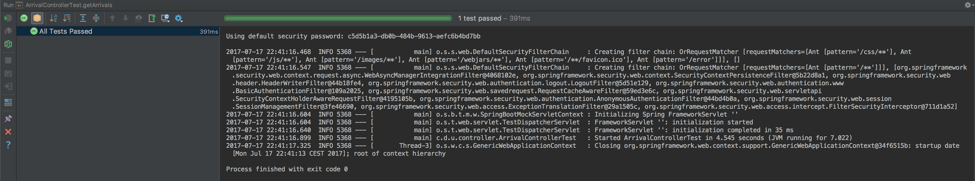
* Creates an Arrival entity and sets the test value for the city.
* Creates a list of Arrivals (because we will have just one member, it can be a singeltonList).
* Using **given** from mockito, makes sure the mocked ArrivalController will return a list of Arrivals.
* Performs a **GET** request to the mocked controller with the credentials and performs simple assertions:
  + Status - 200 (which isOk).
  + The JSON object has one member.
  + The JSON body has a city key with the value we set.

The second test method does the same for the getArrivalsById(). The difference is that it assumes one JSON object result instead of a list of Arrivals of JSON objects.

That’s all. Now we can execute this single test by clicking the play button at the beginning of the line of the method name (see the following picture). The purpose of this execution is to make sure the test is working properly:



After execution, you will see the results of test execution, including the status, count, and stacktrace. We can see that our test passed (on the left side), the number of tests (the progress bar is in the middle up top) and the stacktrace for execution.



That’s it for that one controller. The next step will be to add tests for all controllers.

## Step 4 - Setting Up the Unit Tests for the APIs

Now, let’s test one API repository class with this unit test. This is a unit test that is covering the database testing part. For example, writing test data to the DB and afterward verifying it is properly stored.

Like in the previous part, go to the ArrivalRepository class, hit Cmd + Shift + T and create a new test class via IntelliJ IDEA or by yourself.

This is the code for that test class:

package com.demo.unit.repository;

import com.demo.domain.Arrival;

import com.demo.repository.ArrivalRepository;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

import org.springframework.test.context.ActiveProfiles;

import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

import static org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace.NONE;

@ActiveProfiles("test")

@RunWith(SpringRunner.class)

@DataJpaTest

@AutoConfigureTestDatabase(replace = NONE)

public class ArrivalRepositoryTest {

@Autowired

private TestEntityManager entityManager;

@Autowired

private ArrivalRepository arrivalRepository;

@Test

public void whenFindAll() {

//given

Arrival firstArrival = new Arrival();

firstArrival.setCity("Yerevan");

entityManager.persist(firstArrival);

entityManager.flush();

Arrival secondArrival = new Arrival();

secondArrival.setCity("Israel");

entityManager.persist(secondArrival);

entityManager.flush();

//when

List<Arrival> arrivals = arrivalRepository.findAll();

//then

assertThat(arrivals.size()).isEqualTo(9);

assertThat(arrivals.get(7)).isEqualTo(firstArrival);

assertThat(arrivals.get(8)).isEqualTo(secondArrival);

}

@Test

public void whenFindAllById() {

//given

Arrival arrival = new Arrival();

arrival.setCity("Yerevan");

entityManager.persist(arrival);

entityManager.flush();

//when

Arrival testArrival = arrivalRepository.findAllById(arrival.getId());

//then

assertThat(testArrival.getCity()).isEqualTo(arrival.getCity());

}

}

In this test, we are using the H2 database for testing. This is common practice. Otherwise, you need to have the same type of database set up in all test/dev environments, maintain them and make sure you clean them up after test execution. This is not necessary when you use the H2 DB because it is in the memory. After the test(s) is finished, the database will be fully dropped.

The test covers the following:

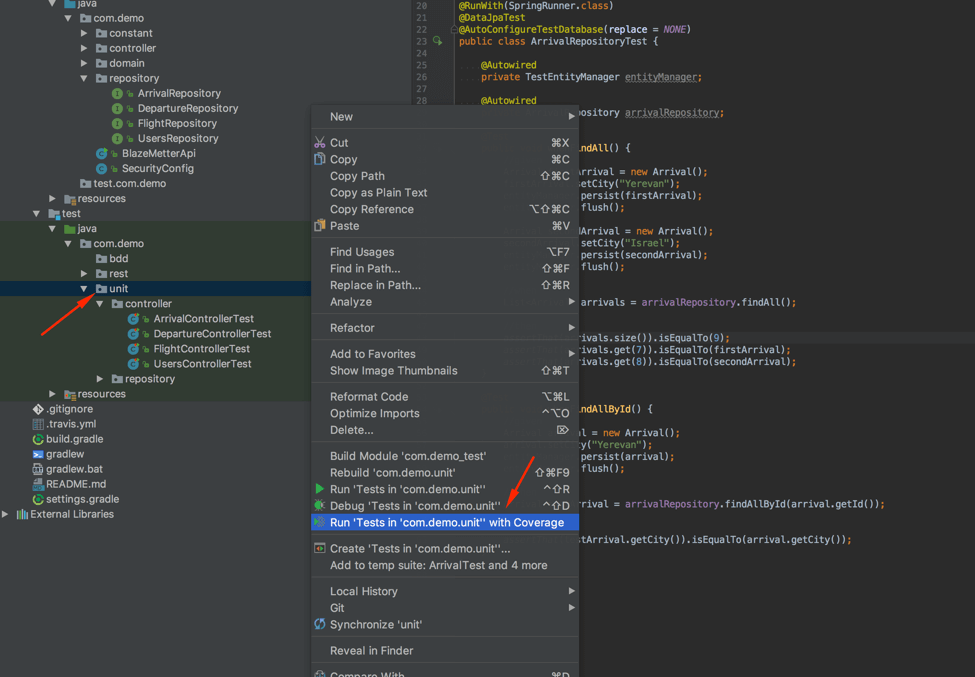
* Use entityManager to create two new rows of data in the Arrival table of test H2 database.
* Search for all records from database via a findAll query.
* Perform assertions on the size of the data and equality of gathered objects.

The second method does the same, but because it is for findAllById, it is checking just one object equality.

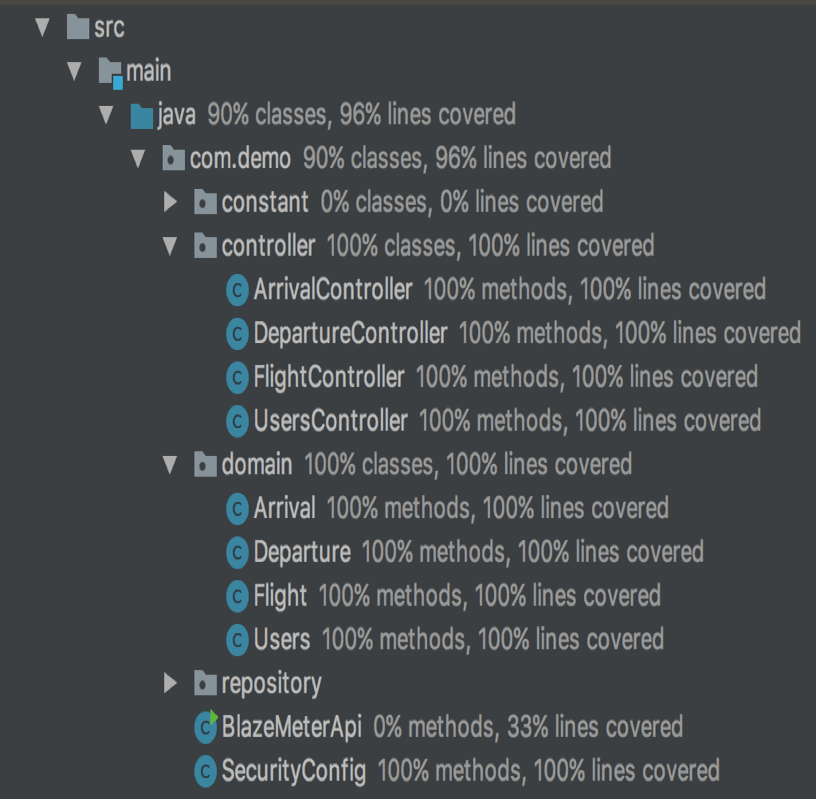
That’s all! Now you can do this for all other repository classes with their methods.

## Step 5 - Running the Unit Tests on Your APIs

After all the controller and repository class tests are set up, we need to execute them. Do that by highlighting the folder unit -> right click -> select “Run ‘Tests’ in ‘com.demo.unit’ with Coverage” (see next picture), so we will also have the code coverage report.



These are the coverage results, showing how many classes and methods are covered by unit tests:



That’s it! You now know how to run Unit Tests with JUnit for REST APIs.

////////////////////////////////////////////////////////////////////

------> Using Java Reflection?

Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them.  
The ability to examine and manipulate a Java class from within itself may not sound like very much, but in other programming languages this feature simply doesn't exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program.  
One tangible use of reflection is in JavaBeans, where software components can be manipulated visually via a builder tool. The tool uses reflection to obtain the properties of Java components (classes) as they are dynamically loaded.  
A Simple Example  
To see how reflection works, consider this simple example:  
 import java.lang.reflect.\*;  
   
 public class DumpMethods {  
 public static void main(String args[])  
 {  
 try {  
 Class c = Class.forName(args[0]);  
 Method m[] = c.getDeclaredMethods();  
 for (int i = 0; i < m.length; i++)  
 System.out.println(m[i].toString());  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
For an invocation of:  
 java DumpMethods java.util.Stack   
the output is:  
 public java.lang.Object java.util.Stack.push(  
 java.lang.Object)  
 public synchronized   
 java.lang.Object java.util.Stack.pop()  
 public synchronized  
 java.lang.Object java.util.Stack.peek()  
 public boolean java.util.Stack.empty()  
 public synchronized   
 int java.util.Stack.search(java.lang.Object)  
That is, the method names of class java.util.Stack are listed, along with their fully qualified parameter and return types.  
This program loads the specified class using class.forName, and then calls getDeclaredMethods to retrieve the list of methods defined in the class. java.lang.reflect.Method is a class representing a single class method.  
Setting Up to Use Reflection  
The reflection classes, such as Method, are found in java.lang.reflect. There are three steps that must be followed to use these classes. The first step is to obtain a java.lang.Class object for the class that you want to manipulate. java.lang.Class is used to represent classes and interfaces in a running Java program.  
One way of obtaining a Class object is to say:  
 Class c = Class.forName("java.lang.String");   
to get the Class object for String. Another approach is to use:  
 Class c = int.class;   
or  
 Class c = Integer.TYPE;   
to obtain Class information on fundamental types. The latter approach accesses the predefined TYPE field of the wrapper (such as Integer) for the fundamental type.  
The second step is to call a method such as getDeclaredMethods, to get a list of all the methods declared by the class.  
Once this information is in hand, then the third step is to use the reflection API to manipulate the information. For example, the sequence:  
 Class c = Class.forName("java.lang.String"); Method m[] = c.getDeclaredMethods(); System.out.println(m[0].toString());   
will display a textual representation of the first method declared in String.  
In the examples below, the three steps are combined to present self contained illustrations of how to tackle specific applications using reflection.  
Simulating the instanceof Operator  
Once Class information is in hand, often the next step is to ask basic questions about the Class object. For example, the Class.isInstance method can be used to simulate the instanceof operator:  
 class A {}  
  
 public class instance1 {  
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("A");  
 boolean b1   
 = cls.isInstance(new Integer(37));  
 System.out.println(b1);  
 boolean b2 = cls.isInstance(new A());  
 System.out.println(b2);  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
   
In this example, a Class object for A is created, and then class instance objects are checked to see whether they are instances of A. Integer(37) is not, but new A() is.  
Finding Out About Methods of a Class  
One of the most valuable and basic uses of reflection is to find out what methods are defined within a class. To do this the following code can be used:  
 import java.lang.reflect.\*;  
  
 public class method1 {  
 private int f1(  
 Object p, int x) throws NullPointerException  
 {  
 if (p == null)  
 throw new NullPointerException();  
 return x;  
 }  
   
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("method1");  
   
 Method methlist[]   
 = cls.getDeclaredMethods();  
 for (int i = 0; i < methlist.length;  
 i++) {   
 Method m = methlist[i];  
 System.out.println("name   
 = " + m.getName());  
 System.out.println("decl class = " +  
 m.getDeclaringClass());  
 Class pvec[] = m.getParameterTypes();  
 for (int j = 0; j < pvec.length; j++)  
 System.out.println("  
 param #" + j + " " + pvec[j]);  
 Class evec[] = m.getExceptionTypes();  
 for (int j = 0; j < evec.length; j++)  
 System.out.println("exc #" + j   
 + " " + evec[j]);  
 System.out.println("return type = " +  
 m.getReturnType());  
 System.out.println("-----");  
 }  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
  
The program first gets the Class description for method1, and then calls getDeclaredMethods to retrieve a list of Method objects, one for each method defined in the class. These include public, protected, package, and private methods. If you use getMethods in the program instead of getDeclaredMethods, you can also obtain information for inherited methods.  
Once a list of the Method objects has been obtained, it's simply a matter of displaying the information on parameter types, exception types, and the return type for each method. Each of these types, whether they are fundamental or class types, is in turn represented by a Class descriptor.  
The output of the program is:  
 name = f1  
 decl class = class method1  
 param #0 class java.lang.Object  
 param #1 int  
 exc #0 class java.lang.NullPointerException  
 return type = int  
 -----  
 name = main  
 decl class = class method1  
 param #0 class [Ljava.lang.String;  
 return type = void  
 -----  
  
Obtaining Information About Constructors  
A similar approach is used to find out about the constructors of a class. For example:  
 import java.lang.reflect.\*;  
   
 public class constructor1 {  
 public constructor1()  
 {  
 }  
   
 protected constructor1(int i, double d)  
 {  
 }  
   
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("constructor1");  
   
 Constructor ctorlist[]  
 = cls.getDeclaredConstructors();  
 for (int i = 0; i < ctorlist.length; i++) {  
 Constructor ct = ctorlist[i];  
 System.out.println("name   
 = " + ct.getName());  
 System.out.println("decl class = " +  
 ct.getDeclaringClass());  
 Class pvec[] = ct.getParameterTypes();  
 for (int j = 0; j < pvec.length; j++)  
 System.out.println("param #"   
 + j + " " + pvec[j]);  
 Class evec[] = ct.getExceptionTypes();  
 for (int j = 0; j < evec.length; j++)  
 System.out.println(  
 "exc #" + j + " " + evec[j]);  
 System.out.println("-----");  
 }  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
  
There is no return-type information retrieved in this example, because constructors don't really have a true return type.  
When this program is run, the output is:  
 name = constructor1  
 decl class = class constructor1  
 -----  
 name = constructor1  
 decl class = class constructor1  
 param #0 int  
 param #1 double  
 -----  
  
Finding Out About Class Fields  
It's also possible to find out which data fields are defined in a class. To do this, the following code can be used:  
 import java.lang.reflect.\*;  
   
 public class field1 {  
 private double d;  
 public static final int i = 37;  
 String s = "testing";  
   
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("field1");  
   
 Field fieldlist[]   
 = cls.getDeclaredFields();  
 for (int i   
 = 0; i < fieldlist.length; i++) {  
 Field fld = fieldlist[i];  
 System.out.println("name  
 = " + fld.getName());  
 System.out.println("decl class = " +  
 fld.getDeclaringClass());  
 System.out.println("type  
 = " + fld.getType());  
 int mod = fld.getModifiers();  
 System.out.println("modifiers = " +  
 Modifier.toString(mod));  
 System.out.println("-----");  
 }  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
  
This example is similar to the previous ones. One new feature is the use of Modifier. This is a reflection class that represents the modifiers found on a field member, for example "private int". The modifiers themselves are represented by an integer, and Modifier.toString is used to return a string representation in the "official" declaration order (such as "static" before "final"). The output of the program is:  
 name = d  
 decl class = class field1  
 type = double  
 modifiers = private  
 -----  
 name = i  
 decl class = class field1  
 type = int  
 modifiers = public static final  
 -----  
 name = s  
 decl class = class field1  
 type = class java.lang.String  
 modifiers =  
 -----   
  
As with methods, it's possible to obtain information about just the fields declared in a class (getDeclaredFields), or to also get information about fields defined in superclasses (getFields).  
Invoking Methods by Name  
So far the examples that have been presented all relate to obtaining class information. But it's also possible to use reflection in other ways, for example to invoke a method of a specified name.  
To see how this works, consider the following example:  
 import java.lang.reflect.\*;  
   
 public class method2 {  
 public int add(int a, int b)  
 {  
 return a + b;  
 }  
   
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("method2");  
 Class partypes[] = new Class[2];  
 partypes[0] = Integer.TYPE;  
 partypes[1] = Integer.TYPE;  
 Method meth = cls.getMethod(  
 "add", partypes);  
 method2 methobj = new method2();  
 Object arglist[] = new Object[2];  
 arglist[0] = new Integer(37);  
 arglist[1] = new Integer(47);  
 Object retobj   
 = meth.invoke(methobj, arglist);  
 Integer retval = (Integer)retobj;  
 System.out.println(retval.intValue());  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
  
Suppose that a program wants to invoke the add method, but doesn't know this until execution time. That is, the name of the method is specified during execution (this might be done by a JavaBeans development environment, for example). The above program shows a way of doing this.  
getMethod is used to find a method in the class that has two integer parameter types and that has the appropriate name. Once this method has been found and captured into a Method object, it is invoked upon an object instance of the appropriate type. To invoke a method, a parameter list must be constructed, with the fundamental integer values 37 and 47 wrapped in Integer objects. The return value (84) is also wrapped in an Integer object.  
Creating New Objects  
There is no equivalent to method invocation for constructors, because invoking a constructor is equivalent to creating a new object (to be the most precise, creating a new object involves both memory allocation and object construction). So the nearest equivalent to the previous example is to say:  
 import java.lang.reflect.\*;  
   
 public class constructor2 {  
 public constructor2()  
 {  
 }  
   
 public constructor2(int a, int b)  
 {  
 System.out.println(  
 "a = " + a + " b = " + b);  
 }  
   
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("constructor2");  
 Class partypes[] = new Class[2];  
 partypes[0] = Integer.TYPE;  
 partypes[1] = Integer.TYPE;  
 Constructor ct   
 = cls.getConstructor(partypes);  
 Object arglist[] = new Object[2];  
 arglist[0] = new Integer(37);  
 arglist[1] = new Integer(47);  
 Object retobj = ct.newInstance(arglist);  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }   
  
which finds a constructor that handles the specified parameter types and invokes it, to create a new instance of the object. The value of this approach is that it's purely dynamic, with constructor lookup and invocation at execution time, rather than at compilation time.  
Changing Values of Fields  
Another use of reflection is to change the values of data fields in objects. The value of this is again derived from the dynamic nature of reflection, where a field can be looked up by name in an executing program and then have its value changed. This is illustrated by the following example:  
 import java.lang.reflect.\*;  
   
 public class field2 {  
 public double d;  
   
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName("field2");  
 Field fld = cls.getField("d");  
 field2 f2obj = new field2();  
 System.out.println("d = " + f2obj.d);  
 fld.setDouble(f2obj, 12.34);  
 System.out.println("d = " + f2obj.d);  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }   
  
In this example, the d field has its value set to 12.34.  
Using Arrays  
One final use of reflection is in creating and manipulating arrays. Arrays in the Java language are a specialized type of class, and an array reference can be assigned to an Object reference.  
To see how arrays work, consider the following example:  
 import java.lang.reflect.\*;  
   
 public class array1 {  
 public static void main(String args[])  
 {  
 try {  
 Class cls = Class.forName(  
 "java.lang.String");  
 Object arr = Array.newInstance(cls, 10);  
 Array.set(arr, 5, "this is a test");  
 String s = (String)Array.get(arr, 5);  
 System.out.println(s);  
 }  
 catch (Throwable e) {  
 System.err.println(e);  
 }  
 }  
 }  
  
This example creates a 10-long array of Strings, and then sets location 5 in the array to a string value. The value is retrieved and displayed.  
A more complex manipulation of arrays is illustrated by the following code:  
 import java.lang.reflect.\*;  
   
 public class array2 {  
 public static void main(String args[])  
 {  
 int dims[] = new int[]{5, 10, 15};  
 Object arr   
 = Array.newInstance(Integer.TYPE, dims);  
   
 Object arrobj = Array.get(arr, 3);  
 Class cls =   
 arrobj.getClass().getComponentType();  
 System.out.println(cls);  
 arrobj = Array.get(arrobj, 5);  
 Array.setInt(arrobj, 10, 37);  
   
 int arrcast[][][] = (int[][][])arr;  
 System.out.println(arrcast[3][5][10]);  
 }  
 }  
  
This example creates a 5 x 10 x 15 array of ints, and then proceeds to set location [3][5][10] in the array to the value 37. Note here that a multi-dimensional array is actually an array of arrays, so that, for example, after the first Array.get, the result in arrobj is a 10 x 15 array. This is peeled back once again to obtain a 15-long array, and the 10th slot in that array is set using Array.setInt.  
Note that the type of array that is created is dynamic, and does not have to be known at compile time.  
Summary  
Java reflection is useful because it supports dynamic retrieval of information about classes and data structures by name, and allows for their manipulation within an executing Java program. This feature is extremely powerful and has no equivalent in other conventional languages such as C, C++, Fortran, or Pascal.  
Glen McCluskey has focused on programming languages since 1988. He consults in the areas of Java and C++ performance, testing, and technical documentation.

////////////////////////////////////////////////////////////////////

Jvm classloader?

The Java Classloader is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine.[1] Usually classes are only loaded on demand. The Java run time system does not need to know about files and file systems because of classloaders. Delegation is an important concept to understand when learning about classloaders.  
  
A software library is a collection of related object code. In the Java language, libraries are typically packaged in JAR files. Libraries can contain objects of different types. The most important type of object contained in a Jar file is a Java class. A class can be thought of as a named unit of code. The class loader is responsible for locating libraries, reading their contents, and loading the classes contained within the libraries. This loading is typically done "on demand", in that it does not occur until the class is called by the program. A class with a given name can only be loaded once by a given classloader.  
  
Each Java class must be loaded by a class loader.[2] Furthermore, Java programs may make use of external libraries (that is, libraries written and provided by someone other than the author of the program) or they may be composed, at least in part, of a number of libraries.  
  
When the JVM is started, three class loaders are used:[3][4]  
  
Bootstrap class loader  
Extensions class loader  
System class loader  
The bootstrap class loader loads the core Java libraries[5] located in the <JAVA\_HOME>/jre/lib directory. This class loader, which is part of the core JVM, is written in native code.  
  
The extensions class loader loads the code in the extensions directories (<JAVA\_HOME>/jre/lib/ext,[6] or any other directory specified by the java.ext.dirs system property). It is implemented by the sun.misc.Launcher$ExtClassLoader class.  
  
The system class loader loads code found on java.class.path, which maps to the CLASSPATH environment variable. This is implemented by the sun.misc.Launcher$AppClassLoader class.

The Java class loader is written in Java. It is therefore possible to create your own class loader without understanding the finer details of the Java Virtual Machine. Every Java class loader has a parent class loader, defined when a new class loader is instantiated or set to the virtual machine's system default class loader.  
  
This makes it possible (for example):  
  
to load or unload classes at runtime (for example to load libraries dynamically at runtime, even from an HTTP resource). This is an important feature for:  
implementing scripting languages, such as Jython  
using bean builders  
allowing user-defined extensibility  
allowing multiple namespaces to communicate. This is one of the foundations of CORBA / RMI protocols for example.  
to change the way the bytecode is loaded (for example, it is possible to use encrypted Java class bytecode[7]).  
to modify the loaded bytecode (for example, for load-time weaving of aspects when using aspect-oriented programming).

//////////////////////////////////////////////////////////////////////////////

What is classpath?

When programming in Java, you make other classes available to the class you are writing by putting something like this at the top of your source file:  
  
import org.javaguy.coolframework.MyClass;  
Or sometimes you 'bulk import' stuff by saying:  
  
import org.javaguy.coolframework.\*;  
So later in your program when you say:  
  
MyClass mine = new MyClass();  
The Java Virtual Machine will know where to find your compiled class.  
  
It would be impractical to have the VM look through every folder on your machine, so you have to provide the VM a list of places to look. This is done by putting folder and jar files on your classpath.  
  
Before we talk about how the classpath is set, let's talk about .class files, packages, and .jar files.  
  
First, let's suppose that MyClass is something you built as part of your project, and it is in a directory in your project called output. The .class file would be at output/org/javaguy/coolframework/MyClass.class (along with every other file in that package). In order to get to that file, your path would simply need to contain the folder 'output', not the whole package structure, since your import statement provides all that information to the VM.  
  
Now let's suppose that you bundle CoolFramework up into a .jar file, and put that CoolFramework.jar into a lib directory in your project. You would now need to put lib/CoolFramework.jar into your classpath. The VM will look inside the jar file for the org/javaguy/coolframework part, and find your class.  
  
So, classpaths contain:  
  
JAR files, and  
Paths to the top of package hierarchies.  
How do you set your classpath?  
  
The first way everyone seems to learn is with environment variables. On a unix machine, you can say something like:  
  
export CLASSPATH=/home/myaccount/myproject/lib/CoolFramework.jar:/home/myaccount/myproject/output/  
On a Windows machine you have to go to your environment settings and either add or modify the value that is already there.  
  
The second way is to use the -cp parameter when starting Java, like this:  
  
java -cp "/home/myaccount/myproject/lib/CoolFramework.jar:/home/myaccount/myproject/output/" MyMainClass  
A variant of this is the third way which is often done with a .sh or .bat file that calculates the classpath and passes it to Java via the -cp parameter.  
  
So what's the best way to do it?  
  
Setting stuff globally via environment variables is bad, generally for the same kinds of reasons that global variables are bad. You change the CLASSPATH environment variable so one program works, and you end up breaking another program.  
  
The -cp is the way to go. I generally make sure my CLASSPATH environment variable is an empty string where I develop, whenever possible, so that I avoid global classpath issues (some tools aren't happy when the global classpath is empty though - I know of two common, mega-thousand dollar licensed J2EE and Java servers that have this kind of issue with their command-line tools).

//////////////////////////////////////////////////////////////////////

HTTP SESSION JAX RS

@POST public String getData(Postdata postdata, @Context HttpServletRequest request) { HttpSession session = request.getSession(); }

Or

@Path("/test") public class TestResource { @Context private HttpServletRequest request; @POST public String getData(Postdata postdata) { HttpSession session = request.getSession(); } }

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Spring Security Hello World Annotation Example  
Http session

By mkyong | April 3, 2014 | Viewed : 396,589 | +894 pv/w  
security  
In preview post, we are using XML files to configure the Spring Security in a Spring MVC environment. In this tutorial, we are going to show you how to convert the previous XML-base Spring Security project into a pure Spring annotation project.  
  
Technologies used :  
  
Spring 3.2.8.RELEASE  
Spring Security 3.2.3.RELEASE  
Eclipse 4.2  
JDK 1.6  
Maven 3  
Tomcat 7 (Servlet 3.x)  
Few Notes  
  
This tutorial is using WebApplicationInitializer to load the Spring Context Loader automatically, which is supported in Servlet 3.x container only, for example, Tomcat 7 and Jetty 8.  
Since we are using WebApplicationInitializer, the web.xml file is NOT required.  
Spring Security annotations are supported in older Servlet 2.x container, for example, Tomcat 6. If you use the classic XML file to load the Spring context, this tutorial is still able to deploy on Servlet 2.x container, for example, Tomcat 6  
1. Project Demo  
See how it works.  
  
  
  
   
2. Directory Structure  
Review the final directory structure of this tutorial.  
  
spring-security-helloworld-annotation-directory  
  
   
3. Spring Security Dependencies  
To use Spring security, you need spring-security-web and spring-security-config.  
  
pom.xml  
 <properties>  
 <jdk.version>1.6</jdk.version>  
 <spring.version>3.2.8.RELEASE</spring.version>  
 <spring.security.version>3.2.3.RELEASE</spring.security.version>  
 <jstl.version>1.2</jstl.version>  
 </properties>  
  
 <dependencies>  
  
 <!-- Spring 3 dependencies -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-web</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
  
 <!-- Spring Security -->  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-web</artifactId>  
 <version>${spring.security.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-config</artifactId>  
 <version>${spring.security.version}</version>  
 </dependency>  
  
 <!-- jstl for jsp page -->  
 <dependency>  
 <groupId>jstl</groupId>  
 <artifactId>jstl</artifactId>  
 <version>${jstl.version}</version>  
 </dependency>  
  
 </dependencies>  
Copy  
4. Spring MVC Web Application  
A simple controller :  
  
If URL = /welcome or / , return hello page.  
If URL = /admin , return admin page.  
If URL = /dba , return admin page.  
Later, we will secure the /admin and /dba URLs.  
  
HelloController.java  
package com.mkyong.web.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.servlet.ModelAndView;  
  
@Controller  
public class HelloController {  
  
 @RequestMapping(value = { "/", "/welcome\*\*" }, method = RequestMethod.GET)  
 public ModelAndView welcomePage() {  
  
 ModelAndView model = new ModelAndView();  
 model.addObject("title", "Spring Security Hello World");  
 model.addObject("message", "This is welcome page!");  
 model.setViewName("hello");  
 return model;  
  
 }  
  
 @RequestMapping(value = "/admin\*\*", method = RequestMethod.GET)  
 public ModelAndView adminPage() {  
  
 ModelAndView model = new ModelAndView();  
 model.addObject("title", "Spring Security Hello World");  
 model.addObject("message", "This is protected page - Admin Page!");  
 model.setViewName("admin");  
  
 return model;  
  
 }  
  
 @RequestMapping(value = "/dba\*\*", method = RequestMethod.GET)  
 public ModelAndView dbaPage() {  
  
 ModelAndView model = new ModelAndView();  
 model.addObject("title", "Spring Security Hello World");  
 model.addObject("message", "This is protected page - Database Page!");  
 model.setViewName("admin");  
  
 return model;  
  
 }  
  
}  
Copy  
Two JSP pages.  
  
hello.jsp  
<%@page session="false"%>  
<html>  
<body>  
 <h1>Title : ${title}</h1>   
 <h1>Message : ${message}</h1>   
</body>  
</html>  
Copy  
admin.jsp  
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>  
<%@page session="true"%>  
<html>  
<body>  
 <h1>Title : ${title}</h1>  
 <h1>Message : ${message}</h1>  
  
 <c:if test="${pageContext.request.userPrincipal.name != null}">  
 <h2>Welcome : ${pageContext.request.userPrincipal.name}   
 | <a href="<c:url value="/logout" />" > Logout</a></h2>   
 </c:if>  
</body>  
</html>  
Copy  
5. Spring Security Configuration  
5.1 Create a Spring Security configuration file, and annotated with @EnableWebSecurity  
  
SecurityConfig.java  
package com.mkyong.config;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
  
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Autowired  
 public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication().withUser("mkyong").password("123456").roles("USER");  
 auth.inMemoryAuthentication().withUser("admin").password("123456").roles("ADMIN");  
 auth.inMemoryAuthentication().withUser("dba").password("123456").roles("DBA");  
 }  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
  
 http.authorizeRequests()  
 .antMatchers("/admin/\*\*").access("hasRole('ROLE\_ADMIN')")  
 .antMatchers("/dba/\*\*").access("hasRole('ROLE\_ADMIN') or hasRole('ROLE\_DBA')")  
 .and().formLogin();  
   
 }  
}  
Copy  
The equivalent of the Spring Security xml file :  
  
 <http auto-config="true">  
 <intercept-url pattern="/admin\*\*" access="ROLE\_ADMIN" />  
 <intercept-url pattern="/dba\*\*" access="ROLE\_ADMIN,ROLE\_DBA" />  
 </http>  
  
 <authentication-manager>  
 <authentication-provider>  
 <user-service>  
 <user name="mkyong" password="123456" authorities="ROLE\_USER" />  
 <user name="admin" password="123456" authorities="ROLE\_ADMIN" />  
 <user name="dba" password="123456" authorities="ROLE\_DBA" />  
 </user-service>  
 </authentication-provider>  
 </authentication-manager>  
Copy  
5.2 Create a class extends AbstractSecurityWebApplicationInitializer, it will load the springSecurityFilterChain automatically.  
  
SpringSecurityInitializer.java  
package com.mkyong.config.core;  
  
import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;  
  
public class SpringSecurityInitializer extends AbstractSecurityWebApplicationInitializer {  
 //do nothing  
}  
Copy  
The equivalent of Spring Security in web.xml file :  
  
 <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>  
Copy  
6. Spring MVC Configuration  
6.1 A Config class, define the view’s technology and imports above SecurityConfig.java.  
  
AppConfig.java  
package com.mkyong.config;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Import;  
import org.springframework.web.servlet.config.annotation.EnableWebMvc;  
import org.springframework.web.servlet.view.InternalResourceViewResolver;  
import org.springframework.web.servlet.view.JstlView;  
  
@EnableWebMvc  
@Configuration  
@ComponentScan({ "com.mkyong.web.\*" })  
@Import({ SecurityConfig.class })  
public class AppConfig {  
  
 @Bean  
 public InternalResourceViewResolver viewResolver() {  
 InternalResourceViewResolver viewResolver   
 = new InternalResourceViewResolver();  
 viewResolver.setViewClass(JstlView.class);  
 viewResolver.setPrefix("/WEB-INF/pages/");  
 viewResolver.setSuffix(".jsp");  
 return viewResolver;  
 }  
   
}  
Copy  
The equivalent of the Spring XML file :  
  
 <context:component-scan base-package="com.mkyong.web.\*" />  
  
 <bean  
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix">  
 <value>/WEB-INF/pages/</value>  
 </property>  
 <property name="suffix">  
 <value>.jsp</value>  
 </property>  
 </bean>  
Copy  
6.2 Create a Initializer class, to load everything.  
  
SpringMvcInitializer.java  
package com.mkyong.config.core;  
  
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;  
import com.mkyong.config.AppConfig;  
  
public class SpringMvcInitializer   
 extends AbstractAnnotationConfigDispatcherServletInitializer {  
  
 @Override  
 protected Class<?>[] getRootConfigClasses() {  
 return new Class[] { AppConfig.class };  
 }  
  
 @Override  
 protected Class<?>[] getServletConfigClasses() {  
 return null;  
 }  
  
 @Override  
 protected String[] getServletMappings() {  
 return new String[] { "/" };  
 }  
   
}  
Copy  
Done.  
  
Note  
In Servlet 3.x container environment + Spring container will detect and loads the Initializer classes automatically.  
7. Demo  
7.1. Welcome Page – http://localhost:8080/spring-security-helloworld-annotation/welcome  
  
spring-security-helloworld-annotation-welcome  
7.2 Try to access /admin page, Spring Security will intercept the request and redirect to /login, and a default login form is displayed.  
  
spring-security-helloworld-annotation-login  
7.3. If username and password is incorrect, error messages will be displayed, and Spring will redirect to this URL /login?error.  
  
spring-security-helloworld-annotation-login-error  
7.4. If username and password is correct, Spring will redirect the request to the original requested URL and display the page.  
  
spring-security-helloworld-annotation-admin  
7.5. For unauthorized user, Spring will display the 403 access denied page. For example, user “mkyong” or “dba” try to access the /admin URL.  
  
spring-security-helloworld-annotation-403

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

AUTHENTICATION VS AUTHERIZATION WEBS ERVICE

There is indeed a fundamental difference. Authentication is the mechanism whereby systems may securely identify their users. Authentication systems seek to provide answers to the questions:

* Who is the user?
* Is the user really who he/she represents himself to be?

Authorization, by contrast, is the mechanism by which a system determines what level of access a particular (authenticated) user should have to resources controlled by the system. For an example that may or may not be related to a web-based scenario, a database management system might be designed so as to provide certain specified individuals with the ability to retrieve information from a database but not the ability to change data stored in the database, while giving other individuals the ability to change data. Authorization systems provide answers to the questions:

* Is user X authorized to access resource R?
* Is user X authorized to perform operation P?
* Is user X authorized to perform operation P on resource R?

//////////////////////////////////////////////

HTTP SESSION FOR SPRING BASED APPLICATION

As others have suggested, you can use Spring security. Or if you don't want to deal with the complexities of Spring Security, you can get HttpSession object in your controller's handlers' methods' arguments. You can set values or objects in that session using HttpSession.setAttribute("name you want to refer to", actual value or object) once a user logs in. And when a user presses logout, you can use HttpSession.invalidate(); to finish the session.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////Multiple request in webs eervice at sametime

A web service is, essentially, an XML request for a method of a class to be

invoked.

If more than one client were to call for the same web service at the same

time, it would simply be a matter of having 2 instances of the same object

created in memory simultaneously.

Assuming, you are not using a Single Threaded Application (STA), the two (or

more) calls to the web service can execute virtually at the same time.

It all depends what server you use and how it is configured. Standard configuration (you have to work hard to make it not standard) is to have multiple threads. In other words - server usually automatically creates or uses another thread for each new request and it is almost certain that it will be processed in parallel.

You can actually see it inside your running code by using java.lang.Thread.currentThread() - print the name of current thread and Rest request and you will see

//////////////////////////////////////////////////////////////////////////////////////////////////

HOW MULTIPLE REQUEST ARE HANDLED IN HTTP SERVER

The short answer is that the server has a thread pool to execute the requests on

But to understand this correctly, you need to know how sockets work and threading and file IO works. Be careful, you are peeling the onion here, and if you are not used to it, you will cry

Before understanding how servlet works, you need to understand how HTTP requests work. HTTP requests are sent by a client to the server. The server sends back a response. The response could be HTML, or JSON, or whatever the server decides to send. THe internal mechanism of this is such:

a) The server is listening on a socket on a certain port

b) THe client connects to the port and tells the server that it's ready to communicate and the server acknowledges

c) The server then serves the request and sends back the response.

When the web was envisioned, it was envisioned as static. ie; the content being sent would all be stored in files. All the HTTP server would do is match the URL to the file on the local storage and send back the file. So, the question is how does the server send multiple files at the same time? Back then, the answer was entirely dependent on whether the storage could serve multiple files at the same time, and/or do some sort of asynchronous IO. If all you are doing is serving files, then you are restricted to how fast your disks can serve files. So, when you got a request, you immediately send an async request to the OS for the contents of the file, and hold the socket in memory. When the OS responds with the contents of the file, you simply dump the contents of the file into your socket. Done!! You have neatly delegated the problem of serving multiple requests to the file system. Your HTTP server is just a marshalling/unmarshalling layer

Pretty quickly, someone figured out:- hey, instead of serving a file, we could call a program and pass the URL, and the program can generate the HTML and we can send the generated HTML back. This allows us to serve dynamic content from web pages. This was called CGI, and essentially servlet is derived from CGI. SO, back then the way CGI worked was that each request was it's own process. Essentially, the HTTP server worked the same way as it did when it served files. Except that instead of sending request for File IO, it forks a process. Remember that the great and powerful Unix treats everything as a stream, whether it's a file or the output of a process. So, the HTTP server basically just waited for the OS to respond back with the output of the program, and it dumped the output back to the response. Multiple requests resulted in multiple processes being forked. And the OS handled all the stuff about executing programs in parallel

The problem with this was every request forced an execution, and forking of execution is not cheap. The OS needs to do a lot of housekeeping, plus there is no control over how many requests will be running at the same time. ALso, the programs had to initialize themselves on every request. They had to open DB connections, blah blah

So, the way J2EE solved this problem was by specifying that servlets will be executed in the same process space as the container. It called the container as web server instead of HTTP server. But, it still needed to be able to support multiple requests, and since it was not forking processes anymore, it had to do something else to support multiple requests. The J2EE spec doesn't specify how web servers should handle multiple requests. It just tells them that they should. Most web servers do it using a thread pool

A thread pool works on a producer/consumer pattern. You have a bunch of consumer threads sitting there sleeping. Another producer thread puts a requrest in the queue. As soon as a request is put in, a consumer wakes up takes the request and works on it and puts a response into an outbound queue. The web server works on this principle. The producer thread is the thread that is doing all the network IO for HTTP requests. As soon as it gets the request, it puts the entire request into a queue. A consumer thread wakes up, figures out which servlet and filter chain to call, calls it, and puts the response back in the outbound queue. The NIO thread picks the response and sends it back to the client

\*Pant\* \*Pant\*

If you are familiar with any of the popular web servers, you will know that there are ways you can configure the size of the thread pool and the size of the queue. Usually, if the queue is full, the NIO thread will send a 503 - Service unavailable error

HIBERNATE

////////////////////////////////////////////////////////L//////////////////////////////////////////

GENERATION type Strategy Hibernate

1. Introduction  
Identifiers in Hibernate represent the primary key of an entity. This implies the values are unique so that they can identify a specific entity, that they aren’t null and that they won’t be modified.  
  
Hibernate provides a few different ways to define identifiers. In this article, we’ll review each method of mapping entity ids using the library.  
  
2. Simple Identifiers  
The most straightforward way to define an identifier is by using the @Id annotation.  
  
Simple ids are mapped using @Id to a single property of one of these types: Java primitive and primitive wrapper types, String, Date, BigDecimal, BigInteger.  
  
Let’s see a quick example of defining an entity with a primary key of type long:  
  
  
@Entity  
public class Student {  
   
 @Id  
 private long studentId;  
   
 // standard constructor, getters, setters  
}  
3. Generated Identifiers  
If we want the primary key value to be generated automatically for us, we can add the @GeneratedValue annotation.  
  
This can use 4 generation types: AUTO, IDENTITY, SEQUENCE, TABLE.  
  
If we don’t specify a value explicitly, the generation type defaults to AUTO.  
  
3.1. AUTO Generation  
If we’re using the default generation type, the persistence provider will determine values based on the type of the primary key attribute. This type can be numerical or UUID.  
  
For numeric values, the generation is based on a sequence or table generator, while UUID values will use the UUIDGenerator.  
  
Let’s see an example of mapping an entity primary key using AUTO generation strategy:  
  
@Entity  
public class Student {  
   
 @Id  
 @GeneratedValue  
 private long studentId;  
   
 // ...  
}  
In this case, the primary key values will be unique at the database level.  
  
An interesting feature introduced in Hibernate 5 is the UUIDGenerator. To use this, all we need to do is declare an id of type UUID with @GeneratedValue annotation:  
  
  
@Entity  
public class Course {  
   
 @Id  
 @GeneratedValue  
 private UUID courseId;  
   
 // ...  
}  
Hibernate will generate an id of the form “8dd5f315-9788-4d00-87bb-10eed9eff566”.  
  
3.2. IDENTITY Generation  
This type of generation relies on the IdentityGenerator which expects values generated by an identity column in the database, meaning they are auto-incremented.  
  
To use this generation type, we only need to set the strategy parameter:  
  
@Entity  
public class Student {  
   
 @Id  
 @GeneratedValue (strategy = GenerationType.IDENTITY)  
 private long studentId;  
   
 // ...  
}  
One thing to note is that IDENTITY generation disables batch updates.  
  
3.3. SEQUENCE Generation  
To use a sequence-based id, Hibernate provides the SequenceStyleGenerator class.  
  
This generator uses sequences if they’re supported by our database, and switches to table generation if they aren’t.  
  
To customize the sequence name, we can use the JPA @SequenceGenerator annotation:  
  
  
@Entity  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.SEQUENCE,   
 generator = "sequence-generator")  
 @SequenceGenerator(name = "sequence-generator",   
 sequenceName = "user\_sequence", initialValue = 4)  
 private long userId;  
   
 // ...  
}  
In this example, we’ve also set an initial value for the sequence, which means the primary key generation will start at 4.  
  
SEQUENCE is the generation type recommended by the Hibernate documentation.  
  
The generated values are unique per sequence. If you don’t specify a sequence name, Hibernate will re-use the same hibernate\_sequence for different types.  
  
3.4. TABLE Generation  
The TableGenerator uses an underlying database table that holds segments of identifier generation values.  
  
Let’s customize the table name using the @TableGenerator annotation:  
  
1  
2  
  
@Entity  
public class Department {  
 @Id  
 @GeneratedValue(strategy = GenerationType.TABLE,   
 generator = "table-generator")  
 @TableGenerator(name = "table-generator",   
 table = "dep\_ids",   
 pkColumnName = "seq\_id",   
 valueColumnName = "seq\_value")  
 private long depId;  
   
 // ...  
}  
In this example, we can see that other attributes such as the pkColumnName and valueColumnName can also be customized.  
  
The disadvantage of this method is that it doesn’t scale well and can negatively affect performance.  
  
To sum up, these four generation types will result in similar values being generated but use different database mechanisms.  
  
3.5. Custom Generator  
If we don’t want to use any of the out-of-the-box strategies, we can define our custom generator by implementing the IdentifierGenerator interface.  
  
Let’s create a generator that builds identifiers containing a String prefix and a number:  
  
public class MyGenerator   
 implements IdentifierGenerator, Configurable {  
   
 private String prefix;  
   
 @Override  
 public Serializable generate(  
 SharedSessionContractImplementor session, Object obj)   
 throws HibernateException {  
   
 String query = String.format("select %s from %s",   
 session.getEntityPersister(obj.getClass().getName(), obj)  
 .getIdentifierPropertyName(),  
 obj.getClass().getSimpleName());  
   
 Stream ids = session.createQuery(query).stream();  
   
 Long max = ids.map(o -> o.replace(prefix + "-", ""))  
 .mapToLong(Long::parseLong)  
 .max()  
 .orElse(0L);  
   
 return prefix + "-" + (max + 1);  
 }  
   
 @Override  
 public void configure(Type type, Properties properties,   
 ServiceRegistry serviceRegistry) throws MappingException {  
 prefix = properties.getProperty("prefix");  
 }  
}  
In this example, we override the generate() method from the IdentifierGenerator interface and first find the highest number from the existing primary keys of the form prefix-XX.  
  
Then we add 1 to the maximum number found and append the prefix property to obtain the newly generated id value.  
  
Our class also implements the Configurable interface, so that we can set the prefix property value in the configure() method.  
  
Next, let’s add this custom generator to an entity. For this, we can use the @GenericGenerator annotation with a strategy parameter that contains the full class name of our generator class:  
  
  
@Entity  
public class Product {  
   
 @Id  
 @GeneratedValue(generator = "prod-generator")  
 @GenericGenerator(name = "prod-generator",   
 parameters = @Parameter(name = "prefix", value = "prod"),   
 strategy = "com.baeldung.hibernate.pojo.generator.MyGenerator")  
 private String prodId;  
   
 // ...  
}  
Also, notice we’ve set the prefix parameter to “prod”.  
  
Let’s see a quick JUnit test for a clearer understanding of the id values generated:  
  
  
@Test  
public void whenSaveCustomGeneratedId\_thenOk() {  
 Product product = new Product();  
 session.save(product);  
 Product product2 = new Product();  
 session.save(product2);  
   
 assertThat(product2.getProdId()).isEqualTo("prod-2");  
}  
Here, the first value generated using the “prod” prefix was “prod-1”, followed by “prod-2”.  
  
4. Composite Identifiers  
Besides the simple identifiers we’ve seen so far, Hibernate also allows us to define composite identifiers.  
  
A composite id is represented by a primary key class with one or more persistent attributes.  
  
The primary key class must fulfill several conditions:  
  
it should be defined using @EmbeddedId or @IdClass annotations  
it should be public, serializable and have a public no-arg constructor  
it should implement equals() and hashCode() methods  
The class’s attributes can be basic, composite or ManyToOne while avoiding collections and OneToOne attributes.  
  
4.1. @EmbeddedId  
To define an id using @EmbeddedId, first we need a primary key class annotated with @Embeddable:  
  
  
@Embeddable  
public class OrderEntryPK implements Serializable {  
   
 private long orderId;  
 private long productId;  
   
 // standard constructor, getters, setters  
 // equals() and hashCode()   
}  
Next, we can add an id of type OrderEntryPK to an entity using @EmbeddedId:  
  
  
@Entity  
public class OrderEntry {  
   
 @EmbeddedId  
 private OrderEntryPK entryId;  
   
 // ...  
}  
Let’s see how we can use this type of composite id to set the primary key for an entity:  
  
  
@Test  
public void whenSaveCompositeIdEntity\_thenOk() {  
 OrderEntryPK entryPK = new OrderEntryPK();  
 entryPK.setOrderId(1L);  
 entryPK.setProductId(30L);  
   
 OrderEntry entry = new OrderEntry();  
 entry.setEntryId(entryPK);  
 session.save(entry);  
   
 assertThat(entry.getEntryId().getOrderId()).isEqualTo(1L);  
}  
Here the OrderEntry object has an OrderEntryPK primary id formed of two attributes: orderId and productId.  
  
4.2. @IdClass  
The @IdClass annotation is similar to the @EmbeddedId, except the attributes are defined in the main entity class using @Id for each one.  
  
The primary-key class will look the same as before.  
  
Let’s rewrite the OrderEntry example with an @IdClass:  
  
  
@Entity  
@IdClass(OrderEntryPK.class)  
public class OrderEntry {  
 @Id  
 private long orderId;  
 @Id  
 private long productId;  
   
 // ...  
}  
Then we can set the id values directly on the OrderEntry object:  
  
  
@Test  
public void whenSaveIdClassEntity\_thenOk() {   
 OrderEntry entry = new OrderEntry();  
 entry.setOrderId(1L);  
 entry.setProductId(30L);  
 session.save(entry);  
   
 assertThat(entry.getOrderId()).isEqualTo(1L);  
}  
Note that for both types of composite ids, the primary key class can also contain @ManyToOne attributes.  
  
Hibernate also allows defining primary-keys made up of @ManyToOne associations combined with @Id annotation. In this case, the entity class should also fulfill the conditions of a primary-key class.  
  
The disadvantage of this method is that there’s no separation between the entity object and the identifier.  
  
5. Derived Identifiers  
Derived identifiers are obtained from an entity’s association using the @MapsId annotation.  
  
First, let’s create a UserProfile entity which derives its id from a one-to-one association with the User entity:  
  
  
@Entity  
public class UserProfile {  
   
 @Id  
 private long profileId;  
   
 @OneToOne  
 @MapsId  
 private User user;  
   
 // ...  
}  
Next, let’s verify that a UserProfile instance has the same id as its associated User instance:  
@Test  
public void whenSaveDerivedIdEntity\_thenOk() {   
 User user = new User();  
 session.save(user);  
   
 UserProfile profile = new UserProfile();  
 profile.setUser(user);  
 session.save(profile);  
   
 assertThat(profile.getProfileId()).isEqualTo(user.getUserId());  
}  
6. Conclusion  
In this article, we’ve seen the multiple ways we can define identifiers in Hibernate.

///////////////////////////////////////////////////////////////////////////////////////////////////////

ONETOMANY

# Hibernate One-to-Many Mapping Using Annotations

November 17, 2012 by Lokesh Gupta

One to many mapping is made between two entities where first entity can have relation with multiple second entity instances but second can be associated with only one instance of first entity. Its 1 to n relationship. For example, in any company an employee can register multiple bank accounts but one bank account will be associated with one and only one employee. In this post, we will learn to make such mapping in database using hibernate 3

Problem statement

We have to write two entities i.e. EmployeeEntity and AccountEntity such that multiple accounts can be associated with a single employee, but these accounts can not be shared between two or more employees.

### Designing the solution

This problem can be solved in two different ways. One is to have a foreign key column in account table i.i EMPLOYEE\_ID. This column will refer to primary key of Employee table. This way no two accounts can be associated with multiple employees. Obviously, account number needs to be unique for enforcing this restriction.

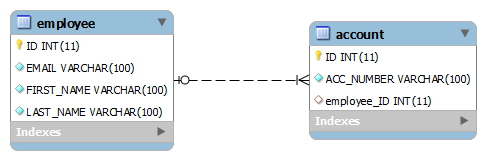
Another approach is to have a common join table lets say EMPLOYEE\_ACCOUNT. This table will have two column i.e. EMP\_ID which will be foreign key referring to primary key in EMPLOYEE table and similarly ACCOUNT\_ID which will be foreign key referring to primary key of ACCOUNT table.

Lets write some code for both approaches.

## Using foreign key association

In this approach, both entity will be responsible for making the relationship and maintaining it. EmployeeEntity should declare that relationship is One to many, and AccountEntity should declare that relationship from its end is many to one.

Lets first see the schema design.



Lets write EmployeeEntity.java

|  |
| --- |
| package hibernate.test.oneToMany.foreignKeyAsso;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ForeignKeyAssoEntity")  @Table(name = "Employee", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID"),  @UniqueConstraint(columnNames = "EMAIL") })  public class EmployeeEntity implements Serializable {    private static final long serialVersionUID = -1798070786993154676L;    @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer employeeId;    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)  private String email;    @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)  private String firstName;    @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)  private String lastName;    @OneToMany(cascade=CascadeType.ALL)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts;    public Integer getEmployeeId() {  return employeeId;  }    public void setEmployeeId(Integer employeeId) {  this.employeeId = employeeId;  }    public String getEmail() {  return email;  }    public void setEmail(String email) {  this.email = email;  }    public String getFirstName() {  return firstName;  }    public void setFirstName(String firstName) {  this.firstName = firstName;  }    public String getLastName() {  return lastName;  }    public void setLastName(String lastName) {  this.lastName = lastName;  }    public Set<AccountEntity> getAccounts() {  return accounts;  }    public void setAccounts(Set<AccountEntity> accounts) {  this.accounts = accounts;  }  } |

Write AccountEntity.java

|  |
| --- |
| package hibernate.test.oneToMany.foreignKeyAsso;    import java.io.Serializable;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToOne;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ForeignKeyAssoAccountEntity")  @Table(name = "ACCOUNT", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class AccountEntity implements Serializable  {    private static final long serialVersionUID = -6790693372846798580L;    @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer accountId;    @Column(name = "ACC\_NUMBER", unique = true, nullable = false, length = 100)  private String accountNumber;    @ManyToOne  private EmployeeEntity employee;    public Integer getAccountId() {  return accountId;  }    public void setAccountId(Integer accountId) {  this.accountId = accountId;  }    public String getAccountNumber() {  return accountNumber;  }    public void setAccountNumber(String accountNumber) {  this.accountNumber = accountNumber;  }    public EmployeeEntity getEmployee() {  return employee;  }    public void setEmployee(EmployeeEntity employee) {  this.employee = employee;  }  } |

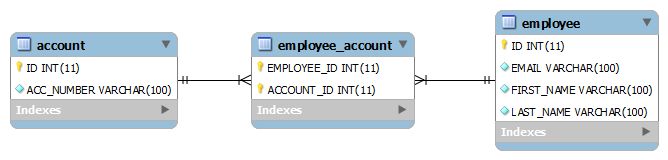
Lets Test the code:

|  |
| --- |
| package hibernate.test.oneToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.oneToMany.foreignKeyAsso.AccountEntity;  import hibernate.test.oneToMany.foreignKeyAsso.EmployeeEntity;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestForeignKeyAssociation  {    public static void main(String[] args)  {  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    AccountEntity account1 = new AccountEntity();  account1.setAccountNumber("Account detail 1");    AccountEntity account2 = new AccountEntity();  account2.setAccountNumber("Account detail 2");    AccountEntity account3 = new AccountEntity();  account3.setAccountNumber("Account detail 3");    //Add new Employee object  EmployeeEntity firstEmployee = new EmployeeEntity();  firstEmployee.setEmail("demo-user-first@mail.com");  firstEmployee.setFirstName("demo-one");  firstEmployee.setLastName("user-one");    EmployeeEntity secondEmployee = new EmployeeEntity();  secondEmployee.setEmail("demo-user-second@mail.com");  secondEmployee.setFirstName("demo-two");  secondEmployee.setLastName("user-two");    Set<AccountEntity> accountsOfFirstEmployee = new HashSet<AccountEntity>();  accountsOfFirstEmployee.add(account1);  accountsOfFirstEmployee.add(account2);    Set<AccountEntity> accountsOfSecondEmployee = new HashSet<AccountEntity>();  accountsOfSecondEmployee.add(account3);    firstEmployee.setAccounts(accountsOfFirstEmployee);  secondEmployee.setAccounts(accountsOfSecondEmployee);  //Save Employee  session.save(firstEmployee);  session.save(secondEmployee);    session.getTransaction().commit();  HibernateUtil.shutdown();  }  }  Output:    Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=? |

### Using a join table

This approach uses a join table to store the associations between account and employee entities. @JoinTable annotation has been used to make this association.

Lets see how the database schema will look like:

**one To Many association in hibernate using join table**

EmployeeEntity.java

|  |
| --- |
| package hibernate.test.oneToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "JoinTableEmployeeEntity")  @Table(name = "Employee", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID"),  @UniqueConstraint(columnNames = "EMAIL") })  public class EmployeeEntity implements Serializable  {  private static final long serialVersionUID = -1798070786993154676L;    @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer employeeId;    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)  private String email;    @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)  private String firstName;    @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)  private String lastName;    @OneToMany(cascade=CascadeType.ALL)  @JoinTable(name="EMPLOYEE\_ACCOUNT", joinColumns={@JoinColumn(name="EMPLOYEE\_ID", referencedColumnName="ID")}  , inverseJoinColumns={@JoinColumn(name="ACCOUNT\_ID", referencedColumnName="ID")})  private Set<AccountEntity> accounts;    public Integer getEmployeeId() {  return employeeId;  }    public void setEmployeeId(Integer employeeId) {  this.employeeId = employeeId;  }    public String getEmail() {  return email;  }    public void setEmail(String email) {  this.email = email;  }    public String getFirstName() {  return firstName;  }    public void setFirstName(String firstName) {  this.firstName = firstName;  }    public String getLastName() {  return lastName;  }    public void setLastName(String lastName) {  this.lastName = lastName;  }    public Set<AccountEntity> getAccounts() {  return accounts;  }    public void setAccounts(Set<AccountEntity> accounts) {  this.accounts = accounts;  }  } |

AccountEntity.java

|  |
| --- |
| package hibernate.test.oneToMany.joinTable;    import java.io.Serializable;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "JoinTableAccountEntity")  @Table(name = "ACCOUNT", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class AccountEntity implements Serializable  {    private static final long serialVersionUID = -6790693372846798580L;    @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer accountId;    @Column(name = "ACC\_NUMBER", unique = true, nullable = false, length = 100)  private String accountNumber;    public Integer getAccountId() {  return accountId;  }    public void setAccountId(Integer accountId) {  this.accountId = accountId;  }    public String getAccountNumber() {  return accountNumber;  }    public void setAccountNumber(String accountNumber) {  this.accountNumber = accountNumber;  }  } |

Configuring entities in config file

We have available both entities to runtime, we have to add them in hibernate.cfg.xml file. Please note that only one set of entities should be configured in configuration file otherwise unexpected results can occur.

|  |
| --- |
| < ?xml version="1.0" encoding="utf-8"?>  < !DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "<http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>">  <hibernate-configuration>  <session-factory>  <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>  <propertyname="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatetest</property>  <property name="hibernate.connection.password">XXXXXX</property>  <property name="hibernate.connection.username">root</property>  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  <property name="show\_sql">true</property>  <property name="hbm2ddl.auto">create</property>  <mapping clas="hibernate.test.oneToMany.foreignKeyAsso.AccountEntity"></mapping>  <mapping clas="hibernate.test.oneToMany.foreignKeyAsso.EmployeeEntity"></mapping>  </session-factory>  </hibernate-configuration> |

Testing the code

Now, its time to test the code. I have written following code to test above entities.

|  |
| --- |
| package hibernate.test.oneToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.oneToMany.joinTable.AccountEntity;  import hibernate.test.oneToMany.joinTable.EmployeeEntity;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestJoinTable  {  public static void main(String[] args)  {  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    AccountEntity account1 = new AccountEntity();  account1.setAccountNumber("123-345-65454");    AccountEntity account2 = new AccountEntity();  account2.setAccountNumber("123-345-6542222");    //Add new Employee object  EmployeeEntity emp = new EmployeeEntity();  emp.setEmail("demo-user@mail.com");  emp.setFirstName("demo");  emp.setLastName("user");    Set<AccountEntity> accounts = new HashSet<AccountEntity>();  accounts.add(account1);  accounts.add(account2);    emp.setAccounts(accounts);  //Save Employee  session.save(emp);    session.getTransaction().commit();  HibernateUtil.shutdown();  }  }    Output:    Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into EMPLOYEE\_ACCOUNT (EMPLOYEE\_ID, ACCOUNT\_ID) values (?, ?)  Hibernate: insert into EMPLOYEE\_ACCOUNT (EMPLOYEE\_ID, ACCOUNT\_ID) values (?, ?) |

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

# Hibernate Many-to-Many Mapping Using Annotations

November 17, 2012 by Lokesh Gupta

Many to many mapping is made between two entities where one can have relation with multiple other entity instances. For example, for a subscription service SubscriptionEntity and ReaderEntity can be two type of entities. Any subscription can have multiple readers, where a reader can subscribe to multiple subscriptions. In this post, we will learn to make such mapping in database using

Proposed solution

To demonstrate many to many mapping, we will associate two entities i.e. ReaderEntity and SubscriptionEntity. Their database schema should look like this. Using these tables, any application can save multiple associations between readers and subscriptions.



## Writing owner entity

Owner entity is the entity which is responsible make making the association and maintaining it. In our case, I am making ReaderEntity the owner entity. @JoinTable annotation has been used to make this association.

|  |
| --- |
| package hibernate.test.manyToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ReaderEntity")  @Table(name = "READER", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID"),  @UniqueConstraint(columnNames = "EMAIL") })  public class ReaderEntity implements Serializable {    private static final long serialVersionUID = -1798070786993154676L;    @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer readerId;    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)  private String email;    @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)  private String firstName;    @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)  private String lastName;    @ManyToMany(cascade=CascadeType.ALL)  @JoinTable(name="READER\_SUBSCRIPTIONS", joinColumns={@JoinColumn(referencedColumnName="ID")}  , inverseJoinColumns={@JoinColumn(referencedColumnName="ID")})  private Set<SubscriptionEntity> subscriptions;    public Integer getReaderId() {  return readerId;  }    public void setReaderId(Integer readerId) {  this.readerId = readerId;  }    public String getEmail() {  return email;  }    public void setEmail(String email) {  this.email = email;  }    public String getFirstName() {  return firstName;  }    public void setFirstName(String firstName) {  this.firstName = firstName;  }    public String getLastName() {  return lastName;  }    public void setLastName(String lastName) {  this.lastName = lastName;  }    public Set<SubscriptionEntity> getSubscriptions() {  return subscriptions;  }    public void setSubscriptions(Set<SubscriptionEntity> subscriptions) {  this.subscriptions = subscriptions;  }  } |

### Writing mapped entity

Our mapped entity is SubscriptionEntity which is mapped to ReaderEntity using “mappedBy” attribute.

|  |
| --- |
| package hibernate.test.manyToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "SubscriptionEntity")  @Table(name = "SUBSCRIPTION", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class SubscriptionEntity implements Serializable  {    private static final long serialVersionUID = -6790693372846798580L;    @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer subscriptionId;    @Column(name = "SUBS\_NAME", unique = true, nullable = false, length = 100)  private String subscriptionName;    @ManyToMany(mappedBy="subscriptions")  private Set<ReaderEntity> readers;    public Integer getSubscriptionId() {  return subscriptionId;  }    public void setSubscriptionId(Integer subscriptionId) {  this.subscriptionId = subscriptionId;  }    public String getSubscriptionName() {  return subscriptionName;  }    public void setSubscriptionName(String subscriptionName) {  this.subscriptionName = subscriptionName;  }    public Set<ReaderEntity> getReaders() {  return readers;  }    public void setReaders(Set<ReaderEntity> readers) {  this.readers = readers;  }  } |

## Configuring entities in config file

We have available both entities to runtime, we have to add them in *hibernate.cfg.xml* file.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "<http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>">  <hibernate-configuration>  <session-factory>  <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>  <propertyname="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatetest</property>  <property name="hibernate.connection.password">XXXXXX</property>  <property name="hibernate.connection.username">root</property>  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  <property name="show\_sql">true</property>  <property name="hbm2ddl.auto">create</property>  <mapping class="hibernate.test.manyToMany.joinTable.ReaderEntity"/>  <mapping class="hibernate.test.manyToMany.joinTable.SubscriptionEntity"/>  </session-factory>  </hibernate-configuration> |

## Testing the code

Now, its time to test the code. I have written following code to test above entities.

|  |
| --- |
| package hibernate.test.manyToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.manyToMany.joinTable.\*;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestJoinTable  {    public static void main(String[] args)  {  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    //Add subscription  SubscriptionEntity subOne = new SubscriptionEntity();  subOne.setSubscriptionName("Entertainment");    SubscriptionEntity subTwo = new SubscriptionEntity();  subTwo.setSubscriptionName("Horror");    Set<SubscriptionEntity> subs = new HashSet<SubscriptionEntity>();  subs.add(subOne);  subs.add(subTwo);    //Add readers  ReaderEntity readerOne = new ReaderEntity();  readerOne.setEmail("demo-user1@mail.com");  readerOne.setFirstName("demo");  readerOne.setLastName("user");    ReaderEntity readerTwo = new ReaderEntity();  readerTwo.setEmail("demo-user2@mail.com");  readerTwo.setFirstName("demo");  readerTwo.setLastName("user");    Set<ReaderEntity> readers = new HashSet<ReaderEntity>();  readers.add(readerOne);  readers.add(readerTwo);    readerOne.setSubscriptions(subs);  readerTwo.setSubscriptions(subs);    session.save(readerOne);  session.save(readerTwo);    session.getTransaction().commit();  HibernateUtil.shutdown();  }  }    Output:    Hibernate: insert into READER (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into SUBSCRIPTION (SUBS\_NAME) values (?)  Hibernate: insert into SUBSCRIPTION (SUBS\_NAME) values (?)  Hibernate: insert into READER (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?) |

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

# Hibernate One-to-One Mapping Using Annotations

November 15, 2012 by Lokesh Gupta

If you are working on any hibernate project or you are planning to work on any in future, then you can easily understand the one-to-one relationships between several entities in your application. In this post, i will discuss variations of one-to-one mappings supported in hibernate.

[Download source code](https://drive.google.com/file/d/0B7yo2HclmjI4VHJnQk4tYjBueDA/view?usp=drive_web)

Sections in this post:  
Various supported techniques  
Using foreign key association  
Using a common join table  
Using shared primary key

For this article, I am extending the example written [for hello world example](https://howtodoinjava.com/hibernate/hibernate-3-introduction-and-writing-hello-world-application/). We have two entities here: *Employee* and *Account*.

## Various supported techniques

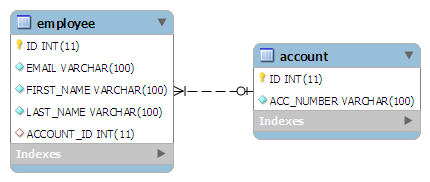
In hibernate there are 3 ways to create one-to-one relationships between two entities. Either way you have to use [@OneToOne](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html) annotation. *First technique* is most widely used and uses a foreign key column in one to table. *Second technique* uses a rather known solution of having a third table to store mapping between first two tables. *Third technique* is something new which uses a common primary key value in both the tables.

Lets see them in action one by one:

## Using foreign key association

In this association, a foreign key column is created in owner entity. For example, if we make EmployeeEntity owner, then a extra column “ACCOUNT\_ID” will be created in Employee table. This column will store the foreign key for Account table.

Table structure will be like this:



To make such association, refer the account entity in EmployeeEntity class as follow:

|  |
| --- |
| @OneToOne  @JoinColumn(name="ACCOUNT\_ID")  private AccountEntity account; |

The join column is declared with the [@JoinColumn](https://docs.oracle.com/javaee/5/api/javax/persistence/JoinColumn.html) annotation which looks like the [@Column](https://docs.oracle.com/javaee/5/api/javax/persistence/Column.html)annotation. It has one more parameters named referencedColumnName. This parameter declares the column in the targeted entity that will be used to the join.

If no @JoinColumn is declared on the owner side, the defaults apply. A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, \_ (underscore), and the name of the primary key column(s) in the owned side.

In a bidirectional relationship, one of the sides (and only one) has to be the owner: the owner is responsible for the association column(s) update. *To declare a side as not responsible for the relationship, the attribute* [*mappedBy*](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html#mappedBy%28%29) *is used*. mappedBy refers to the property name of the association on the owner side.

|  |
| --- |
| @OneToOne(mappedBy="account")  private EmployeeEntity employee; |

Above “mappedBy” attribute declares that it is dependent on owner entity for mapping.

Lets test above mappings in running code:

|  |
| --- |
| public class TestForeignKeyAssociation {    public static void main(String[] args) {  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    AccountEntity account = new AccountEntity();  account.setAccountNumber("123-345-65454");    // Add new Employee object  EmployeeEntity emp = new EmployeeEntity();  emp.setEmail("demo-user@mail.com");  emp.setFirstName("demo");  emp.setLastName("user");    // Save Account  session.saveOrUpdate(account);  // Save Employee  emp.setAccount(account);  session.saveOrUpdate(emp);    session.getTransaction().commit();  HibernateUtil.shutdown();  }  } |

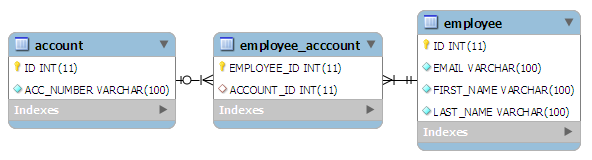
Running above code creates desired schema in database and run these SQL queries.

|  |
| --- |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?) |

You can verify the data and mappings in both tables when you run above program.. :-)

## Using a common join table

This approach is not new to all of us. Lets start with targeted DB structure in this technique.



In this technique, main annotation to be used is [@JoinTable](https://docs.oracle.com/javaee/5/api/javax/persistence/JoinTable.html). This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables. Lets see how it is used:

|  |
| --- |
| @OneToOne(cascade = CascadeType.ALL)  @JoinTable(name="EMPLOYEE\_ACCCOUNT", joinColumns = @JoinColumn(name="EMPLOYEE\_ID"),  inverseJoinColumns = @JoinColumn(name="ACCOUNT\_ID"))  private AccountEntity account; |

@JoinTable annotation is used in EmployeeEntity class. It declares that a new table EMPLOYEE\_ACCOUNT will be created with two columns EMPLOYEE\_ID (primary key of EMPLOYEE table) and ACCOUNT\_ID (primary key of ACCOUNT table).

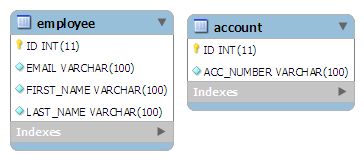
Testing above entities generates following SQL queries in log files:

|  |
| --- |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into EMPLOYEE\_ACCCOUNT (ACCOUNT\_ID, EMPLOYEE\_ID) values (?, ?) |

## Using shared primary key

In this technique, hibernate will ensure that it will use a common primary key value in both the tables. This way primary key of EmployeeEntity can safely be assumed the primary key of AccountEntity also.

Table structure will be like this:



In this approach, [@PrimaryKeyJoinColumn](https://docs.oracle.com/javaee/5/api/javax/persistence/PrimaryKeyJoinColumn.html) is the main annotation to be used.Let see how to use it.

|  |
| --- |
| @OneToOne(cascade = CascadeType.ALL)  @PrimaryKeyJoinColumn  private AccountEntity account; |

In AccountEntity side, it will remain dependent on owner entity for the mapping.

|  |
| --- |
| @OneToOne(mappedBy="account", cascade=CascadeType.ALL)  private EmployeeEntity employee; |

Testing above entities generates following SQL queries in log files:

|  |
| --- |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?) |

So, we have seen all 3 types of one to one mappings supported in hibernate. I will suggest you to download the source code and play with it.

//////////////////////////////////////////////////////////////////////////////////////////////////////////

1st level cache

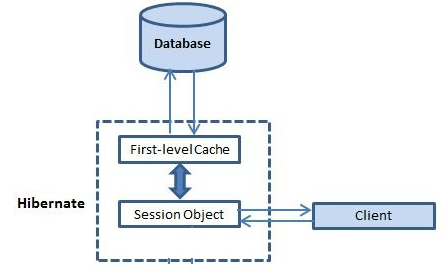
# Understanding Hibernate First Level Cache with Example

July 1, 2013 by Lokesh Gupta

Caching is a facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate achieves the second goal by implementing first level cache.

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that it is associated with Session object. As we know session object is created on demand from session factory and it is lost, once the session is closed. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is not accessible to any other session object in any other part of application.



## Important facts

1. First level cache is associated with “session” object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. First level cache is enabled by default and you can not disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
6. The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
7. The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

Lets verify above facts using examples.

## First level cache retrieval example

In this example, I am retrieving DepartmentEntity object from database using hibernate session. I will retrieve it multiple times, and will observe the sql logs to see the differences.

|  |
| --- |
| //Open the hibernate session  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    //fetch the department entity from database first time  DepartmentEntity department = (DepartmentEntity)  session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    session.getTransaction().commit();  HibernateUtil.shutdown();    Output:    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource  Human Resource |

As you can see that second “session.load()” statement does not execute select query again and load the department entity directly.

## First level cache retrieval example with new session

With new session, entity is fetched from database again irrespective of it is already present in any other session in application.

|  |
| --- |
| //Open the hibernate session  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    Session sessionTemp = HibernateUtil.getSessionFactory().openSession();  sessionTemp.beginTransaction();  try  {  //fetch the department entity from database first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    department = (DepartmentEntity) sessionTemp.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());  }  finally  {  session.getTransaction().commit();  HibernateUtil.shutdown();    sessionTemp.getTransaction().commit();  HibernateUtil.shutdown();  }    Output:    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource  Human Resource    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource |

You can see that even if the department entity was stored in “session” object, still another database query was executed when we use another session object “sessionTemp”.

## Removing cache objects from first level cache example

Though we can not disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods :

* evict()
* clear()

Here evict() is used to remove a particular object from cache associated with session, and clear() method is used to remove all cached objects associated with session. So they are essentially like remove one and remove all.

|  |
| --- |
| //Open the hibernate session  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();  try  {  //fetch the department entity from database first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    session.evict(department);  //session.clear();    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());  }  finally  {  session.getTransaction().commit();  HibernateUtil.shutdown();  }    Output:    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource  Human Resource    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource |

Clearly, evict() method removed the department object from cache so that it was fetched again from database.

/////////////////////////////////////////////////////////////////////////////////////////////////

HIbernate second level cache

# How Hibernate Second Level Cache Works?

July 2, 2013 by Lokesh Gupta

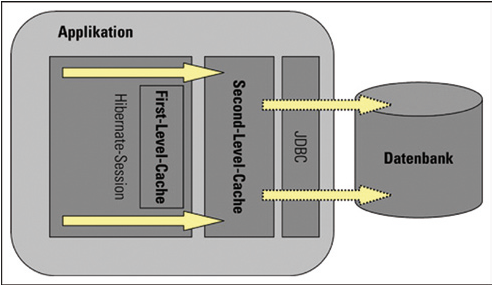
Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.

* Fist level cache: This is enabled by default and works in session scope. Read more about [hibernate first level cache](https://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/).
* Second level cache: This is apart from first level cache which is available to be used globally in session factory scope.

Above statement means, second level cache is created in session factory scope and is available to be used in all sessions which are created using that particular session factory.

It also means that once session factory is closed, all cache associated with it die and cache manager also closed down.

Further, It also means that if you have two instances of session factory (normally no application does that), you will have two cache managers in your application and while accessing cache stored in physical store, you might get unpredictable results like cache-miss.

**Hibernate first and second level cache**

In this tutorial, I am giving concepts around hibernate second level cache and give example using code snippets.

## How second level cache works

Lets write all the facts point by point:

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

|  |
| --- |
| /\*\*  \* Evicts all second level cache hibernate entites. This is generally only  \* needed when an external application modifies the databaase.  \*/  public void evict2ndLevelCache() {  try {  Map<String, ClassMetadata> classesMetadata = sessionFactory.getAllClassMetadata();  for (String entityName : classesMetadata.keySet()) {  logger.info("Evicting Entity from 2nd level cache: " + entityName);  sessionFactory.evictEntity(entityName);  }  } catch (Exception e) {  logger.logp(Level.SEVERE, "SessionController", "evict2ndLevelCache", "Error evicting 2nd level hibernate cache entities: ", e);  }  } |

To understand more using examples, I wrote an application for testing in which I configured EhCache as 2nd level cache. Lets see various scenarios:

a) Entity is fetched very first time

|  |
| --- |
| DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount()); //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount()); //Prints 0    Output: 1 0 |

*Explanation*: Entity is not present in either 1st or 2nd level cache so, it is fetched from database.

b) Entity is fetched second time

|  |
| --- |
| //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount()); //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount()); //Prints 0    Output: 1 0 |

*Explanation*: Entity is present in first level cache so, it is fetched from there. No need to go to second level cache.

c) Entity is evicted from first level cache and fetched again

|  |
| --- |
| //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //Evict from first level cache  session.evict(department);    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount()); //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount()); //Prints 1    Output: 1 1 |

*Explanation*: First time entity is fetched from database. Which cause it store in 1st and 2nd level cache. Second load call fetched from first level cache. Then we evicted entity from 1st level cache. So third load() call goes to second level cache and getSecondLevelCacheHitCount() returns 1.

d) Access second level cache from another session

|  |
| --- |
| //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //Evict from first level cache  session.evict(department);    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount()); //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount()); //Prints 2    Output: 1 2 |

*Explanation*: When another session created from same session factory try to get entity, it is successfully looked up in second level cache and no database call is made.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Second level EH CACHE HIBERNATE

Hibernate EhCache Configuration Tutorial  
July 4, 2013 by Lokesh Gupta  
  
Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.  
  
Fist level cache: This is enabled by default and works in session scope. Read more about hibernate first level cache.  
Second level cache: This is apart from first level cache which is available to be used globally in session factory scope.  
In this tutorial, I am giving an example using ehcache configuration as second level cache in hibernate.  
  
:  
  
Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).  
If cached copy of entity is present in first level cache, it is returned as result of load method.  
If there is no cached entity in first level cache, then second level cache is looked up for cached entity.  
If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.  
If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.  
Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.  
If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.  
About EhCache  
Terracotta Ehcache is a popular open source Java cache that can be used as a Hibernate second level cache. It can be used as a standalone second level cache, or can be configured for clustering to provide a replicated coherent second level cache.  
  
Hibernate ships with the ehcache library. If you want any particular version of ehcache, visit the Terracotta Ehcache download site:  
  
http://www.terracotta.org/products/enterprise-ehcache  
  
The maven dependency is for Ehcache 2.0 and any upgrades is:  
  
<dependency>  
 <groupId>net.sf.ehcache</groupId>  
 <artifactId>ehcache</artifactId>  
 <version>[2.0.0]</version>  
 <type>pom</type>  
</dependency>  
Configuring EhCache  
To configure ehcache, you need to do two steps:  
  
configure Hibernate for second level caching  
specify the second level cache provider  
Hibernate 4.x and above  
  
<property key="hibernate.cache.use\_second\_level\_cache">true</property>  
<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>  
Hibernate 3.3 and above  
  
<property key="hibernate.cache.use\_second\_level\_cache">true</property>  
<property name="hibernate.cache.region.factory\_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</property>  
Hibernate 3.2 and below  
  
<property key="hibernate.cache.use\_second\_level\_cache">true</property>  
<property name="hibernate.cache.region.provider\_class">net.sf.ehcache.hibernate.EhCacheProvider</property>  
Configuring entity objects  
This may done in two ways.  
  
1) If you are using hbm.xml files then use below configuration:  
  
<class name="com.application.entity.DepartmentEntity" table="...">  
 <cache usage="read-write"/>  
</class>  
2) Otherwise, if you are using annotations, use these annotations:  
  
@Entity  
@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY,  
region="department")  
public class DepartmentEntity implements Serializable  
{  
 //code  
}  
For both options, caching strategy can be of following types:  
  
none : No caching will happen.  
read-only : If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.  
read-write : If the application needs to update data, a read-write cache might be appropriate.  
nonstrict-read-write : If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.  
transactional : The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager\_lookup\_class.  
Query caching  
You can also enable query caching. To do so configure it in your hbm.xml:  
  
<property key="hibernate.cache.use\_query\_cache">true</property>  
and where queries are defined in your code, add the method call setCacheable(true) to the queries that should be cached:  
  
sessionFactory.getCurrentSession().createQuery("...").setCacheable(true).list();  
By default, Ehcache will create separate cache regions for each entity that you configure for caching. You can change the defaults for these regions by adding the configuration to your ehcache.xml. To provide this configuration file, use this property in hibernate configuration:  
  
<property name="net.sf.ehcache.configurationResourceName">/ehcache.xml</property>  
And use below configuration to override the default configuration:  
  
<cache  
 name="com.somecompany.someproject.domain.Country"  
 maxElementsInMemory="10000"  
 eternal="false"  
 timeToIdleSeconds="300"  
 timeToLiveSeconds="600"  
 overflowToDisk="true"  
/>  
Please note that in ehcache.xml, if eternal=”true” then we should not write timeToIdealSeconds, timeToLiveSeconds, hibernate will take care about those values  
So if you want to give values manually better use eternal=”false” always, so that we can assign values into timeToIdealSeconds, timeToLiveSeconds manually.  
  
timeToIdealSeconds=”seconds” means, if the object in the global cache is ideal, means not using by any other class or object then it will be waited for some time we specified and deleted from the global cache if time is exceeds more than timeToIdealSeconds value.  
  
timeToLiveSeconds=”seconds” means, the other Session or class using this object or not, i mean may be it is using by other sessions or may not, what ever the situation might be, once it competed the time specified timeToLiveSeconds, then it will be removed from the global cache by hibernate.  
  
Example application  
In our example application, I have one DepartmentEntity for which I want to enable second level cache using ehcache. Lets record the changes step by step:  
  
1) hibernate.cfg.xml  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-configuration PUBLIC  
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
 <session-factory>  
 <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>  
 <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatedemo</property>  
 <property name="hibernate.connection.password">password</property>  
 <property name="hibernate.connection.username">root</property>  
 <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  
 <property name="show\_sql">true</property>  
 <property name="hbm2ddl.auto">create</property>  
 <property name="hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider</property>  
 <mapping class="hibernate.test.dto.DepartmentEntity"></mapping>  
 </session-factory>  
</hibernate-configuration>  
2) DepartmentEntity.java  
  
package hibernate.test.dto;  
   
import java.io.Serializable;  
   
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import javax.persistence.UniqueConstraint;  
   
import org.hibernate.annotations.Cache;  
import org.hibernate.annotations.CacheConcurrencyStrategy;  
   
@Entity (name = "dept")  
@Table(name = "DEPARTMENT", uniqueConstraints = {  
 @UniqueConstraint(columnNames = "ID"),  
 @UniqueConstraint(columnNames = "NAME") })  
   
@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY, region="department")  
   
public class DepartmentEntity implements Serializable {  
   
 private static final long serialVersionUID = 1L;  
   
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "ID", unique = true, nullable = false)  
 private Integer id;  
   
 @Column(name = "NAME", unique = true, nullable = false, length = 100)  
 private String name;  
   
 public Integer getId() {  
 return id;  
 }  
   
 public void setId(Integer id) {  
 this.id = id;  
 }  
   
 public String getName() {  
 return name;  
 }  
   
 public void setName(String name) {  
 this.name = name;  
 }  
}  
3) HibernateUtil.java  
  
package hibernate.test;  
   
import java.io.File;  
   
import org.hibernate.SessionFactory;  
import org.hibernate.cfg.AnnotationConfiguration;  
   
public class HibernateUtil  
{  
 private static final SessionFactory sessionFactory = buildSessionFactory();  
   
 private static SessionFactory buildSessionFactory()  
 {  
 try  
 {  
 // Create the SessionFactory from hibernate.cfg.xml  
 return new AnnotationConfiguration().configure(new File("hibernate.cgf.xml")).buildSessionFactory();  
 }  
 catch (Throwable ex) {  
 // Make sure you log the exception, as it might be swallowed  
 System.err.println("Initial SessionFactory creation failed." + ex);  
 throw new ExceptionInInitializerError(ex);  
 }  
 }  
   
 public static SessionFactory getSessionFactory() {  
 return sessionFactory;  
 }  
   
 public static void shutdown() {  
 // Close caches and connection pools  
 getSessionFactory().close();  
 }  
}  
4) TestHibernateEhcache.java  
  
public class TestHibernateEhcache  
{   
 public static void main(String[] args)  
 {  
 storeData();  
   
 try  
 {  
 //Open the hibernate session  
 Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
   
 //fetch the department entity from database first time  
 DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  
 System.out.println(department.getName());  
   
 //fetch the department entity again; Fetched from first level cache  
 department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  
 System.out.println(department.getName());  
   
 //Let's close the session  
 session.getTransaction().commit();  
 session.close();  
   
 //Try to get department in new session  
 Session anotherSession = HibernateUtil.getSessionFactory().openSession();  
 anotherSession.beginTransaction();  
   
 //Here entity is already in second level cache so no database query will be hit  
 department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class, new Integer(1));  
 System.out.println(department.getName());  
   
 anotherSession.getTransaction().commit();  
 anotherSession.close();  
 }  
 finally  
 {  
 System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount()); //Prints 1  
 System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount()); //Prints 1  
   
 HibernateUtil.shutdown();  
 }  
 }  
   
 private static void storeData()  
 {  
 Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
   
 DepartmentEntity department = new DepartmentEntity();  
 department.setName("Human Resource");  
   
 session.save(department);  
 session.getTransaction().commit();  
 }  
}  
   
Output:  
   
Hibernate: insert into DEPARTMENT (NAME) values (?)  
Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  
Human Resource  
Human Resource  
Human Resource  
1  
1  
In above output, first time the department is fetched from database. but next two times it is fetched from cache. Last fetch is from second level cache.

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

LAZY LOADING

# Hibernate – Lazy Loading Tutorial

September 26, 2014 by Lokesh Gupta

So far in previous tutorials, we have learned the concepts around [entities persistence life cycle states](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/), Some CRUD operations such as [Save a hibernate entity](https://howtodoinjava.com/hibernate/save-and-saveorupdate-for-saving-hibernate-entities/), [merge or refresh an entity](https://howtodoinjava.com/hibernate/merging-and-refreshing-hibernate-entities/) and then we learned about [cascading effect](https://howtodoinjava.com/hibernate/hibernate-jpa-cascade-types/) as well. Moving forward in this this tutorial, I will be discussing a must-known feature in hibernate, specially if you working in a very large application, known as lazy loading.

## When Lazy Loading is Needed : Sample Problem

Consider one of common Internet web application: the online store. The store maintains a catalog of products. At the crudest level, this can be modeled as a catalog entity managing a series of product entities. In a large store, there may be tens of thousands of products grouped into various overlapping categories.

When a customer visits the store, the catalog must be loaded from the database. We probably don’t want the implementation to load every single one of the entities representing the tens of thousands of products to be loaded into memory. For a sufficiently large retailer, this might not even be possible, given the amount of physical memory available on the machine. Even if this were possible, it would probably cripple the performance of the site. Instead, we want only the catalog to load, possibly with the categories as well. Only when the user drills down into the categories should a subset of the products in that category be loaded from the database.

To manage this problem, Hibernate provides a facility called lazy loading. When enabled, an entity’s associated entities will be loaded only when they are directly requested.

## How Lazy Loading Solve the Problem

Now when we have understood the problem, let’s understand how lazy loading actually helps in real life. If we consider to solve the problem discussed above then we would be accessing a category (or catalog) in below manner:

|  |
| --- |
| //Following code loads only a single category from the database:  Category category = (Category)session.get(Category.class,new Integer(42)); |

However, if all products of this category are accessed, and lazy loading is in effect, the products are pulled from the database as needed. For instance, in the following snippet, the associated product objects will be loaded since it is explicitly referenced in second line.

|  |
| --- |
| //Following code loads only a single category from the database  Category category = (Category)session.get(Category.class,new Integer(42));  //This code will fetch all products for category 42 from database 'NOW'  Set<Product> products = category.getProducts(); |

This solve our problem of loading the products only when they are needed.

## How to Enable Lazy Loading in Hibernate

Before moving further, it is important to recap the default behavior of lazy loading in case of using hibernate mappings vs annotations.

The default behavior is to load ‘property values eagerly’ and to load ‘collections lazily’. Contrary to what you might remember if you have used plain Hibernate 2 (mapping files) before, where all references (including collections) are loaded eagerly by default.

Also note that @OneToMany and @ManyToMany associations are defaulted to LAZY loading; and @OneToOne and @ManyToOne are defaulted to EAGER loading. This is important to remember to avoid any pitfall in future.

To enable lazy loading explicitly you must use "fetch = FetchType.LAZY" on a association which you want to lazy load when you are using hibernate annotations.

An example usage will look like this:

|  |
| --- |
| @OneToMany( mappedBy = "category", fetch = FetchType.LAZY )  private Set<ProductEntity> products; |

Another attribute parallel to "FetchType.LAZY" is "FetchType.EAGER" which is just opposite to LAZY i.e. it will load association entity as well when owner entity is fetched first time.

## How Lazy Loading Works in Hibernate

The simplest way that Hibernate can apply lazy load behavior upon your entities and associations is by providing a proxy implementation of them. Hibernate intercepts calls to the entity by substituting a proxy for it derived from the entity’s class. Where the requested information is missing, it will be loaded from the database before control is ceded to the parent entity’s implementation.

Please note that when the association is represented as a collection class, then a wrapper (essentially a proxy for the collection, rather than for the entities that it contains) is created and substituted for the original collection. When you access this collection proxy then what you get inside returned proxy collection are not proxy entities; rather they are actual entities. You need not to put much pressure on understanding this concept because on runtime it hardly matters.

## Effect of Lazy Loading on Detached Entities

As we know that hibernate can only access the database via a session, So If an entity is detached from the session and when we try to access an association (via a proxy or collection wrapper) that has not yet been loaded, Hibernate throws a LazyInitializationException.

The cure is to ensure either that the entity is made persistent again by attaching it to a session or that all of the fields that will be required are accessed (so they are loaded into entity) before the entity is detached from the session.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

[HowToDoInJava](https://howtodoinjava.com/)

* [All Tutorials](https://howtodoinjava.com/)
* [TypeScript](https://howtodoinjava.com/typescript/typescript-tutorial/)
* [Spring Boot](https://howtodoinjava.com/spring-boot-tutorials/)
* [Java 8](https://howtodoinjava.com/java-8-tutorial/)
* [Interview Questions](https://howtodoinjava.com/java-interview-questions/)

# Hibernate JPA Cascade Types

September 25, 2014 by Lokesh Gupta

We learned about [mapping associated entities](https://howtodoinjava.com/hibernate/how-to-define-association-mappings-between-hibernate-entities/) in hibernate already in previous tutorials such as [one-to-one mapping](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping-using-annotations/) and [one-to-many](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping-using-annotations/) mappings. There we wanted to save the mapped entity whenever relationship owner entity got saved. To enable this we had use “CascadeType” attribute. In this tutorial, we will learn about various type of available options for cascading via CascadeType.

Before moving forward, let’s look at how this cascade type attribute is defined in your code. Let’s have an example for more clear understanding. Take a scenario where an Employee can have multiple Accounts; but one account must be associated with only one employee. Let’s create entities with minimum information for sake of clarity.

EmployeeEntity.java

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {  private static final long serialVersionUID = -1798070786993154676L;  @Id  @Column(name = "ID", unique = true, nullable = false)  private Integer employeeId;  @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)  private String firstName;  @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)  private String lastName;    @OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts;    //Getters and Setters Ommited  } |

AccountEntity.java

|  |
| --- |
| @Entity  @Table(name = "Account")  public class AccountEntity implements Serializable  {  private static final long serialVersionUID = 1L;  @Id  @Column(name = "ID", unique = true, nullable = false)  @GeneratedValue(strategy = GenerationType.SEQUENCE)  private Integer accountId;  @Column(name = "ACC\_NO", unique = false, nullable = false, length = 100)  private String accountNumber;    @OneToOne (mappedBy="accounts", fetch = FetchType.LAZY)  private EmployeeEntity employee;    } |

Look at the bold line in above source code for EmployeeEntity.java. It defines “cascade=CascadeType.ALL” and it essentially means that any change happened on EmployeeEntitymust cascade to AccountEntity as well. If you save an employee, then all associated accounts will also be saved into database. If you delete an Employee then all accounts associated with that Employee also be deleted. Simple enough.

But what if we only want to cascade only save operations but not delete operation. Then we need to clearly specify it using below code.

|  |
| --- |
| @OneToMany(cascade=CascadeType.PERSIST, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts; |

Now only when save() or persist() methods are called using employee instance then only accounts will be persisted. If any other method is called on session, it’s effect will not affect/cascade to accounts.

## JPA Cascade Types

The cascade types supported by the Java Persistence Architecture are as below:

1. CascadeType.PERSIST : means that save() or persist() operations cascade to related entities.
2. CascadeType.MERGE : means that related entities are merged when the owning entity is merged.
3. CascadeType.REFRESH : does the same thing for the refresh() operation.
4. CascadeType.REMOVE : removes all related entities association with this setting when the owning entity is deleted.
5. CascadeType.DETACH : detaches all related entities if a “manual detach” occurs.
6. CascadeType.ALL : is shorthand for all of the above cascade operations.

The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:

|  |
| --- |
| @OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts; |

Above cascading will cause accounts collection to be only merged and refreshed.

## Hibernate Specific Cascade Types

There’s one more cascading operation that’s not part of the normal set above discussed, called “orphan removal“, which removes an owned object from the database when it’s removed from its owning relationship.

Let’s understand with an example. In our Employee and Account entity example, I have updated them as below and have mentioned “orphanRemoval = true” on accounts. It essentially means that whenever I will remove an ‘account from accounts set’ (which means I am removing the relationship between that account and Employee); the account entity which is not associated with any other Employee on database (i.e. orphan) should also be deleted.

EmployeeEntity.java

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {  private static final long serialVersionUID = -1798070786993154676L;  @Id  @Column(name = "ID", unique = true, nullable = false)  private Integer employeeId;  @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)  private String firstName;  @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)  private String lastName;    @OneToMany(orphanRemoval = true, mappedBy = "employee")  private Set<AccountEntity> accounts;    } |

AccountEntity.java

|  |
| --- |
| @Entity (name = "Account")  @Table(name = "Account")  public class AccountEntity implements Serializable  {  private static final long serialVersionUID = 1L;  @Id  @Column(name = "ID", unique = true, nullable = false)  @GeneratedValue(strategy = GenerationType.SEQUENCE)  private Integer accountId;  @Column(name = "ACC\_NO", unique = false, nullable = false, length = 100)  private String accountNumber;    @ManyToOne  private EmployeeEntity employee;  } |

TestOrphanRemovalCascade.java

|  |
| --- |
| public class TestOrphanRemovalCascade  {  public static void main(String[] args)  {  setupTestData();    Session sessionOne = HibernateUtil.getSessionFactory().openSession();  org.hibernate.Transaction tx = sessionOne.beginTransaction();    //Load the employee in another session  EmployeeEntity employee = (EmployeeEntity) sessionOne.load(EmployeeEntity.class, 1);  //Verify there are 3 accounts  System.out.println("Step 1 : " + employee.getAccounts().size());    //Remove an account from first position of collection  employee.getAccounts().remove(employee.getAccounts().iterator().next());    //Verify there are 2 accounts in collection  System.out.println("Step 2 : " + employee.getAccounts().size());    tx.commit();  sessionOne.close();    //In another session check the actual data in database  Session sessionTwo = HibernateUtil.getSessionFactory().openSession();  sessionTwo.beginTransaction();    EmployeeEntity employee1 = (EmployeeEntity) sessionTwo.load(EmployeeEntity.class, 1);  //Verify there are 2 accounts now associated with Employee  System.out.println("Step 3 : " + employee1.getAccounts().size());    //Verify there are 2 accounts in Account table  Query query = sessionTwo.createQuery("from Account a");  @SuppressWarnings("unchecked")  List<AccountEntity> accounts = query.list();  System.out.println("Step 4 : " + accounts.size());    sessionTwo.close();    HibernateUtil.shutdown();  }    private static void setupTestData(){  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    //Create Employee  EmployeeEntity emp = new EmployeeEntity();  emp.setEmployeeId(1);  emp.setFirstName("Lokesh");  emp.setLastName("Gupta");  session.save(emp);    //Create Account 1  AccountEntity acc1 = new AccountEntity();  acc1.setAccountId(1);  acc1.setAccountNumber("11111111");  acc1.setEmployee(emp);  session.save(acc1);    //Create Account 2  AccountEntity acc2 = new AccountEntity();  acc2.setAccountId(2);  acc2.setAccountNumber("2222222");  acc2.setEmployee(emp);  session.save(acc2);    //Create Account 3  AccountEntity acc3 = new AccountEntity();  acc3.setAccountId(3);  acc3.setAccountNumber("33333333");  acc3.setEmployee(emp);  session.save(acc3);    session.getTransaction().commit();  session.close();  }  }    Output:    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: insert into Account (ACC\_NO, employee\_ID, ID) values (?, ?, ?)  Hibernate: insert into Account (ACC\_NO, employee\_ID, ID) values (?, ?, ?)  Hibernate: insert into Account (ACC\_NO, employee\_ID, ID) values (?, ?, ?)  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as  LAST\_NAM3\_1\_0\_ from Employee employeeen0\_ where employeeen0\_.ID=?  Hibernate: select accounts0\_.employee\_ID as employee3\_1\_0\_, accounts0\_.ID as ID1\_0\_0\_, accounts0\_.ID as ID1\_0\_1\_,  accounts0\_.ACC\_NO as ACC\_NO2\_0\_1\_, accounts0\_.employee\_ID as employee3\_0\_1\_ from Account accounts0\_ where accounts0\_.employee\_ID=?  Step 1 : 3  Step 2 : 2  Hibernate: delete from Account where ID=?  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as  LAST\_NAM3\_1\_0\_ from Employee employeeen0\_ where employeeen0\_.ID=?  Hibernate: select accounts0\_.employee\_ID as employee3\_1\_0\_, accounts0\_.ID as ID1\_0\_0\_, accounts0\_.ID as ID1\_0\_1\_,  accounts0\_.ACC\_NO as ACC\_NO2\_0\_1\_, accounts0\_.employee\_ID as employee3\_0\_1\_ from Account accounts0\_ where accounts0\_.employee\_ID=?  Step 3 : 2  Hibernate: select accountent0\_.ID as ID1\_0\_, accountent0\_.ACC\_NO as ACC\_NO2\_0\_, accountent0\_.employee\_ID as employee3\_0\_  from Account accountent0\_  Step 4 : 2 |

It’s a very good way of removing the matching/mismatching items from a collection (i.e. many-to-one or one-to-many relationships). You just remove the item from collection and hibernate take care of rest of the things for you. It will check whether entity is referenced from any place or not; If it is not then it will delete the entity from database itself.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////

JDBC\

# Java JDBC PreparedStatement Example

November 24, 2013 by Lokesh Gupta

In database management systems, a [prepared statement](https://en.wikipedia.org/wiki/Prepared_statement) or parameterized statement is a feature used to execute the same or similar database statements repeatedly with high efficiency. Typically used with SQL statements such as queries or updates, the prepared statement takes the form of a template into which certain constant values are substituted during each execution.

A typical template would look like this: “INSERT INTO EMPLOYEE (ID, NAME) VALUES (?, ?);”

Here values are set in runtime at placeholders represented by “?”.

## How prepared statement works?

Most relational databases handles a JDBC / SQL query in four steps:

1. Parse the incoming SQL query
2. Compile the SQL query
3. Plan/optimize the data acquisition path
4. Execute the optimized query / acquire and return data

A Statement will always proceed through the four steps above for each SQL query sent to the database. A Prepared Statement pre-executes steps (1) – (3) in the execution process above. Thus, when creating a Prepared Statement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time.

## Advantages of using prepared statement over simple JDBC statement

* Pre-compilation and DB-side caching of the SQL statement leads to overall faster execution and the ability to reuse the same SQL statement in batches.
* Automatic prevention of SQL injection attacks by builtin escaping of quotes and other special characters. Note that this requires that you use any of the PreparedStatement setXxx() methods to set the values and not use inline the values in the SQL string by string-concatenating.
* Apart from above two main usage, prepared statements makes it easy to work with complex objects like BLOBs and CLOBs.

If you have missed, in previous posts, we have learned about types of [JDBC drivers](https://howtodoinjava.com/core-java/jdbc/jdbc-basics-types-of-jdbc-drivers/) and some basic operations like [making database connection using JDBC](https://howtodoinjava.com/core-java/jdbc/jdbc-mysql-database-connection-example/) and then how to [execute SELECT Query](https://howtodoinjava.com/misc/jdbc-select-query-example/), and then [INSET Query example](https://howtodoinjava.com/core-java/jdbc/jdbc-sql-insert-query-example/)

Execution of prepared statements requires following steps:

1) Make a database connection

2) Set values and execute prepared statement

Pre-requisites include setting up a database schema and creating a table at least.

|  |
| --- |
| CREATE SCHEMA 'JDBCDemo' ;    CREATE TABLE 'JDBCDemo'.'EMPLOYEE'  (  'ID' INT NOT NULL DEFAULT 0 ,  'FIRST\_NAME' VARCHAR(100) NOT NULL ,  'LAST\_NAME' VARCHAR(100) NULL ,  'STAT\_CD' TINYINT NOT NULL DEFAULT 0  ); |

Let’s write above steps in code:

## 1) Make JDBC database connection

Though we have already learned about it in making [JDBC connection](https://howtodoinjava.com/core-java/jdbc/jdbc-mysql-database-connection-example/), lets recap with this simple code snippet.

|  |
| --- |
| Class.forName("com.mysql.jdbc.Driver");  connection = DriverManager  .getConnection("jdbc:mysql://localhost:3306/JDBCDemo", "root", "password"); |

## 2) Set values and execute PreparedStatement

This is the main step and core part in the post. It requires creating a Statement object and then using it’s executeQuery() method.

|  |
| --- |
| PreparedStatement pstmt = connection.prepareStatement(sql);  pstmt.setInt(1, 87);  pstmt.setString(2, "Lokesh");  pstmt.setString(3, "Gupta");  pstmt.setInt(4, 5);    int affectedRows = pstmt.executeUpdate(); |

Let’s see the whole code in working.

## Complete JDBC PreparedStatement Example

|  |
| --- |
| package com.howtodoinjava.jdbc.demo;    import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;    public class PreparedStatementDemo  {  public static void main(String[] args)  {  Connection connection = null;  PreparedStatement pstmt = null;  String sql = "INSERT INTO EMPLOYEE (ID,FIRST\_NAME,LAST\_NAME,STAT\_CD) VALUES (?,?,?,?)";  try  {  Class.forName("com.mysql.jdbc.Driver");  connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/JDBCDemo","root", "password");    pstmt = connection.prepareStatement(sql);  pstmt.setInt(1, 87);  pstmt.setString(2, "Lokesh");  pstmt.setString(3, "Gupta");  pstmt.setInt(4, 5);  int affectedRows = pstmt.executeUpdate();  System.out.println(affectedRows + " row(s) affected !!");  }  catch (Exception e) {  e.printStackTrace();  }finally {  try {  pstmt.close();  connection.close();  } catch (Exception e) {  e.printStackTrace();  }  }  }  }    Output:    1 row(s) affected !! |

SQL DELETE query are executed to remove/delete data stored in relational databases. It requires following steps:

1) Make a database connection

2) Execute the SQL DELETE Query

Pr-requisites include setting up a database schema and creating a table first.

|  |
| --- |
| CREATE SCHEMA 'JDBCDemo' ;    CREATE TABLE 'JDBCDemo'.'EMPLOYEE'  (  'ID' INT NOT NULL DEFAULT 0 ,  'FIRST\_NAME' VARCHAR(100) NOT NULL ,  'LAST\_NAME' VARCHAR(100) NULL ,  'STAT\_CD' TINYINT NOT NULL DEFAULT 0  ); |

Let’s write above steps in code:

## 1) Make a database connection

Though we have already learned about it in making JDBC connection, lets recap with this simple code snippet.

|  |
| --- |
| Class.forName("com.mysql.jdbc.Driver");  connection = DriverManager  .getConnection("jdbc:mysql://localhost:3306/JDBCDemo", "root", "password"); |

## 2) Execute the SQL DELETE Query

This is the main step and core part in the post. It requires creating a Statement object and then using it’s execute() method.

|  |
| --- |
| Statement stmt = connection.createStatement();  stmt.execute("DELETE FROM EMPLOYEE WHERE ID >= 1"); |

Above statement will execute delete statement in database we are connected to. This will remove all records which match by where clause.

Let’s see the whole code in working.

|  |
| --- |
| package com.howtodoinjava.jdbc.demo;    import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.Statement;    public class DeleteDataDemo {  public static void main(String[] args) {  Connection connection = null;  Statement stmt = null;  try  {  Class.forName("com.mysql.jdbc.Driver");  connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/JDBCDemo","root", "password");    stmt = connection.createStatement();  stmt.execute("DELETE FROM EMPLOYEE WHERE ID >= 1");  }  catch (Exception e) {  e.printStackTrace();  }finally {  try {  stmt.close();  connection.close();  } catch (Exception e) {  e.printStackTrace();  }  }  }  } |

SQL INSERT query are executed to push/store data stored in relational databases. It requires following steps:

1) Make a database connection

2) Execute the SQL INSERT Query

Pr-requisites include setting up a database schema and creating a table at least.

|  |
| --- |
| CREATE SCHEMA 'JDBCDemo' ;    CREATE TABLE 'JDBCDemo'.'EMPLOYEE'  (  'ID' INT NOT NULL DEFAULT 0 ,  'FIRST\_NAME' VARCHAR(100) NOT NULL ,  'LAST\_NAME' VARCHAR(100) NULL ,  'STAT\_CD' TINYINT NOT NULL DEFAULT 0  ); |

Let’s write above steps in code:

## 1) Make a database connection

Though we have already learned about it in making JDBC connection, lets recap with this simple code snippet.

|  |
| --- |
| Class.forName("com.mysql.jdbc.Driver");  connection = DriverManager  .getConnection("jdbc:mysql://localhost:3306/JDBCDemo", "root", "password"); |

## 2) Execute the SQL INSERT Query

This is the main step and core part in ths post. It requires creating a Statement object and then using it’s execute() method.

|  |
| --- |
| Statement stmt = connection.createStatement();  stmt.execute("INSERT INTO EMPLOYEE (ID,FIRST\_NAME,LAST\_NAME,STAT\_CD) VALUES (1,'Lokesh','Gupta',5)"); |

Above statement will execute an insert statement in database we are connected to.

Let’s see the whole code in working.

|  |
| --- |
| package com.howtodoinjava.jdbc.demo;    import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.Statement;    public class InsertDataDemo {  public static void main(String[] args) {  Connection connection = null;  Statement stmt = null;  try  {  Class.forName("com.mysql.jdbc.Driver");  connection = DriverManager  .getConnection("jdbc:mysql://localhost:3306/JDBCDemo", "root", "password");    stmt = connection.createStatement();  stmt.execute("INSERT INTO EMPLOYEE (ID,FIRST\_NAME,LAST\_NAME,STAT\_CD) "  + "VALUES (1,'Lokesh','Gupta',5)");  }  catch (Exception e) {  e.printStackTrace();  }finally {  try {  stmt.close();  connection.close();  } catch (Exception e) {  e.printStackTrace();  }  }  }  } |

That’s all in this post. Drop me a comment if something needs explanation.

Happy Leaning !!

SELECT  
In previous posts, we have learned about types of JDBC drivers and the how to make database connection using JDBC. Let’s move forward and start interacting with database. First example I am picking up is SQL SELECT queries.  
  
JDBC-Icon  
  
SQL SELECT query are executed to fetch data stored in relational databases. It requires following steps:  
  
1) Make a database connection  
2) Execute the SQL Query  
3) Fetch the data from result set  
  
Pre-requisites include setting up a database schema and creating a table at least.  
  
CREATE SCHEMA 'JDBCDemo' ;  
   
CREATE TABLE 'JDBCDemo'.'EMPLOYEE'  
(  
 'ID' INT NOT NULL DEFAULT 0 ,  
 'FIRST\_NAME' VARCHAR(100) NOT NULL ,  
 'LAST\_NAME' VARCHAR(100) NULL ,  
 'STAT\_CD' TINYINT NOT NULL DEFAULT 0  
);  
Let’s write above steps in code:  
  
1) Make a database connection  
Though we have already learned about it in making JDBC connection, lets recap with this simple code snippet.  
  
Class.forName("com.mysql.jdbc.Driver");  
connection = DriverManager  
 .getConnection("jdbc:mysql://localhost:3306/JDBCDemo", "root", "password");  
2) Execute the SQL Query  
This is the main step and core part in the post. It requires creating a Statement object and then using it’s executeQuery() method.  
  
Statement selectStmt = connection.createStatement();  
ResultSet rs = selectStmt  
 .executeQuery("SELECT ID,FIRST\_NAME,LAST\_NAME,STAT\_CD FROM EMPLOYEE WHERE ID <= 10");  
3) Fetch the data from result set  
You can use various getXXX() methods available in ResultSet. But if you want to make it generic then use getString() method and parse the data as and when needed.  
  
ResultSet rs = selectStmt  
 .executeQuery("SELECT ID,FIRST\_NAME,LAST\_NAME,STAT\_CD FROM EMPLOYEE WHERE ID <= 10");  
while(rs.next())  
{  
 System.out.println(rs.getString(1)); //First Column  
 System.out.println(rs.getString(2)); //Second Column  
 System.out.println(rs.getString(3)); //Third Column  
 System.out.println(rs.getString(4)); //Fourth Column  
}  
Let’s see the whole code in working.  
  
package com.howtodoinjava.jdbc.demo;  
   
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.Statement;  
   
public class SelectDataDemo {  
 public static void main(String[] args) {  
 Connection connection = null;  
 Statement insertStmt = null;  
 Statement selectStmt = null;  
 try  
 {  
 Class.forName("com.mysql.jdbc.Driver");  
 connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/JDBCDemo", "root", "password");  
   
 /\*insertStmt = connection.createStatement();  
 insertStmt.execute("INSERT INTO EMPLOYEE (ID,FIRST\_NAME,LAST\_NAME,STAT\_CD) VALUES (1,'Lokesh','Gupta',5)");  
 insertStmt.execute("INSERT INTO EMPLOYEE (ID,FIRST\_NAME,LAST\_NAME,STAT\_CD) VALUES (2,'howtodoinjava','com',5)");\*/  
   
 selectStmt = connection.createStatement();  
 ResultSet rs = selectStmt.executeQuery("SELECT ID,FIRST\_NAME,LAST\_NAME,STAT\_CD FROM EMPLOYEE WHERE ID <= 10");  
 while(rs.next())  
 {  
 System.out.println(rs.getString(1)); //First Column  
 System.out.println(rs.getString(2)); //Second Column  
 System.out.println(rs.getString(3)); //Third Column  
 System.out.println(rs.getString(4)); //Fourth Column  
 }  
 }  
 catch (Exception e) {  
 e.printStackTrace();  
 }finally {  
 try {  
 selectStmt.close();  
 insertStmt.close();  
 connection.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
   
Output:  
   
1  
Lokesh  
Gupta  
5  
2  
howtodoinjava  
com  
5

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/What is factory Pattern

In Java, you have 3 different kinds of Design Patterns.  
  
Creational - How objects are created  
Behavioral - How objects interact (behave) with each other  
Structural - How objects are structured or laid out (relation with each other)  
Factory Design Pattern is one of the creational design patterns. It is to help you with the creation of an object at runtime without the client needing to know the internal implementation details (not even the name of the class). For this, the factory needs to have one additional but mandatory input - which indicates what type of object you want to create.  
  
For example, you have 2 different Databases in your application. Say MySQL and Oracle and you want to work with both but you would decide at runtime which database you want to work with. In this scenario, you go with a Factory Pattern by providing the name of the database you want to work with, as an input parameter.  
  
See the below example. It is NOT a complete example but for your better understanding in a short and sweet manner.  
  
public DAOInstance getDAOInstance(String input)  
{  
 if(input.equalsIgnoreCase("Oracle"))  
 {   
 return new OracleDAO();  
 }  
 else if (input.equalsIgnoreCase("MySQL"))  
 {  
 return new MySQLDAO();  
 }   
 else  
 return null;  
}

/////////////////////////////////////////)//////////////////

**COMPARE AND SWAP WORKING**

**How CAS (Compare And Swap) in Java works?**Before we dig into CAS (Compare And Swap) strategy and how is it used by atomic constructs like AtomicInteger, first consider this code:  
public class MyApp  
{  
 private volatile int count = 0;  
 public void upateVisitors()   
 {  
 ++count; //increment the visitors count  
 }  
}  
  
This sample code is tracking the count of visitors to the application. Is there anything wrong with this code? What will happen if multiple threads try to update count? Actually the problem is simply marking count as volatile does not guarantee atomicity and ++count is not an atomic operations. To read more check this.  
  
Can we solve this problem if we mark the method itself synchronized as shown below:  
public class MyApp  
{  
 private int count = 0;  
 public synchronized void upateVisitors()   
 {  
 ++count; //increment the visitors count  
 }  
}  
  
Will this work? If yes then what changes have we made actually?  
Does this code guarantee atomicity? Yes.  
Does this code guarantee visibility? Yes.  
  
Then what is the problem?  
It makes use of locking and that introduces lot of delay and overhead. Check this article. This is very expensive way of making things work.  
  
To overcome these problems atomic constructs were introduced. If we make use of an AtomicInteger to track the count it will work.   
public class MyApp  
{  
 private AtomicInteger count = new AtomicInteger(0);  
 public void upateVisitors()   
 {  
 count.incrementAndGet(); //increment the visitors count  
 }  
}  
  
The classes that support atomic operations e.g. AtomicInteger, AtomicLong etc. makes use of CAS. CAS does not make use of locking rather it is very optimistic in nature. It follows these steps:   
Compare the value of the primitive to the value we have got in hand.  
If the values do not match it means some thread in between has changed the value. Else it will go ahead and swap the value with new value.  
  
Check the following code in AtomicLong class:  
public final long incrementAndGet() {  
 for (;;) {  
 long current = get();  
 long next = current + 1;  
 if (compareAndSet(current, next))  
 return next;  
 }  
}  
  
In JDK 8 the above code has been changed to a single intrinsic:  
public final long incrementAndGet() {  
 return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;  
}  
  
What advantage this single intrinsic have?  
Actually this single line is JVM intrinsic which is translated by JIT into an optimized instruction sequence. In case of x86 architecture it is just a single CPU instruction LOCK XADD which might yield better performance than classic load CAS loop.  
  
Now think about the possibility when we have high contention and a number of threads want to update the same atomic variable. In that case there is a possibility that locking will outperform the atomic variables but in realistic contention levels atomic variables outperform lock. There is one more construct introduced in Java

//////////////////////////////////////////////////////////

ATOMIC VOLATILE SYNCHRONIZED

You are specifically asking about how they internally work, so here you are:  
  
No synchronization  
private int counter;  
  
public int getNextUniqueIndex() {  
 return counter++;   
}  
It basically reads value from memory, increments it and puts back to memory. This works in single thread but nowadays, in the era of multi-core, multi-CPU, multi-level caches it won't work correctly. First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems. The value might only be stored in "local" CPU memory (some cache) and not be visible for other CPUs/cores (and thus - threads). This is why many refer to local copy of a variable in a thread. It is very unsafe. Consider this popular but broken thread-stopping code:  
  
private boolean stopped;  
  
public void run() {  
 while(!stopped) {  
 //do some work  
 }  
}  
  
public void pleaseStop() {  
 stopped = true;  
}  
Add volatile to stopped variable and it works fine - if any other thread modifies stopped variable via pleaseStop() method, you are guaranteed to see that change immediately in working thread's while(!stopped) loop. BTW this is not a good way to interrupt a thread either, see: How to stop a thread that is running forever without any use and Stopping a specific java thread.  
  
AtomicInteger  
private AtomicInteger counter = new AtomicInteger();  
  
public int getNextUniqueIndex() {  
 return counter.getAndIncrement();  
}  
The AtomicInteger class uses CAS (compare-and-swap) low-level CPU operations (no synchronization needed!) They allow you to modify a particular variable only if the present value is equal to something else (and is returned successfully). So when you execute getAndIncrement() it actually runs in a loop (simplified real implementation):  
  
int current;  
do {  
 current = get();  
} while(!compareAndSet(current, current + 1));  
So basically: read; try to store incremented value; if not successful (the value is no longer equal to current), read and try again. The compareAndSet() is implemented in native code (assembly).  
  
volatile without synchronization  
private volatile int counter;  
  
public int getNextUniqueIndex() {  
 return counter++;   
}  
This code is not correct. It fixes the visibility issue (volatile makes sure other threads can see change made to counter) but still has a race condition. This has been explained multiple times: pre/post-incrementation is not atomic.  
  
The only side effect of volatile is "flushing" caches so that all other parties see the freshest version of the data. This is too strict in most situations; that is why volatile is not default.  
  
volatile without synchronization (2)  
volatile int i = 0;  
void incIBy5() {  
 i += 5;  
}  
The same problem as above, but even worse because i is not private. The race condition is still present. Why is it a problem? If, say, two threads run this code simultaneously, the output might be + 5 or + 10. However, you are guaranteed to see the change.  
  
Multiple independent synchronized  
void incIBy5() {  
 int temp;  
 synchronized(i) { temp = i }  
 synchronized(i) { i = temp + 5 }  
}  
Surprise, this code is incorrect as well. In fact, it is completely wrong. First of all you are synchronizing on i, which is about to be changed (moreover, i is a primitive, so I guess you are synchronizing on a temporary Integer created via autoboxing...) Completely flawed. You could also write:  
  
synchronized(new Object()) {  
 //thread-safe, SRSLy?  
}  
No two threads can enter the same synchronized block with the same lock. In this case (and similarly in your code) the lock object changes upon every execution, so synchronized effectively has no effect.  
  
Even if you have used a final variable (or this) for synchronization, the code is still incorrect. Two threads can first read i to temp synchronously (having the same value locally in temp), then the first assigns a new value to i (say, from 1 to 6) and the other one does the same thing (from 1 to 6).  
  
The synchronization must span from reading to assigning a value. Your first synchronization has no effect (reading an int is atomic) and the second as well. In my opinion, these are the correct forms:  
  
void synchronized incIBy5() {  
 i += 5   
}  
  
void incIBy5() {  
 synchronized(this) {  
 i += 5   
 }  
}  
  
void incIBy5() {  
 synchronized(this) {  
 int temp = i;  
 i = temp + 5;  
 }  
}

///////////////////////////////////////////////////

atomic Opera tion

Doing a = 28 (with a being an int) is an atomic operation. But doing a++ is not an atomic operation because it requires a read of the value of a, an incrementation, and a write to a of the result. As a result, if you used a++ to implement a thread-safe counter, you could have two threads reading the value concurrently (26 for example), then have both increment it and writing it concurrently, resulting in 27 as a result, instead of 28.  
  
AtomicInteger solves this issue by providing atomic operations like the ones you listed. In my example, you would use incrementAndGet() for example, which would guarantee the end value is 28 and not 27.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Writing Thread safe code

# [How to write Thread-Safe Code in Java](https://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html)

**thread-safety** or **thread-safe code in Java** refers to code which can safely be used or shared in concurrent or multi-threading environment and they will behave as expected. any code, class or object which can behave differently from its contract on concurrent environment is not thread-safe. thread-safety is one of the risk introduced by using [threads in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) and I have seen java programmers and developers struggling to *write thread-safe code* or just understanding *what is thread-safe code* and what is not? This will not be very detailed article on thread-safety or low level details of [synchronization in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) instead we will keep it simple and focus on one example of non thread-safe code and try to understand what is thread-safety and **how to make an code thread-safe**.

## How to make Thread-Safe Code in Java

### Example of Non Thread Safe Code in Java

/\*

**\* Non Thread-Safe Class in Java**

\*/

public class **Counter** {

private int count;

/\*

\* This method is not thread-safe because ++ is not an atomic operation

\*/

public int getCount(){

return count++;

}

}

**Above example is not thread-safe** because ++ (increment operator) is not an **atomic operation** and can be broken down into read, update and write operation. if multiple thread call getCount() approximately same time each of these three operation may coincide or overlap with each other for example while thread 1 is updating value , thread 2 reads and still gets old value, which eventually let thread 2 override thread 1 increment and **one count is lost** because multiple thread called it concurrently.

**What is deadlock?**"

Answer is simple, when two or more threads are waiting for each other to release lock and get stuck for infinite time, situation is called deadlock . It will only happen in case of multitasking.

## How do you detect deadlock in Java ?

Though this could have many answers , my version is first I would look the code if I see nested synchronized block or calling one synchronized method from other or trying to get lock on different object then there is good chance of deadlock if developer is not very careful.

Other way is to find it when you actually get locked while running the application , try to take thread dump , in Linux you can do this by command **"kill -3"** , this will print status of all the thread in application log file and you can see which thread is locked on which object

## Write a Java program which will result in deadlock?

Once you answer this , they may ask you to **write code which will result in deadlock ?**

here is one of my version

/\*\*  
 \* Java program to create a deadlock by imposing circular wait.  
 \*   
 \* @author WINDOWS 8  
 \*  
 \*/  
public class DeadLockDemo {  
  
 /\*  
 \* This method request two locks, first String and then Integer  
 \*/  
 public void method1() {  
 synchronized (String.class) {  
 System.out.println("Aquired lock on String.class object");  
  
 synchronized (Integer.class) {  
 System.out.println("Aquired lock on Integer.class object");  
 }  
 }  
 }  
  
 /\*  
 \* This method also requests same two lock but in exactly  
 \* Opposite order i.e. first Integer and then String.   
 \* This creates potential deadlock, if one thread holds String lock  
 \* and other holds Integer lock and they wait for each other, forever.  
 \*/  
 public void method2() {  
 synchronized (Integer.class) {  
 System.out.println("Aquired lock on Integer.class object");  
  
 synchronized (String.class) {  
 System.out.println("Aquired lock on String.class object");  
 }  
 }  
 }  
}

## How to avoid deadlock in Java?

Now interviewer comes to final part, one of the most important in my view; *How do you fix deadlock?* or **How to avoid deadlock in Java?**

If you have looked above code carefully then you may have figured out that real reason for deadlock is not multiple threads but ***the way they are requesting lock*** , if you provide an ordered access then problem will be resolved , here is my fixed version, which avoids deadlock by avoiding circular wait with no preemption.

public class DeadLockFixed {  
  
 /\*\*  
 \* Both method are now requesting lock in same order, first Integer and then String.  
 \* You could have also done reverse e.g. first String and then Integer,  
 \* both will solve the problem, as long as both method are requesting lock  
 \* in consistent order.  
 \*/  
 public void method1() {  
 synchronized (Integer.class) {  
 System.out.println("Aquired lock on Integer.class object");  
  
 synchronized (String.class) {  
 System.out.println("Aquired lock on String.class object");  
 }  
 }  
 }  
  
 public void method2() {  
 synchronized (Integer.class) {  
 System.out.println("Aquired lock on Integer.class object");  
  
 synchronized (String.class) {  
 System.out.println("Aquired lock on String.class object");  
 }  
 }  
 }  
}

Now there would not be any deadlock because both methods are accessing lock on Integer and String class literal in same order. So, if thread A acquires lock on Integer object , thread B will not proceed until thread A releases Integer lock, same way thread A will not be blocked even if thread B holds String lock because now thread B will not expect thread A to release Integer lock to proceed further.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////

# Java FutureTask Example Program

APRIL 2, 2018 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [13 COMMENTS](https://www.journaldev.com/1650/java-futuretask-example-program#comments)

Sometime back I wrote a post about [Java Callable Future](https://www.journaldev.com/1090/java-callable-future-example) interfaces that we can use to get the concurrent processing benefits of threads as well as they are capable of returning value to the calling program.

FutureTask is base concrete implementation of Future interface and provides asynchronous processing. It contains the methods to start and cancel a task and also methods that can return the state of the FutureTask as whether it’s completed or cancelled. We need a callable object to create a future task and then we can use [Java Thread Pool Executor](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) to process these asynchronously.

Let’s see the example of FutureTask with a simple program.

Since FutureTask requires a callable object, we will create a simple Callable implementation.

package com.journaldev.threads;  
  
import java.util.concurrent.Callable;  
  
public class MyCallable implements Callable<String> {  
  
 private long waitTime;  
   
 public MyCallable(int timeInMillis){  
 this.waitTime=timeInMillis;  
 }  
 @Override  
 public String call() throws Exception {  
 Thread.sleep(waitTime);  
 //return the thread name executing this callable task  
 return Thread.currentThread().getName();  
 }  
  
}

Here is an example of FutureTask method and it’s showing commonly used methods of FutureTask.

package com.journaldev.threads;  
  
import java.util.concurrent.ExecutionException;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.FutureTask;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.TimeoutException;  
  
public class FutureTaskExample {  
  
 public static void main(String[] args) {  
 MyCallable callable1 = new MyCallable(1000);  
 MyCallable callable2 = new MyCallable(2000);  
  
 FutureTask<String> futureTask1 = new FutureTask<String>(callable1);  
 FutureTask<String> futureTask2 = new FutureTask<String>(callable2);  
  
 ExecutorService executor = Executors.newFixedThreadPool(2);  
 executor.execute(futureTask1);  
 executor.execute(futureTask2);  
   
 while (true) {  
 try {  
 if(futureTask1.isDone() && futureTask2.isDone()){  
 System.out.println("Done");  
 //shut down executor service  
 executor.shutdown();  
 return;  
 }  
   
 if(!futureTask1.isDone()){  
 //wait indefinitely for future task to complete  
 System.out.println("FutureTask1 output="+futureTask1.get());  
 }  
   
 System.out.println("Waiting for FutureTask2 to complete");  
 String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);  
 if(s !=null){  
 System.out.println("FutureTask2 output="+s);  
 }  
 } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }catch(TimeoutException e){  
 //do nothing  
 }  
 }  
   
 }  
  
}

When we run above program, you will notice that it doesn’t print anything for sometime because get()method of FutureTask waits for the task to get completed and then returns the output object. There is an overloaded method also to wait for only specified amount of time and we are using it for futureTask2. Also notice the use of isDone() method to make sure program gets terminated once all the tasks are executed.

Output of above program will be:

FutureTask1 output=pool-1-thread-1  
Waiting for FutureTask2 to complete  
Waiting for FutureTask2 to complete  
Waiting for FutureTask2 to complete  
Waiting for FutureTask2 to complete  
Waiting for FutureTask2 to complete  
FutureTask2 output=pool-1-thread-2  
Done

As such there is no benefit of FutureTask but it comes handy when we want to override some of Future interface methods and don’t want to implement every method of Future interface.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

# Java Callable Future Example

APRIL 3, 2018 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [25 COMMENTS](https://www.journaldev.com/1090/java-callable-future-example#comments)

Java Callable and Future are used a lot in multithreaded programming. In last few posts, we learned a lot about [java threads](https://www.journaldev.com/1079/multithreading-in-java) but sometimes we wish that a thread could return some value that we can use. Java 5 introduced **java.util.concurrent.Callable** interface in [concurrency](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers) package that is similar to Runnable interface but it can return any Object and able to throw Exception.

## Java Callable

Java Callable interface use Generic to define the return type of Object. [Executors](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) class provide useful methods to execute Java Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object.

## Java Future

Java Callable tasks return **java.util.concurrent.Future** object. Using **Java Future** object, we can find out the status of the Callable task and get the returned Object. It provides **get()** method that can wait for the Callable to finish and then return the result.

Java Future provides **cancel()** method to cancel the associated Callable task. There is an overloaded version of get() method where we can specify the time to wait for the result, it’s useful to avoid current thread getting blocked for longer time. There are **isDone()** and **isCancelled()** methods to find out the current status of associated Callable task.

Here is a simple example of Java Callable task that returns the name of thread executing the task after one second. We are using [Executor framework](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) to execute 100 tasks in parallel and use Java Future to get the result of the submitted tasks.

package com.journaldev.threads;  
  
import java.util.ArrayList;  
import java.util.Date;  
import java.util.List;  
import java.util.concurrent.Callable;  
import java.util.concurrent.ExecutionException;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.Future;  
  
public class MyCallable implements Callable<String> {  
  
 @Override  
 public String call() throws Exception {  
 Thread.sleep(1000);  
 //return the thread name executing this callable task  
 return Thread.currentThread().getName();  
 }  
   
 public static void main(String args[]){  
 //Get ExecutorService from Executors utility class, thread pool size is 10  
 ExecutorService executor = Executors.newFixedThreadPool(10);  
 //create a list to hold the Future object associated with Callable  
 List<Future<String>> list = new ArrayList<Future<String>>();  
 //Create MyCallable instance  
 Callable<String> callable = new MyCallable();  
 for(int i=0; i< 100; i++){  
 //submit Callable tasks to be executed by thread pool  
 Future<String> future = executor.submit(callable);  
 //add Future to the list, we can get return value using Future  
 list.add(future);  
 }  
 for(Future<String> fut : list){  
 try {  
 //print the return value of Future, notice the output delay in console  
 // because Future.get() waits for task to get completed  
 System.out.println(new Date()+ "::"+fut.get());  
 } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
 //shut down the executor service now  
 executor.shutdown();  
 }  
  
}

Once we execute the above program, you will notice the delay in output because java Future get() method waits for the java callable task to complete. Also notice that there are only 10 threads executing these tasks.

Here is snippet of the output of above program.

Mon Dec 31 20:40:15 PST 2012::pool-1-thread-1  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-2  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-3  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-4  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-5  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-6  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-7  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-8  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-9  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-10  
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-2  
...

**Tip**: What if we want to override some of the methods of Java Future interface, for example overriding get()method to timeout after some default time rather than waiting indefinitely, in this case **Java FutureTask** class comes handy that is the base implementation of Future interface. Check out [**Java FutureTask Example**](https://www.journaldev.com/1650/java-futuretask-example-program) to learn more about this class.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

# ThreadPoolExecutor – Java Thread Pool Example

APRIL 3, 2018 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [52 COMMENTS](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice#comments)

[**Java thread**](https://www.journaldev.com/1079/multithreading-in-java) **pool** manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. We can use ThreadPoolExecutor to create thread pool in java.

Java thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue. **java.util.concurrent.Executors** provide implementation of **java.util.concurrent.Executor**interface to create the thread pool in java. Let’s write a simple program to explain it’s working.

First we need to have a Runnable class, named WorkerThread.java

package com.journaldev.threadpool;  
  
public class WorkerThread implements Runnable {  
   
 private String command;  
   
 public WorkerThread(String s){  
 this.command=s;  
 }  
  
 @Override  
 public void run() {  
 System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);  
 processCommand();  
 System.out.println(Thread.currentThread().getName()+" End.");  
 }  
  
 private void processCommand() {  
 try {  
 Thread.sleep(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public String toString(){  
 return this.command;  
 }  
}

## ExecutorService Example

Here is the test program class SimpleThreadPool.java, where we are creating fixed thread pool from **Executors framework**.

package com.journaldev.threadpool;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class SimpleThreadPool {  
  
 public static void main(String[] args) {  
 ExecutorService executor = Executors.newFixedThreadPool(5);  
 for (int i = 0; i < 10; i++) {  
 Runnable worker = new WorkerThread("" + i);  
 executor.execute(worker);  
 }  
 executor.shutdown();  
 while (!executor.isTerminated()) {  
 }  
 System.out.println("Finished all threads");  
 }  
}

In above program, we are creating fixed size thread pool of 5 worker threads. Then we are submitting 10 jobs to this pool, since the pool size is 5, it will start working on 5 jobs and other jobs will be in wait state, as soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and get’s executed.

Here is the output of the above program.

pool-1-thread-2 Start. Command = 1  
pool-1-thread-4 Start. Command = 3  
pool-1-thread-1 Start. Command = 0  
pool-1-thread-3 Start. Command = 2  
pool-1-thread-5 Start. Command = 4  
pool-1-thread-4 End.  
pool-1-thread-5 End.  
pool-1-thread-1 End.  
pool-1-thread-3 End.  
pool-1-thread-3 Start. Command = 8  
pool-1-thread-2 End.  
pool-1-thread-2 Start. Command = 9  
pool-1-thread-1 Start. Command = 7  
pool-1-thread-5 Start. Command = 6  
pool-1-thread-4 Start. Command = 5  
pool-1-thread-2 End.  
pool-1-thread-4 End.  
pool-1-thread-3 End.  
pool-1-thread-5 End.  
pool-1-thread-1 End.  
Finished all threads

The output confirms that there are five threads in the pool named from “pool-1-thread-1” to “pool-1-thread-5” and they are responsible to execute the submitted tasks to the pool.

## ThreadPoolExecutor Example

**Executors** class provide simple implementation of **ExecutorService** using **ThreadPoolExecutor** but ThreadPoolExecutor provides much more feature than that. We can specify the number of threads that will be alive when we create ThreadPoolExecutor instance and we can limit the size of thread pool and create our own **RejectedExecutionHandler** implementation to handle the jobs that can’t fit in the worker queue.

Here is our custom implementation of RejectedExecutionHandler interface.

package com.journaldev.threadpool;  
  
import java.util.concurrent.RejectedExecutionHandler;  
import java.util.concurrent.ThreadPoolExecutor;  
  
public class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {  
  
 @Override  
 public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  
 System.out.println(r.toString() + " is rejected");  
 }  
  
}

ThreadPoolExecutor provides several methods using which we can find out the current state of executor, pool size, active thread count and task count. So I have a monitor thread that will print the executor information at certain time interval.

package com.journaldev.threadpool;  
  
import java.util.concurrent.ThreadPoolExecutor;  
  
public class MyMonitorThread implements Runnable  
{  
 private ThreadPoolExecutor executor;  
 private int seconds;  
 private boolean run=true;  
  
 public MyMonitorThread(ThreadPoolExecutor executor, int delay)  
 {  
 this.executor = executor;  
 this.seconds=delay;  
 }  
 public void shutdown(){  
 this.run=false;  
 }  
 @Override  
 public void run()  
 {  
 while(run){  
 System.out.println(  
 String.format("[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s",  
 this.executor.getPoolSize(),  
 this.executor.getCorePoolSize(),  
 this.executor.getActiveCount(),  
 this.executor.getCompletedTaskCount(),  
 this.executor.getTaskCount(),  
 this.executor.isShutdown(),  
 this.executor.isTerminated()));  
 try {  
 Thread.sleep(seconds\*1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
   
 }  
}

Here is the thread pool implementation example using **ThreadPoolExecutor**.

package com.journaldev.threadpool;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ThreadFactory;  
import java.util.concurrent.ThreadPoolExecutor;  
import java.util.concurrent.TimeUnit;  
  
public class WorkerPool {  
  
 public static void main(String args[]) throws InterruptedException{  
 //RejectedExecutionHandler implementation  
 RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandlerImpl();  
 //Get the ThreadFactory implementation to use  
 ThreadFactory threadFactory = Executors.defaultThreadFactory();  
 //creating the ThreadPoolExecutor  
 ThreadPoolExecutor executorPool = new ThreadPoolExecutor(2, 4, 10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(2), threadFactory, rejectionHandler);  
 //start the monitoring thread  
 MyMonitorThread monitor = new MyMonitorThread(executorPool, 3);  
 Thread monitorThread = new Thread(monitor);  
 monitorThread.start();  
 //submit work to the thread pool  
 for(int i=0; i<10; i++){  
 executorPool.execute(new WorkerThread("cmd"+i));  
 }  
   
 Thread.sleep(30000);  
 //shut down the pool  
 executorPool.shutdown();  
 //shut down the monitor thread  
 Thread.sleep(5000);  
 monitor.shutdown();  
   
 }  
}

Notice that while initializing the ThreadPoolExecutor, we are keeping initial pool size as 2, maximum pool size to 4 and work queue size as 2. So if there are 4 running tasks and more tasks are submitted, the work queue will hold only 2 of them and rest of them will be handled by RejectedExecutionHandlerImpl.

Here is the output of above program that confirms above statement.

pool-1-thread-1 Start. Command = cmd0  
pool-1-thread-4 Start. Command = cmd5  
cmd6 is rejected  
pool-1-thread-3 Start. Command = cmd4  
pool-1-thread-2 Start. Command = cmd1  
cmd7 is rejected  
cmd8 is rejected  
cmd9 is rejected  
[monitor] [0/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false  
pool-1-thread-4 End.  
pool-1-thread-1 End.  
pool-1-thread-2 End.  
pool-1-thread-3 End.  
pool-1-thread-1 Start. Command = cmd3  
pool-1-thread-4 Start. Command = cmd2  
[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false  
pool-1-thread-1 End.  
pool-1-thread-4 End.  
[monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false  
[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true  
[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

Notice the change in active, completed and total completed task count of the executor. We can invoke **shutdown()** method to finish execution of all the submitted tasks and terminate the thread pool.

If you want to schedule a task to run with delay or periodically then you can use **ScheduledThreadPoolExecutor** class. Read more about them at [**Java Schedule Thread Pool Executor**](https://www.journaldev.com/2340/java-scheduler-scheduledexecutorservice-scheduledthreadpoolexecutor-example).

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**Java BlockingQueue Example**  
  
  
Today we will look into Java BlockingQueue. java.util.concurrent.BlockingQueue is a java Queue that support operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.  
  
Table of Contents [hide]  
  
1 Java BlockingQueue  
1.1 Java BlockingQueue Example – Message  
1.2 Java BlockingQueue Example – Producer  
1.3 Java BlockingQueue Example – Consumer  
1.4 Java BlockingQueue Example – Service  
Java BlockingQueue  
Java BlockingQueue  
Java BlockingQueue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.  
  
Java BlockingQueue implementations are thread-safe. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.  
  
Java BlockingQueue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem. We don’t need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue because it’s handled by implementation classes of BlockingQueue.  
  
Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.  
  
While implementing producer consumer problem in BlockingQueue, we will use ArrayBlockingQueue implementation. Following are some important methods you should know.  
  
  
put(E e): This method is used to insert elements to the queue. If the queue is full, it waits for the space to be available.  
E take(): This method retrieves and remove the element from the head of the queue. If queue is empty it waits for the element to be available.  
Let’s implement producer consumer problem using java BlockingQueue now.  
  
Java BlockingQueue Example – Message  
Just a normal java object that will be produced by Producer and added to the queue. You can also call it as payload or queue message.  
  
  
package com.journaldev.concurrency;  
  
public class Message {  
 private String msg;  
   
 public Message(String str){  
 this.msg=str;  
 }  
  
 public String getMsg() {  
 return msg;  
 }  
  
}  
Java BlockingQueue Example – Producer  
Producer class that will create messages and put it in the queue.  
  
  
  
package com.journaldev.concurrency;  
  
import java.util.concurrent.BlockingQueue;  
  
public class Producer implements Runnable {  
  
 private BlockingQueue<Message> queue;  
   
 public Producer(BlockingQueue<Message> q){  
 this.queue=q;  
 }  
 @Override  
 public void run() {  
 //produce messages  
 for(int i=0; i<100; i++){  
 Message msg = new Message(""+i);  
 try {  
 Thread.sleep(i);  
 queue.put(msg);  
 System.out.println("Produced "+msg.getMsg());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 //adding exit message  
 Message msg = new Message("exit");  
 try {  
 queue.put(msg);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
}  
Java BlockingQueue Example – Consumer  
Consumer class that will process on the messages from the queue and terminates when exit message is received.  
  
  
package com.journaldev.concurrency;  
  
import java.util.concurrent.BlockingQueue;  
  
public class Consumer implements Runnable{  
  
private BlockingQueue<Message> queue;  
   
 public Consumer(BlockingQueue<Message> q){  
 this.queue=q;  
 }  
  
 @Override  
 public void run() {  
 try{  
 Message msg;  
 //consuming messages until exit message is received  
 while((msg = queue.take()).getMsg() !="exit"){  
 Thread.sleep(10);  
 System.out.println("Consumed "+msg.getMsg());  
 }  
 }catch(InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
Java BlockingQueue Example – Service  
Finally we have to create BlockingQueue service for producer and consumer. This producer consumer service will create the BlockingQueue with fixed size and share with both producers and consumers. This service will start producer and consumer threads and exit.  
  
  
   
  
package com.journaldev.concurrency;  
  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
  
public class ProducerConsumerService {  
  
 public static void main(String[] args) {  
 //Creating BlockingQueue of size 10  
 BlockingQueue<Message> queue = new ArrayBlockingQueue<>(10);  
 Producer producer = new Producer(queue);  
 Consumer consumer = new Consumer(queue);  
 //starting producer to produce messages in queue  
 new Thread(producer).start();  
 //starting consumer to consume messages from queue  
 new Thread(consumer).start();  
 System.out.println("Producer and Consumer has been started");  
 }  
  
}  
Output of the above java BlockingQueue example program is shown below.  
  
  
Producer and Consumer has been started  
Produced 0  
Produced 1  
Produced 2  
Produced 3  
Produced 4  
Consumed 0  
Produced 5  
Consumed 1  
Produced 6  
Produced 7  
Consumed 2  
Produced 8  
...  
Java Thread sleep is used in producer and consumer to produce and consume messages with some delay.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

# Java Thread wait, notify and notifyAll Example

APRIL 2, 2018 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [52 COMMENTS](https://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example#comments)

The Object class in java contains three final methods that allows threads to communicate about the lock status of a resource. These methods are **wait()**, **notify()** and **notifyAll()**. So today we will look into wait, notify and notifyAll in java program.

## wait, notify and notifyAll in Java

The current thread which invokes these methods on any object should have the object **monitor** else it throws **java.lang.IllegalMonitorStateException** exception.

### wait

Object wait methods has three variance, one which waits indefinitely for any other thread to call notify or notifyAll method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.

### notify

notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.

### notifyAll

notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

These methods can be used to implement [producer consumer problem](https://www.journaldev.com/1034/java-blockingqueue-example) where consumer threads are waiting for the objects in Queue and producer threads put object in queue and notify the waiting threads.

Let’s see an example where multiple threads work on the same object and we use wait, notify and notifyAll methods.

### Message

A java bean class on which threads will work and call wait and notify methods.

package com.journaldev.[concurrency](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers);  
  
public class Message {  
 private String msg;  
   
 public Message(String str){  
 this.msg=str;  
 }  
  
 public String getMsg() {  
 return msg;  
 }  
  
 public void setMsg(String str) {  
 this.msg=str;  
 }  
  
}

### Waiter

A class that will wait for other threads to invoke notify methods to complete it’s processing. Notice that Waiter thread is owning monitor on Message object using synchronized block.

package com.journaldev.concurrency;  
  
public class Waiter implements Runnable{  
   
 private Message msg;  
   
 public Waiter(Message m){  
 this.msg=m;  
 }  
  
 @Override  
 public void run() {  
 String name = Thread.currentThread().getName();  
 synchronized (msg) {  
 try{  
 System.out.println(name+" waiting to get notified at time:"+System.currentTimeMillis());  
 msg.wait();  
 }catch(InterruptedException e){  
 e.printStackTrace();  
 }  
 System.out.println(name+" waiter thread got notified at time:"+System.currentTimeMillis());  
 //process the message now  
 System.out.println(name+" processed: "+msg.getMsg());  
 }  
 }  
  
}

### Notifier

A class that will process on Message object and then invoke notify method to wake up threads waiting for Message object. Notice that synchronized block is used to own the monitor of Message object.

package com.journaldev.concurrency;  
  
public class Notifier implements Runnable {  
  
 private Message msg;  
   
 public Notifier(Message msg) {  
 this.msg = msg;  
 }  
  
 @Override  
 public void run() {  
 String name = Thread.currentThread().getName();  
 System.out.println(name+" started");  
 try {  
 Thread.sleep(1000);  
 synchronized (msg) {  
 msg.setMsg(name+" Notifier work done");  
 msg.notify();  
 // msg.notifyAll();  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
   
 }  
  
}

### WaitNotifyTest

Test class that will create multiple threads of Waiter and Notifier and start them.

package com.journaldev.concurrency;  
  
public class WaitNotifyTest {  
  
 public static void main(String[] args) {  
 Message msg = new Message("process it");  
 Waiter waiter = new Waiter(msg);  
 new Thread(waiter,"waiter").start();  
   
 Waiter waiter1 = new Waiter(msg);  
 new Thread(waiter1, "waiter1").start();  
   
 Notifier notifier = new Notifier(msg);  
 new Thread(notifier, "notifier").start();  
 System.out.println("All the threads are started");  
 }  
  
}

When we will invoke the above program, we will see below output but program will not complete because there are two threads waiting on Message object and notify() method has wake up only one of them, the other thread is still waiting to get notified.

waiter waiting to get notified at time:1356318734009  
waiter1 waiting to get notified at time:1356318734010  
All the threads are started  
notifier started  
waiter waiter thread got notified at time:1356318735011  
waiter processed: notifier Notifier work done

If we comment the notify() call and uncomment the notifyAll() call in Notifier class, below will be the output produced.

waiter waiting to get notified at time:1356318917118  
waiter1 waiting to get notified at time:1356318917118  
All the threads are started  
notifier started  
waiter1 waiter thread got notified at time:1356318918120  
waiter1 processed: notifier Notifier work done  
waiter waiter thread got notified at time:1356318918120  
waiter processed: notifier Notifier work done

Since notifyAll() method wake up both the Waiter threads and program completes and terminates after execution. That’s all for wait, notify and notifyAll in java.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////

### How to make code Thread-Safe in Java

There are multiple ways to make this code thread safe in Java:

1) Use [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and lock the getCount() method so that only one thread can execute it at a time which removes possibility of coinciding or interleaving.

2) use **Atomic Integer**, which makes this ++ operation atomic and since **atomic operations are thread-safe** and saves cost of external synchronization.

here is a thread-safe version of Counter class in Java:

/\*

\* **Thread-Safe Example in Java**

\*/

public class Counter {

private int count;

AtomicInteger atomicCount = new AtomicInteger( 0 );

/\*

\* **This method thread-safe now because of locking and synchornization**

\*/

public synchronized int getCount(){

return count++;

}

/\*

\* **This method is thread-safe because count is incremented atomically**

\*/

public int getCountAtomically(){

return atomicCount.incrementAndGet();

}

}

### Important points about Thread-Safety in Java

Here is some points worth remembering to **write thread safe code in Java**, these knowledge also helps you to avoid some serious concurrency issues in Java like race condition or [deadlock in Java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html):

1) Immutable objects are by default thread-safe because there state can not be modified once created. Since String is immutable in Java, its inherently thread-safe.

2) Read only or [final variables in Java](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) are also thread-safe in Java.

3) Locking is one way of achieving thread-safety in Java.

4) [Static variables](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) if not synchronized properly becomes major cause of thread-safety issues.

5) Example of thread-safe class in Java: Vector, Hashtable, ConcurrentHashMap, String etc.

6) Atomic operations in Java are thread-safe e.g. reading a 32 bit int from memory because its an atomic operation it can't interleave with other thread.

7) local variables are also thread-safe because each thread has there own copy and using local variables is good way to writing thread-safe code in Java.

8) In order to avoid thread-safety issue minimize sharing of objects between multiple thread.

9) [Volatile keyword in Java](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective.

That’s all on **how to write thread safe class or code in Java** and avoid serious concurrency issues in Java. To be frank thread-safety is a little tricky concept to grasp, you need to think concurrently in order to catch whether a code is thread-safe or not. Also [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) plays a spoiler since it can **reorder code** for optimization, so the code which looks sequential and runs fine in development environment not guaranteed to run similarly in production environment because JVM may ergonomically adjust itself as server JVM and perform more optimization and reorder which cause **thread-safety issues**.

Read more: <https://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html#ixzz5LnLO3gRB>

/////////////////////////////////////////////////////////////////////////////////////////////////////////

````````````````WHAT IS RACE CONDITION IN <MULTITHREADING``````````````

. Race conditions occurs when two thread operate on same object without proper synchronization and there operation interleaves on each other. Classical example of Race condition is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write. if two threads tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread. atomic operations are not subject to race conditions because those operation cannot be interleaved.

//////////////////////////////////////////////////////////

--------------> Java and Multiple Inheritance  
Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.   
Why Java doesn't support Multiple Inheritance?  
  
Consider the below Java code. It shows error.  
 // First Parent class  
class Parent1  
{  
 void fun()  
 {  
 System.out.println("Parent1");  
 }  
}  
// Second Parent Class  
class Parent2  
{  
 void fun()  
 {  
 System.out.println("Parent2");  
 }  
}  
// Error : Test is inheriting from multiple  
// classes  
class Test extends Parent1, Parent2  
{  
 public static void main(String args[])  
 {  
 Test t = new Test();  
 t.fun();  
 }  
}  
Output :  
Compiler Error  
  
From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1’s fun() or Parent2’s fun() method.   
1. The Diamond Problem:   
 GrandParent  
 / \  
 / \  
 Parent1 Parent2  
 \ /  
 \ /  
 Test  
  
 // A Grand parent class in diamond  
class GrandParent  
{  
 void fun()  
 {  
 System.out.println("Grandparent");  
 }  
}  
// First Parent class  
class Parent1 extends GrandParent  
{  
 void fun()  
 {  
 System.out.println("Parent1");  
 }  
}  
// Second Parent Class  
class Parent2 extends GrandParent  
{  
 void fun()  
 {  
 System.out.println("Parent2");  
 }  
}  
// Error : Test is inheriting from multiple  
// classes  
class Test extends Parent1, Parent2  
{  
 public static void main(String args[])  
 {  
 Test t = new Test();  
 t.fun();  
 }  
}  
From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1’s fun() or Beta’s fun() method.   
Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.  
2. Simplicity - Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.  
   
How are above problems handled for Default Methods and Interfaces ?  
  
Java 8 supports default methods where interfaces can provide default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.   
 // A simple Java program to demonstrate multiple  
// inheritance through default methods.  
interface PI1  
{  
 // default method  
 default void show()  
 {  
 System.out.println("Default PI1");  
 }  
}  
interface PI2  
{  
 // Default method  
 default void show()  
 {  
 System.out.println("Default PI2");  
 }  
}  
// Implementation class code  
class TestClass implements PI1, PI2  
{  
 // Overriding default show method  
 public void show()  
 {  
 // use super keyword to call the show  
 // method of PI1 interface  
 PI1.super.show();  
 // use super keyword to call the show  
 // method of PI2 interface  
 PI2.super.show();  
 }  
 public static void main(String args[])  
 {  
 TestClass d = new TestClass();  
 d.show();  
 }  
}  
Output:  
Default PI1  
Default PI2  
  
If we remove implementation of default method from "TestClass", we get compiler error. See this for a sample run.  
If there is a diamond through interfaces, then there is no issue if none of the middle interfaces provide implementation of root interface. If they provide implementation, then implementation can be accessed as above using super keyword.  
 // A simple Java program to demonstrate how diamond  
// problem is handled in case of default methods  
interface GPI  
{  
 // default method  
 default void show()  
 {  
 System.out.println("Default GPI");  
 }  
}  
interface PI1 extends GPI { }  
interface PI2 extends GPI { }  
// Implementation class code  
class TestClass implements PI1, PI2  
{  
 public static void main(String args[])  
 {  
 TestClass d = new TestClass();  
 d.show();  
 }  
}  
Output:  
Default GPI  
//////////////////////////////////////////////////////////////////

Why java not 100% object oriented

Why Java is not a purely Object-Oriented Language?  
Pure Object Oriented Language or Complete Object Oriented Language are Fully Object Oriented Language which supports or have features which treats everything inside program as objects. It doesn’t support primitive datatype(like int, char, float, bool, etc.). There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:  
  
Encapsulation/Data Hiding  
Inheritance  
Polymorphism  
Abstraction  
All predefined types are objects  
All user defined types are objects  
All operations performed on objects must be only through methods exposed at the objects.  
Example: Smalltalk  
  
**Why Java is not a Pure Object Oriented Language?**  
  
  
Java supports property 1, 2, 3, 4 and 6 but fails to support property 5 and 7 given above. Java language is not a Pure Object Oriented Language as it contain these properties:  
  
Primitive Data Type ex. int, long, bool, float, char, etc as Objects: Smalltalk is a “pure” object-oriented programming language unlike Java and C++ as there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects.  
In Java, we have predefined types as non-objects (primitive types).  
int a = 5;   
System.out.print(a);  
The static keyword: When we declares a class as static then it can be used without the use of an object in Java. If we are using static function or static variable then we can’t call that function or variable by using dot(.) or class object defying object oriented feature.  
Wrapper Class: Wrapper class provides the mechanism to convert primitive into object and object into primitive. In Java, you can use Integer, Float etc. instead of int, float etc. We can communicate with objects without calling their methods. ex. using arithmetic operators.  
String s1 = "ABC" + "A" ;  
Even using Wrapper classes does not make Java a pure OOP language, as internally it will use the operations like Unboxing and Autoboxing. So if you create instead of int Integer and do any mathematical operation on it, under the hoods Java is going to use primitive type int only.  
  
public class BoxingExample   
{   
 public static void main(String[] args)   
 {   
 Integer i = new Integer(10);   
 Integer j = new Integer(20);   
 Integer k = new Integer(i.intValue() + j.intValue());   
 System.out.println("Output: "+ k);   
 }   
}   
In the above code, there are 2 problems where Java fails to work as pure OOP:  
  
  
  
While creating Integer class you are using primitive type “int” i.e. numbers 10, 20.  
While doing addition Java is using primitive type “int”.

////////////////////////////////////////////////////////////////

Thread class not overriding run will it work  
  
Implement and without using &&

////////////////////////////////////////////////////////////////

**Use of wrapper class**

Wrapper classes allows us to convert the primitive types into an object type.  
 java is not 100% object oriented programming language because of the 8 primitive types. Then wrapper classes are introduced to give the primitive types an object form. So the primitive types can also be stored as an object of its respective wrapper class  
 The 8 primitive types and its wrapper classes are,  
byte. - Byte  
int - Integer  
short - Short  
long - Long  
float - Float  
double - Double  
char - Character  
boolean - Boolean  
  
all this wrapper classes are available in java.lang package  
 Now if you want to store an integer as an object type you can write it as  
 Integer i=new Integer(10);  
 This is known as Boxing, converting a primitive type into an object.  
 Now to get that integer value back,  
 int a=i.intValue();  
This operation is known as Unboxing converting the value of a wrapper class object into a primitive type.  
After java 1.5/5 Boxing and Unboxing became automatic. You can directly assign the primitive as an object of wrapper class.  
 Integer i=10;  
  
Before java 1.5/5, databases stores only Object class objects so to store a primitive type into an ArrayList or Vector wrapper classes are used. Primitive types are converted into wrapper class object through boxing and upcasted to Object type to store in the database.  
  
The significance of wrapper class comes when you want to write a program which will work with any type of value. To write such a program declare the arguments as Object type since Object class is extended by all the other class in java Object class type can store any kind of objects.  
 Lets take an example, you want to write a program which will work if you pass any kind of values   
 public static void printHello(Object a)   
 {  
 System.out.println("Hello");  
 }  
This program will work with all kind of values. Since Object class is the super most class, Object type variable can accept any objects. the wrapper class comea into picture when you pass a primitive type value. Imagine you are calling the function with a value 10  
 printHello(10);  
Now the 10 will be boxed and becomes an object of Integer class(wrapper class) since Integer extends Object the method will work fine.  
 There are many other uses which comes while overriding the build in methods like compare(), equals() etc because all these functions have Object type as parameters.

//////////////////////////////////////////////////////////////////  
  
**Difference between notify() and notifyAll() in Java**  
notify() and notifyAll() methods with wait() method are used to for communication between the threads. A thread which goes into waiting state by calling wait() method will be in waiting state until any other thread calls either notify() or notifyAll() method on the same object.  
Now the question is both notify() and notifyAll() method is used to give notification to the waiting thread, then what is the difference between them or where we should use notify() method and where we should go for notifyAll() method?  
Lets understand how notify() method behaves:  
  
// Java program to illustrate the   
// behaviour of notify() method   
class Geek1 extends Thread {   
public void run()   
 {   
 synchronized(this)   
 {   
 System.out.println   
 (Thread.currentThread().getName() + "...starts");   
 try {   
 this.wait();   
 }   
 catch (InterruptedException e) {   
 e.printStackTrace();   
 }   
 System.out.println   
 (Thread.currentThread().getName() + "...notified");   
 }   
 }   
} class Geek2 extends Thread {   
 Geek1 geeks1;   
 Geek2(Geek1 geeks1)   
 {   
 this.geeks1 = geeks1;   
 }   
public void run()   
 {   
 synchronized(this.geeks1)   
 {   
 System.out.println   
 (Thread.currentThread().getName() + "...starts");   
   
 try {   
 this.geeks1.wait();   
 }   
 catch (InterruptedException e) {   
 e.printStackTrace();   
 }   
 System.out.println   
 (Thread.currentThread().getName() + "...notified");   
 }   
 }   
} class Geek3 extends Thread {   
 Geek1 geeks1;   
 Geek3(Geek1 geeks1)   
 {   
 this.geeks1 = geeks1;   
 }   
public void run()   
 {   
 synchronized(this.geeks1)   
 {   
 System.out.println   
 (Thread.currentThread().getName() + "...starts");   
 this.geeks1.notify();   
 System.out.println   
 (Thread.currentThread().getName() + "...notified");   
 }   
 }   
} class MainClass {   
public static void main(String[] args) throws InterruptedException   
 {   
   
 Geek1 geeks1 = new Geek1();   
 Geek2 geeks2 = new Geek2(geeks1);   
 Geek3 geeks3 = new Geek3(geeks1);   
 Thread t1 = new Thread(geeks1, "Thread-1");   
 Thread t2 = new Thread(geeks2, "Thread-2");   
 Thread t3 = new Thread(geeks3, "Thread-3");   
 t1.start();   
 t2.start();   
 Thread.sleep(100);   
 t3.start();   
 }   
}  
Output:  
  
Thread-1...start  
Thread-2...starts  
Thread-3...starts  
Thread-3...notified  
Thread-1...notified  
Lets understand how notifyAll() method behaves:  
  
  
  
// Java program to illustrate the   
// behavior of notifyAll() method   
class Geek1 extends Thread {   
public void run()   
 {   
 synchronized(this)   
 {   
 System.out.println   
 (Thread.currentThread().getName() + "...starts");   
 try {   
 this.wait();   
 }   
 catch (InterruptedException e) {   
 e.printStackTrace();   
 }   
 System.out.println   
 (Thread.currentThread().getName() + "...notified");   
 }   
 }   
} class Geek2 extends Thread {   
 Geek1 geeks1;   
 Geek2(Geek1 geeks1)   
 {   
 this.geeks1 = geeks1;   
 }   
public void run()   
 {   
 synchronized(this.geeks1)   
 {   
 System.out.println   
 (Thread.currentThread().getName() + "...starts");   
   
 try {   
 this.geeks1.wait();   
 }   
 catch (InterruptedException e) {   
 e.printStackTrace();   
 }   
 System.out.println   
 (Thread.currentThread().getName() + "...notified");   
 }   
 }   
} class Geek3 extends Thread {   
 Geek1 geeks1;   
 Geek3(Geek1 geeks1)   
 {   
 this.geeks1 = geeks1;   
 }   
public void run()   
 {   
 synchronized(this.geeks1)   
 {   
 System.out.println   
 (Thread.currentThread().getName() + "...starts");   
   
 this.geeks1.notifyAll();   
 System.out.println   
 (Thread.currentThread().getName() + "...notified");   
 }   
 }   
} class MainClass {   
public static void main(String[] args) throws InterruptedException   
 {   
   
 Geek1 geeks1 = new Geek1();   
 Geek2 geeks2 = new Geek2(geeks1);   
 Geek3 geeks3 = new Geek3(geeks1);   
 Thread t1 = new Thread(geeks1, "Thread-1");   
 Thread t2 = new Thread(geeks2, "Thread-2");   
 Thread t3 = new Thread(geeks3, "Thread-3");   
 t1.start();   
 t2.start();   
 Thread.sleep(100);   
 t3.start();   
 }   
}  
Output:  
  
Thread-1...starts  
Thread-2...starts  
Thread-3...starts  
Thread-3...notified  
Thread-2...notified  
Thread-1...notified  
Differences between notify() and notifyAll()  
  
Notification to number of threads : We can use notify() method to give the notification for only one thread which is waiting for a particular object whereas by the help of notifyAll() methods we can give the notification to all waiting threads of a particular object.  
Notifying a thread by JVM : If multiple threads are waiting for the notification and we use notify() method then only one thread get the notification and the remaining thread have to wait for further notification. Which thread will get the notification we can’t expect because it totally depends upon the JVM. But when we use notifyAll() method then multiple threads got the notification but execution of threads will be performed one by one because thread requires lock and only one lock is available for one object.  
Interchangeability of threads : We should go for notify() if all your waiting threads are interchangeable (the order they wake up doesn’t matter). A common example is a thread pool. But we should use notifyAll() for other cases where the waiting threads may have different purposes and should be able to run concurrently. An example is a maintenance operation on a shared resource, where multiple threads are waiting for the operation to complete before accessing the resource.  
When to use notify() method and notifyAll()  
  
In case of mutually exclusive locking, only one of the waiting threads can do something useful after being notified (in this case acquire the lock). In such a case, you would rather use notify(). Properly implemented, you could use notifyAll() in this situation as well, but you would unnecessarily wake threads that can’t do anything anyway.  
In some cases, all waiting threads can take useful action once the wait finishes. An example would be a set of threads waiting for a certain task to finish; once the task has finished, all waiting threads can continue with their business. In such a case you would use notifyAll() to wake up all waiting threads at the same time.  
Applications of notify() and notifyAll()  
  
  
  
A maintenance operation on a shared resource, where multiple threads are waiting for the operation to complete before accessing the resource; for these we should go for notifyAll().  
Let’s say we have a producer thread and a consumer thread. Each “packet” produced by the producer should be consumed by a consumer. The consumer puts something in a queue and then calls notify().  
We want to have a notification when a lengthy process has finished. You want a beep and a screen update. The process performs notifyAll() to notify both the beeping-thread and the screen-update-thread.

///////////////////////////////////////////////////////////////////

--------------->Jar vs war

---->.jar files: The .jar files contain libraries, resources and accessories files like property files.

.war files: The war file contains the web application that can be deployed on any servlet/jsp container. The .war file contains jsp, html, javascript and other files necessary for the development of web applications.

/////////////////////////////////////////////////////////////////////

------->Finalize()

``````methd called by garbage collector java just before

object has to destroy object, to release resources associated with object

like Database Connection, Network Connection

ince Object class contains finalize method hence finalize method is available for every java class since Object is superclass of all java classes. Since it is available for every java class hence Garbage Collector can call finalize method on any java object

class Hello {

public static void main(String[] args)

{

// Requesting JVM to call Garbage Collector method

System.gc();

Hello s = new Hello();

s = null;

public void finalize()

{

System.out.println("finalize method overriden");

}

}

o/p

finalize method overriden

Main Completes

///////////////////////////////////////////////////////////////////////////////////

---------->Producer-Consumer solution using threads in Java

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

The producer’s job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Recommended Reading- Multithreading in JAVA, Synchronized in JAVA , Inter-thread Communication

Implementation of Producer Consumer Class

A LinkedList list – to store list of jobs in queue.

A Variable Capacity – to check for if the list is full or not

A mechanism to control the insertion and extraction from this list so that we do not insert into list if it is full or remove from it if it is empty.

Note: It is recommended to test the below program on a offline IDE as infinite loops and sleep method may lead to it time out on any online IDE

// Java program to implement solution of producer

// consumer problem.

import java.util.LinkedList;

public class Threadexample

{

public static void main(String[] args)

throws InterruptedException

{

// Object of a class that has both produce()

// and consume() methods

final PC pc = new PC();

// Create producer thread

Thread t1 = new Thread(new Runnable()

{

@Override

public void run()

{

try

{

pc.produce();

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

});

// Create consumer thread

Thread t2 = new Thread(new Runnable()

{

@Override

public void run()

{

try

{

pc.consume();

}

catch(InterruptedException e)

{

e.printStackTrace();

}

}

});

// Start both threads

t1.start();

t2.start();

// t1 finishes before t2

t1.join();

t2.join();

}

// This class has a list, producer (adds items to list

// and consumber (removes items).

public static class PC

{

// Create a list shared by producer and consumer

// Size of list is 2.

LinkedList<Integer> list = new LinkedList<>();

int capacity = 2;

// Function called by producer thread

public void produce() throws InterruptedException

{

int value = 0;

while (true)

{

synchronized (this)

{

// producer thread waits while list

// is full

while (list.size()==capacity)

wait();

System.out.println("Producer produced-"

+ value);

// to insert the jobs in the list

list.add(value++);

// notifies the consumer thread that

// now it can start consuming

notify();

// makes the working of program easier

// to understand

Thread.sleep(1000);

}

}

}

// Function called by consumer thread

public void consume() throws InterruptedException

{

while (true)

{

synchronized (this)

{

// consumer thread waits while list

// is empty

while (list.size()==0)

wait();

//to retrive the ifrst job in the list

int val = list.removeFirst();

System.out.println("Consumer consumed-"

+ val);

// Wake up producer thread

notify();

// and sleep

Thread.sleep(1000);

}

}

}

}

}

Run on IDE

Output:

Producer produced-0

Producer produced-1

Consumer consumed-0

Consumer consumed-1

Producer produced-2

Important Points

In PC class (A class that has both produce and consume methods), a linked list of jobs and a capacity of the list is added to check that producer does not produce if the list is full.

In Producer class, the value is initialized as 0.

Also, we have an infinite outer loop to insert values in the list. Inside this loop, we have a synchronized block so that only a producer or a consumer thread runs at a time.

An inner loop is there before adding the jobs to list that checks if the job list is full, the producer thread gives up the intrinsic lock on PC and goes on the waiting state.

If the list is empty, the control passes to below the loop and it adds a value in the list.

In the Consumer class, we again have an infinite loop to extract a value from the list.

Inside, we also have an inner loop which checks if the list is empty.

If it is empty then we make the consumer thread give up the lock on PC and passes the control to producer thread for producing more jobs.

If the list is not empty, we go round the loop and removes an item from the list.

In both the methods, we use notify at the end of all statements. The reason is simple, once you have something in list, you can have the consumer thread consume it, or if you have consumed something, you can have the producer produce something.

sleep() at the end of both methods just make the output of program run in step wise manner and not display everything all at once so that you can see what actually is happening in the program.

Exercise :

Readers are advised to use if condition in place of inner loop for checking boundary conditions.

Try to make your program produce one item and immediately after that the consumer consumes it before any other item is produced by the consumer.

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**Spring batch**

**Spring Batch** Introduction. ... **Batch processing** is used to **process** billions of transactions every day for enterprises. **Spring Batch** is a lightweight, comprehensive **batch** framework designed to enable the development of robust**batch** applications vital for the daily operations of enterprise systems.  
  
DROP TABLE people IF EXISTS;  
  
CREATE TABLE people (  
 person\_id BIGINT IDENTITY NOT NULL PRIMARY KEY,  
 first\_name VARCHAR(20),  
 last\_name VARCHAR(20)  
);  
Spring Boot runs schema-@@platform@@.sql automatically during startup. -all is the default for all platforms.  
Create a business class  
Now that you see the format of data inputs and outputs, you write code to represent a row of data.  
  
src/main/java/hello/Person.java  
  
package hello;  
  
public class Person {  
  
 private String lastName;  
 private String firstName;  
  
 public Person() {  
 }  
  
 public Person(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 @Override  
 public String toString() {  
 return "firstName: " + firstName + ", lastName: " + lastName;  
 }  
  
}  
You can instantiate the Person class either with first and last name through a constructor, or by setting the properties.  
  
Create an intermediate processor  
A common paradigm in batch processing is to ingest data, transform it, and then pipe it out somewhere else. Here you write a simple transformer that converts the names to uppercase.  
  
src/main/java/hello/PersonItemProcessor.java  
  
package hello;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import org.springframework.batch.item.ItemProcessor;  
  
public class PersonItemProcessor mplements ItemProcessor<Person, Person> {  
  
 private static final Logger log = LoggerFactory.getLogger(PersonItemProcessor.class);  
  
 @Override  
 public Person process(final Person person) throws Exception {  
 final String firstName = person.getFirstName().toUpperCase();  
 final String lastName = person.getLastName().toUpperCase();  
  
 final Person transformedPerson = new Person(firstName, lastName);  
  
 log.info("Converting (" + person + ") into (" + transformedPerson + ")");  
  
 return transformedPerson;  
 }  
  
}  
PersonItemProcessor implements Spring Batch’s ItemProcessor interface. This makes it easy to wire the code into a batch job that you define further down in this guide. According to the interface, you receive an incoming Person object, after which you transform it to an upper-cased Person.  
  
There is no requirement that the input and output types be the same. In fact, after one source of data is read, sometimes the application’s data flow needs a different data type.  
Put together a batch job  
Now you put together the actual batch job. Spring Batch provides many utility classes that reduce the need to write custom code. Instead, you can focus on the business logic.  
  
src/main/java/hello/BatchConfiguration.java  
  
package hello;  
  
import javax.sql.DataSource;  
  
import org.springframework.batch.core.Job;  
import org.springframework.batch.core.JobExecutionListener;  
import org.springframework.batch.core.Step;  
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;  
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;  
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;  
import org.springframework.batch.core.launch.support.RunIdIncrementer;  
import org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider;  
import org.springframework.batch.item.database.JdbcBatchItemWriter;  
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;  
import org.springframework.batch.item.file.FlatFileItemReader;  
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;  
import org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper;  
import org.springframework.batch.item.file.mapping.DefaultLineMapper;  
import org.springframework.batch.item.file.transform.DelimitedLineTokenizer;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.core.io.ClassPathResource;  
import org.springframework.jdbc.core.JdbcTemplate;  
  
@Configuration  
@EnableBatchProcessing  
public class BatchConfiguration {  
  
 @Autowired  
 public JobBuilderFactory jobBuilderFactory;  
  
 @Autowired  
 public StepBuilderFactory stepBuilderFactory;  
  
 // tag::readerwriterprocessor[]  
 @Bean  
 public FlatFileItemReader<Person> reader() {  
 return new FlatFileItemReaderBuilder<Person>()  
 .name("personItemReader")  
 .resource(new ClassPathResource("sample-data.csv"))  
 .delimited()  
 .names(new String[]{"firstName", "lastName"})  
 .fieldSetMapper(new BeanWrapperFieldSetMapper<Person>() {{  
 setTargetType(Person.class);  
 }})  
 .build();  
 }  
  
 @Bean  
 public PersonItemProcessor processor() {  
 return new PersonItemProcessor();  
 }  
  
 @Bean  
 public JdbcBatchItemWriter<Person> writer(DataSource dataSource) {  
 return new JdbcBatchItemWriterBuilder<Person>()  
 .itemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider<>())  
 .sql("INSERT INTO people (first\_name, last\_name) VALUES (:firstName, :lastName)")  
 .dataSource(dataSource)  
 .build();  
 }  
 // end::readerwriterprocessor[]  
  
 // tag::jobstep[]  
 @Bean  
 public Job importUserJob(JobCompletionNotificationListener listener, Step step1) {  
 return jobBuilderFactory.get("importUserJob")  
 .incrementer(new RunIdIncrementer())  
 .listener(listener)  
 .flow(step1)  
 .end()  
 .build();  
 }  
  
 @Bean  
 public Step step1(JdbcBatchItemWriter<Person> writer) {  
 return stepBuilderFactory.get("step1")  
 .<Person, Person> chunk(10)  
 .reader(reader())  
 .processor(processor())  
 .writer(writer)  
 .build();  
 }  
 // end::jobstep[]  
}  
For starters, the @EnableBatchProcessing annotation adds many critical beans that support jobs and saves you a lot of leg work. This example uses a memory-based database (provided by @EnableBatchProcessing), meaning that when it’s done, the data is gone.  
  
Break it down:  
  
src/main/java/hello/BatchConfiguration.java  
  
 @Bean  
 public FlatFileItemReader<Person> reader() {  
 return new FlatFileItemReaderBuilder<Person>()  
 .name("personItemReader")  
 .resource(new ClassPathResource("sample-data.csv"))  
 .delimited()  
 .names(new String[]{"firstName", "lastName"})  
 .fieldSetMapper(new BeanWrapperFieldSetMapper<Person>() {{  
 setTargetType(Person.class);  
 }})  
 .build();  
 }  
  
 @Bean  
 public PersonItemProcessor processor() {  
 return new PersonItemProcessor();  
 }  
  
 @Bean  
 public JdbcBatchItemWriter<Person> writer(DataSource dataSource) {  
 return new JdbcBatchItemWriterBuilder<Person>()  
 .itemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider<>())  
 .sql("INSERT INTO people (first\_name, last\_name) VALUES (:firstName, :lastName)")  
 .dataSource(dataSource)  
 .build();  
 }  
. The first chunk of code defines the input, processor, and output. - reader() creates an ItemReader. It looks for a file called sample-data.csv and parses each line item with enough information to turn it into a Person. - processor() creates an instance of our PersonItemProcessor you defined earlier, meant to uppercase the data. - write(DataSource) creates an ItemWriter. This one is aimed at a JDBC destination and automatically gets a copy of the dataSource created by @EnableBatchProcessing. It includes the SQL statement needed to insert a single Person driven by Java bean properties.  
  
The next chunk focuses on the actual job configuration.  
  
src/main/java/hello/BatchConfiguration.java  
  
 @Bean  
 public Job importUserJob(JobCompletionNotificationListener listener, Step step1) {  
 return jobBuilderFactory.get("importUserJob")  
 .incrementer(new RunIdIncrementer())  
 .listener(listener)  
 .flow(step1)  
 .end()  
 .build();  
 }  
  
 @Bean  
 public Step step1(JdbcBatchItemWriter<Person> writer) {  
 return stepBuilderFactory.get("step1")  
 .<Person, Person> chunk(10)  
 .reader(reader())  
 .processor(processor())  
 .writer(writer)  
 .build();  
 }  
. The first method defines the job and the second one defines a single step. Jobs are built from steps, where each step can involve a reader, a processor, and a writer.  
  
In this job definition, you need an incrementer because jobs use a database to maintain execution state. You then list each step, of which this job has only one step. The job ends, and the Java API produces a perfectly configured job.  
  
In the step definition, you define how much data to write at a time. In this case, it writes up to ten records at a time. Next, you configure the reader, processor, and writer using the injected bits from earlier.  
  
chunk() is prefixed <Person,Person> because it’s a generic method. This represents the input and output types of each "chunk" of processing, and lines up with ItemReader<Person> and ItemWriter<Person>.  
src/main/java/hello/JobCompletionNotificationListener.java  
  
package hello;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.batch.core.BatchStatus;  
import org.springframework.batch.core.JobExecution;  
import org.springframework.batch.core.listener.JobExecutionListenerSupport;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.stereotype.Component;  
  
@Component  
public class JobCompletionNotificationListener extends JobExecutionListenerSupport {  
  
 private static final Logger log = LoggerFactory.getLogger(JobCompletionNotificationListener.class);  
  
 private final JdbcTemplate jdbcTemplate;  
  
 @Autowired  
 public JobCompletionNotificationListener(JdbcTemplate jdbcTemplate) {  
 this.jdbcTemplate = jdbcTemplate;  
 }  
  
 @Override  
 public void afterJob(JobExecution jobExecution) {  
 if(jobExecution.getStatus() == BatchStatus.COMPLETED) {  
 log.info("!!! JOB FINISHED! Time to verify the results");  
  
 jdbcTemplate.query("SELECT first\_name, last\_name FROM people",  
 (rs, row) -> new Person(  
 rs.getString(1),  
 rs.getString(2))  
 ).forEach(person -> log.info("Found <" + person + "> in the database."));  
 }  
 }  
}  
This code listens for when a job is BatchStatus.COMPLETED, and then uses JdbcTemplate to inspect the results.  
  
Make the application executable  
Although batch processing can be embedded in web apps and WAR files, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java main() method.  
  
src/main/java/hello/Application.java  
  
package hello;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class Application {  
  
 public static void main(String[] args) throws Exception {  
 SpringApplication.run(Application.class, args);  
 }  
}  
@SpringBootApplication is a convenience annotation that adds all of the following:  
  
@Configuration tags the class as a source of bean definitions for the application context.  
  
@EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.  
  
Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.  
  
@ComponentScan tells Spring to look for other components, configurations, and services in the hello package, allowing it to find the controllers.  
  
The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there wasn’t a single line of XML? No web.xml file either. This web application is 100% pure Java and you didn’t have to deal with configuring any plumbing or infrastructure.  
  
For demonstration purposes, there is code to create a JdbcTemplate, query the database, and print out the names of people the batch job inserts.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

--->Overloading Overriding of static methods

->can overload all static methods

-> can overload just on the basis of static method(coz

it is not included in signature)

->overriding static method base and in child class

will result in method hiding not overriding as base classmethos will be called

even if object of child is created with ref to base class

Can we Overload or Override static methods in java ?

Let us first define Overloading and Overriding.

Overriding : Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class).

The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same. Refer Overriding in Java for details.

Overloading: Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, methods cannot be overloaded according to return type.

Can we overload static methods?

The answer is ‘Yes’. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program.

// filename Test.java

public class Test {

public static void foo() {

System.out.println("Test.foo() called ");

}

public static void foo(int a) {

System.out.println("Test.foo(int) called ");

}

public static void main(String args[])

{

Test.foo();

Test.foo(10);

}

}

Run on IDE

Output:

Test.foo() called

Test.foo(int) called

Can we overload methods that differ only by static keyword?

We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of this).

// filename Test.java

public class Test {

public static void foo() {

System.out.println("Test.foo() called ");

}

public void foo() { // Compiler Error: cannot redefine foo()

System.out.println("Test.foo(int) called ");

}

public static void main(String args[]) {

Test.foo();

}

}

Run on IDE

Output: Compiler Error, cannot redefine foo()

Can we Override static methods in java?

We can declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.

If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

/\* Java program to show that if static method is redefined by

a derived class, then it is not overriding. \*/

// Superclass

class Base {

// Static method in base class which will be hidden in subclass

public static void display() {

System.out.println("Static or class method from Base");

}

// Non-static method which will be overridden in derived class

public void print() {

System.out.println("Non-static or Instance method from Base");

}

}

// Subclass

class Derived extends Base {

// This method hides display() in Base

public static void display() {

System.out.println("Static or class method from Derived");

}

// This method overrides print() in Base

public void print() {

System.out.println("Non-static or Instance method from Derived");

}

}

// Driver class

public class Test {

public static void main(String args[ ]) {

Base obj1 = new Derived();

// As per overriding rules this should call to class Derive's static

// overridden method. Since static method can not be overridden, it

// calls Base's display()

obj1.display();

// Here overriding works and Derive's print() is called

obj1.print();

}

}

Run on IDE

Output:

Static or class method from Base

Non-static or Instance method from Derived

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

->Singleton class

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

The singleton class must provide a global access point to get the instance of the class.

Singleton pattern is used for logging, drivers objects, caching and thread pool. Be Hb

Eager Initialization

->the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

public class EagerInitialized{

private EagerInitialized(){}

private static final EagerInitialized instance= new EagerInitialized();

public static EagerInitializedSingleton getInstance(){

return instance;

}

}

---->Eager static initialization

package com.journaldev.singleton;

public class StaticBlockSingleton {

private static StaticBlockSingleton instance;

private StaticBlockSingleton(){}

//static block initialization for exception handling

static{

try{

instance = new StaticBlockSingleton();

}catch(Exception e){

throw new RuntimeException("Exception occured in creating singleton instance");

}

}

public static StaticBlockSingleton getInstance(){

return instance;

}

}

\*\*\* Lazy Initialization

public class LazyInitializedSingleton {

private static LazyInitializedSingleton instance;

private LazyInitializedSingleton(){}

public static LazyInitializedSingleton getInstance(){

if(instance == null){

instance = new LazyInitializedSingleton();

}

return instance;

}

}

\*\*\*\*\*\*\*\*\*Thread Safe Singleton

package com.journaldev.singleton;

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton(){}

public static synchronized ThreadSafeSingleton getInstance(){

if(instance == null){

instance = new ThreadSafeSingleton();

}

return instance;

}

}

\*\*\*\* Synchronized Block

public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){

if(instance == null){

synchronized (ThreadSafeSingleton.class) {

if(instance == null){

instance = new ThreadSafeSingleton();

}

}

}

return instance;

}

Achieve singleton

1)Private constructor to restrict instantiation of the class from other classes.

2)Private static variable of the same class that is the only instance of the class.

3)Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Immuatble class///////////

:class is immutable if its state cannot change, state of a class is determined by the value

its member variable are holding.

benenfits:

immutable class are thread safe,

immutable class can be used as key in collections that use hashing, like hashmap and hashtable

can be used in set

do not need implementation of clone

howToMake:

make a class as final so that no other class can extend and override its methods

no setter only getter

make its member variable as private and final

if class has reference to mutable object send a copy of it.

class Address{

Integer pinCode;

String street;

public Integer getPinCode(){

return pinCode;}

public void setPinCode(Integer pinCode){

pinCode=pinCode; }

public String getStreet(){

return street;}

public void setStreet(String street)

{street=street;}

}

public final class Employee{

private final Address address;

private final String name;

private final Integer age;

private Employee(String name,INteger age,Address address){

this.address.setPinCode(address.getPinCode());

this.address.setStreet(address.getStreet());

this.name=name;

this.age=age;

}

@Override

public String toString() {

return immutableField1 +" - "+ immutableField2 +" - "+ mutableField;

}

public static createInstance(String name,Iteger age,Address address){

return new Employee(name,age,address);

}

public Address getAddress(){

Address address1= new Address()

addrerss1.setPin(address.getPin());

address1.setStreet(address.getStreet());

return address1;

}

public String getName(){

return name;

}

public Integer getAge(){

return age;

}

}

class tetsImmutable{

public sttaic voidd main(String a[]args){

Address add= new Address();

add.pincode=2233;

add.setStreet("lane1 JPnagar");

Employee emp=createInstance("rish",24,add);

System.out.println(emp.toString());

Employee emp2-createInstance("prat",27,add);

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

java .util .data represent data time of day

java .sql .date represent date

The java.sql.Date class is used with JDBC and it was

intended to not have a time part, that is,

hours, minutes, seconds, and milliseconds should be zero…

but this is not enforced by the class.

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Marlker interface

design pattern used in Obj Oriented lang that provide runtime info about object,

it provide mean to associate metadata to class where language does not have explicit support for such metadata.

A good example of use of marker interface in java is Serializable interface. A class implements this interface to

indicate that its non-transient data members can be written to a byte steam or file system.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

-> creating string as new() and literal

String created using new is placed in heap memory everytime new() is used , unlike strings have same or different value

String created using literal are placed in string pool, a memory specifired in heap

area where same valued string literal point to each other , hence when same value is provide

to get reference to already existing value is returned without creating new

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Keyword ‘intern’ usage

This is best described by java docs:

When the intern() method is invoked, if the pool already contains a

string equal to this String object as determined by the equals(Object)

method, then the string from the pool is returned. Otherwise, this String

object is added to the pool and a reference to this String object is returned.

String str = new String("abc");

str.intern();

It follows that for any two strings s and t, s.intern() == t.intern()

is true if and only if s.equals(t) is true. Means if s and t both are

different string objects and have same character sequence, then calling

intern() on both will result in single string pool literal referred by

both variables.

Java automatically interns String literals. This means that in many cases, the == operator appears to work for Strings in the same way that it does for ints or other primitive values.

Since interning is automatic for String literals, the intern() method is to be used on Strings constructed with new String()

Using your example:

String s1 = "Rakesh";

String s2 = "Rakesh";

String s3 = "Rakesh".intern();

String s4 = new String("Rakesh");

String s5 = new String("Rakesh").intern();

if ( s1 == s2 ){

System.out.println("s1 and s2 are same"); // 1.

}

if ( s1 == s3 ){

System.out.println("s1 and s3 are same" ); // 2.

}

if ( s1 == s4 ){

System.out.println("s1 and s4 are same" ); // 3.

}

if ( s1 == s5 ){

System.out.println("s1 and s5 are same" ); // 4.

}

will return:

s1 and s2 are same

s1 and s3 are same

s1 and s5 are same

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

equals vs ==

== operator compare for object references i.e. memory address

equality. So if two string objects are referring to same literal in

string pool

or same string object in heap then s ==t will return true, else false.

equals() method is overridden in String class and it verify the char

sequences hold by string objects. If they store the same char sequence

, the s.equals(t) will return true, else false.

In general both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two:

Main difference between .equals() method and == operator is that one is method and other is operator.

We can use == operators for reference comparison (address comparison) and .equals() method for content comparison. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

If a class does not override the equals method, then by default it uses equals(Object o) method of the closest parent class that has overridden this method. See this for detail

Coding Example:

// Java program to understand

// the concept of == operator

public class Test {

public static void main(String[] args)

{

String s1 = new String("HELLO");

String s2 = new String("HELLO");

System.out.println(s1 == s2);

System.out.println(s1.equals(s2));

}

}

Run on IDE

Output:

false

true

Explanation: Here we are creating two objects namely s1 and s2.

Both s1 and s2 refers to different objects.

When we use == operator for s1 and s2 comparison then the result is false as both have different addresses in memory.

Using equals, the result is true because its only comparing the values given in s1 and s2.

Let us understand both the operators in detail:

Equality operator(==)

We can apply equality operators for every primitive types including boolean type. we can also apply equality operators for object types.

// Java program to illustrate

// == operator for compatible data

// types

class Test {

public static void main(String[] args)

{

// integer-type

System.out.println(10 == 20);

// char-type

System.out.println('a' == 'b');

// char and double type

System.out.println('a' == 97.0);

// boolean type

System.out.println(true == true);

}

}

Run on IDE

Output:

false

false

true

true

If we apply == for object types then, there should be compatibility between arguments types (either child to parent or parent to child or same type). Otherwise we will get compile time error.

// Java program to illustrate

// == operator for incompatible data types

class Test {

public static void main(String[] args)

{

Thread t = new Thread();

Object o = new Object();

String s = new String("GEEKS");

System.out.println(t == o);

System.out.println(o == s);

// Uncomment to see error

System.out.println(t==s);

}

}

Run on IDE

Output:

false

false

// error: incomparable types: Thread and String

.equals()

In Java, string equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are same then it returns true. If all characters are not matched then it returns false.

public class Test {

public static void main(String[] args)

{

Thread t1 = new Thread();

Thread t2 = new Thread();

Thread t3 = t1;

String s1 = new String("GEEKS");

String s2 = new String("GEEKS");

System.out.println(t1 == t3);

System.out.println(t1 == t2);

System.out.println(t1.equals(t2));

System.out.println(s1.equals(s2));

}

}

Run on IDE

Output:

true

false

false

true

Explanation: Here we are using .equals method to check whether two objects contains the same data or not.

In the above example, we are creating 3 Thread objects and 2 String objects.

In the first comparison, we are checking that t1 == t3 or not. As we know that both t1 and t3 pointing to same object that’s why it returns true.

When we are comparing 2 String objects by .equals() operator then we are checking that is both objects contains the same data or not.

Both the objects contains the same String i.e. GEEKS that’s why it returns true.

This article is contributed by Bishal Kumar Dubey. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

also if string has already defined equals in its class so may not need to override explicitly,

but when reading or writng from file u may need to override,

and for custom class when setting same value it may result in false,

as need to override equals, and also hashcode...

example:

Account class have same value but result in false as object class equals got called on it

public boolean equals(Object obj) {

return (this == obj);

}

// Java program to understand

// the concept of == operator

public class Test {

public static void main(String[] args)

{

============================================================

String s1 = new String();

String s2 = new String();

System.out.println(s1 == s2);

System.out.println(s1.equals(s2));

false

true

System.out.println("====================");

String s3="hello";

String s4="hello";

System.out.println(s3.equals(s4));

System.out.println(s3==s4);

true

true

System.out.println("====================");

Account acc= new Account();

acc.setType("1");

Account acc1= new Account();

acc1.setType("2");

System.out.println(acc.equals(acc1));

false

System.out.println("====================");

}

}

class Account{

private String type;

public void setType(String type){

this.type=type;

}

}

//////////////////////////////////////////////

--------------->equal() by default checks memory address

``````

Object o1 = new Object();

Object o2 = new Object();

//o1=o2;

System.out.println(o1.equals(o2));

It returns false. It can return true, if the comment is removed.

Why isn't the same thing applicable to the String class?

String s1=new String();

String s2=new String();

System.out.println(s1.equals(s2));

Object equals by default is

/\* Object.equals() \*/

public boolean equals(Object obj) {

return (this == obj);

}

String equals is overriden is

/\* String.equals() \*/

public boolean equals(Object anObject) {

if (this == anObject) {

return true;

}

if (anObject instanceof String) {

String anotherString = (String)anObject;

int n = count;

if (n == anotherString.count) {

char v1[] = value;

char v2[] = anotherString.value;

int i = offset;

int j = anotherString.offset;

while (n-- != 0) {

if (v1[i++] != v2[j++])

return false;

}

return true;

}

}

return false;

}

als0------------>

Object a = new Integer(2);

Object b = new Integer(2);

System.out.println(a.equals(b));

Object x = new Object();

Object y = new Object();

System.out.println(x.equals(y));

The first print statement prints true and the second false.

/////////////////////////////////////////////////////////////////////////////

How does substring () inside String works?

String in java are like any other programming language, a sequence of characters. This is more like a utility class to work on that char sequence. This char sequence is maintained in following variable:

/\*\* The value is used for character storage. \*/

private final char value[];

To access this array in different scenarios, following variables are used:

/\*\* The offset is the first index of the storage that is used. \*/

private final int offset;

/\*\* The count is the number of characters in the String. \*/

private final int count;

Whenever we create a substring from any existing string instance, substring() method only set’s the new values of offset and count variables. The internal char array is unchanged

Memory leak issue

Till now we gone through basic stuff. Now something serious. Have you tried creating substrings from a string object. I bet, Yes. Do you know the internals of substring in java. How they create memory leaks?

Sub strings in java are created using method substring(int beginIndex) and some other overloaded forms of this method. All these methods create a new String object and update the offset and count variable which we saw in start of this article.

The original value[] is unchaged. Thus if you create a string with 10000 chars and create 100 substrings with 5-10 chars in each, all 101 objects will have same char array of size 10000 chars. It is memory wastage without any doubt.

how to prevent it:

--------->A simple but very common example that can lead to a memory leak is to use a HashSet with objects that are missing their hashCode() or equals() implementations.

Specifically, when we start adding duplicate objects into a Set – this will only ever grow, instead of ignoring duplicates as it should. We also won’t be able to remove these objects, once added.

Let’s create a simple class without either equals or hashCode:

public class Key {

public String key;

public Key(String key) {

Key.key = key;

}

}

---------> Close stream when unused

--------->Do profiling using visual VM

--------->Review Code

///////////////////////////////////////////////////////////////

./------>Working of HashMap, and duplicate collision resolved

--- “How HashMap works?”,“On principles of Hashing“

Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/ algorithm on its properties.

A map by definition is : “An object that maps keys to values”. Very easy.. right? So, HashMap has an inner class Entry, which looks like this:

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

{

final K key;

V value;

Entry<k ,V> next;

final int hash;

...//More code goes here

}

\*\*

if I am hashing integers and my hashing function is simply (n % 10) then the number 17 and the number 27 will produce the same result. This does not mean that those numbers are the same.

First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.

Then on next step, a hash value is calculated using key’s hash code by calling its hashCode() method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written hashCode() functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function, and passed the object’s hash code to this hash() function to bring hash value in range of array index size.

Now indexFor(hash, table.length) function is called to calculate exact index position for storing the Entry object.

Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [called bucket]. Answer is LinkedList. If you remember, Entry class had an attribute “next”. This attribute always points to next object in chain. This is exactly the behavior of LinkedList.

So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.

If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.

What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals() method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.

````````HOW PUT WORKS```````````

->IN HASHMAP ARRAY OF eNTRy class is used as--> transient Entry[] trable;

.>) First of all, key object is checked for null. If key is null,

value is stored in table[0] position. Because hash code for null is always 0.

-> Then on next step, a hash value is calculated using key’s hash code

by calling its hashCode() method. This hash value is used to calculate

index in array for storing Entry object. JDK designers well

assumed that there might be some poorly written hashCode() functions

that can return very high or low hash code value. To solve this issue,

they introduced another hash() function, and passed the object’s hash code to this hash()

function to bring hash value in range of array index size.

-> Now indexFor(hash, table.length) function is called to

calculate exact index position for storing the Entry object.

-> So if two unequal obj can also have same hahscode,so both have to store in same location

called bucket (LinkedlIst) as Entry classhad attribute next(pointing to next obj in chain)

If there is already an object sitting on calculated index, its

next attribute is checked. If it is null, and current Entry object

becomes next node in LinkedList. If next variable is

not null, procedure is followed until next is evaluated as null.

->

What if we add the another value object with same key as entered before.

Logically, it should replace the old value. How it is done? Well, after

determining the index position of Entry object, while iterating over LinkedList

on calculated index, HashMap calls equals method on key object for each Entry

object. All these Entry objects in LinkedList will have similar hash code but

equals() method will test for true equality. If key.equals(k) will be true

then both keys are treated as same key object. This will cause the replacing

of value object inside Entry object only.

`````````HOW GET WORKS``````````````

--->

Now we have got the idea, how key-value pairs are stored in HashMap.

Next big question is : what happens when an object is passed in get method of

HashMap? How the value object is determined?

Answer we already should know that the way key uniqueness is determined

in put() method , same logic is applied in get() method also. The moment

HashMap identify exact match for the key object passed as argument, it simply

returns the value object stored in current Entry object.

If no match is found, get() method returns null.

//////////////////////

-------------->Why Override hashcode () and equals ()

--->shCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

hashCode() method is used to get a unique integer for given object. This integer is used for determining the bucket location, when this object needs to be stored in some HashTable like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.

equals() method, as name suggest, is used to simply verify the equality of two objects. Default implementation simply check the object references of two objects to verify their equality.

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode() method, which states that equal objects must have equal hash codes.

equals() must define an equality relation (it must be reflexive, symmetric and transitive). In addition, it must be consistent (if the objects are not modified, then it must keep returning the same value). Furthermore, o.equals(null) must always return false.

hashCode() must also be consistent (if the object is not modified in terms of equals(), it must keep returning the same value).

The relation between the two methods is:

Whenever a.equals(b) then a.hashCode() must be same as b.hashCode().

////////////////////////////////////////////////////////

-------->Abstract vs Interface

--->Difference between interfaces and abstract classes?

This is very common question if you are appearing interview for junior level programmer. Well, most noticeable differences are as below:

Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.

Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.

Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, abstract.

Java interface should be implemented using keyword “implements“; A Java abstract class should be extended using keyword “extends“.

A Java class can implement multiple interfaces but it can extend only one abstract class.

Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists. Since Java 8, you can define default methods in interfaces.

Abstract class are slightly faster than interface because interface involves a search before calling any overridden method in Java. This is not a significant difference in most of cases but if you are writing a time critical application than you may not want to leave any stone unturned.

//////////////////////////////////////////////////////

----------------->how to create instance without new()

1) Using newInstance method of Class class

Class ref = Class.forName("DemoClass");

DemoClass obj = (DemoClass) ref.newInstance();

2) Using class loader’s loadClass()

Just like above mechanism, class loader’s loadClass() method does the same thing.

instance.getClass().getClassLoader().loadClass("NewClass").newInstance();

3) Using clone()

NewClass obj = new NewClass;

NewClass obj2= (NewClass)obj.clone();

4) Object serialiation/deserialization

ObjectInputStream objStream = new ObjectInputStream(inputStream);

NewClass obj = (NewClass ) inStream.readObject();

5)uSING REFLECTION

constructor.newInstance(); or

class.newInstance();

//////////////////////////////////////////////////////

----------------->jAVA WRAPPER CLASS INTERNAL CACHING

for everytime an object is made/new instance is create memory in heap is taken

So, it can be understood that having objects ready made has its own

benefit and should be promoted as well.

So, like string pool, they can also have their pool, right? “Great Idea”.

Well, it’s already there. JDK provided wrapper classes also provide this in

form of instance pooling i.e. each wrapper class store a list of commonly used

instances of own type in form of cache and whenever required, you can use

them in your code.

It helps in saving lots of byes in your program runtime.

`````````Integer``````````

Declaring Integer like below

Integer i1 = 127;

Results in to Integer i1 = Integer.valueOf(127);

So what actually happening for first case is

Integer i1 = 127;<---Integer.valueOf(127);

Integer i2 = 127;<---Integer.valueOf(127);<---Same reference as first

From source code of Integer for class valueOf method

public static Integer valueOf(int i) {

if(i >= -128 && i <= IntegerCache.high)

return IntegerCache.cache[i + 128];

else

return new Integer(i);

}

So you get same reference if value is between -128 to 127 and

you call valueOf else it just returns new Integer(i)

And because reference is same your == operator works

for integers returned by valueOf between this range.

On the other hand, if you assign a value outside this range

to a wrapper reference type, Integer.valueOf will create a new Integer

object for that value. And hence, comparing the reference for

Integer objects having value outside this range will give you false

Integer i = 127; --> // Equivalent to `Integer.valueOf(127)`

Integer i2 = 127;

// Equivalent to `Integer.valueOf(128)`

// returns `new Integer(128)` for value outside the `Range - [-128, 127]`

Integer i3 = 128;

Integer i4 = 128;

System.out.println(i == i2); // true, reference pointing to same literal

System.out.println(i3 == i4); // false, reference pointing to different objects

But , when you create your integer instances using new operator, a

new object will be created on Heap. So,

Integer i = new Integer(127);

Integer i2 = new Integer(127);

System.out.println(i == i2); // false

----iternal Integer Structire us

Lets see how this IntegerCache look like in code:

private static class IntegerCache

{

private IntegerCache(){}

static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static {

for(int i = 0; i < cache.length; i++)

cache[i] = new Integer(i - 128);

}

}

And this is how valueOf() method looks like:

public static Integer valueOf(int i)

{

final int offset = 128;

if (i >= -128 && i <= 127) // must cache

return IntegerCache.cache[i + offset];

}

return new Integer(i);

}

``````Other wrapper class ``````

Other wrapper classes

1)java.lang.Boolean store two inbuilt instances TRUE and FALSE,

and return their reference if new keyword is not used.

2)java.lang.Character has a cache for chars between unicodes 0 and 127 (ascii-7 / us-ascii).

3)java.lang.Long has a cache for long between -128 to +127.

4)java.lang.String has a whole new concept of string pool.

/////////////////////////////////////////////////////////////////////////////////

-->HOw to design god key in hashmap

``````we should be able to retrieve the value object back from the

map without failure

-> key hashcode along with equals method finds the location and availability to get keys value

->On runtime, JVM computes hash code for each object and provide it

on demand. When we modify an object’s state, JVM set a flag that object is

modified and hash code must be AGAIN computed. So, next time you call object’s

-> immutability is recommended and not mandatory.

If you want to make a mutable object as key in hashmap, then you

have to make sure that state change for key object does not change the hash

code of object. This can be done by overriding the hashCode() method.

But, you must make sure you are honoring the contract with equals() also.

hashCode() method, JVM recalculate the hash code for that object.

For this basic reasoning, key objects are suggested to be IMMUTABLE.

IMMUTABILITY allows you to get same hash code every time, for a key object.

So it actually solves most of the problems in one go. Also, this class must honor

the hashCode() and equals() methods contract.

exapmle--

package com.howtodoinjava.demo.map;

public class Account

{

private int accountNumber;

private String holderName;

public Account(int accountNumber) {

this.accountNumber = accountNumber;

}

public String getHolderName() {

return holderName;

}

public void setHolderName(String holderName) {

this.holderName = holderName;

}

public int getAccountNumber() {

return accountNumber;

}

//Depends only on account number

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + accountNumber;

return result;

}

//Compare only account numbers

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Account other = (Account) obj;

if (accountNumber != other.accountNumber)

return false;

return true;

}

}

1) Whenever a.equals(b) is true, then a.hashCode() must be same as b.hashCode().

2) Whenever a.equals(b) is false, then a.hashCode() may/may not be same as b.hashCode().

-->

//////////////////////////////////////////////////////////////////////////////////////

---->Collection Framework Advantage

\*Reduced programming effort due to ready to use code

\*Increased performance because of high-performance implementations of

data structures and algorithms

\*Provides interoperability between unrelated APIs by establishing a

common language to pass collections back and forth

\*Easy to learn APIs by learning only some top level

interfaces and supported operations

//////////////////////////////////////////////////////////////

----->wh collection does not extend cloneable and serializable

````not required , Collection interface is not expected to do

what Cloneable and Serializable interfaces do.

Another reason is that not everybody will have a reason

to have Cloneable collection because if it has very large data,

then every unnecessary clone operation will consume a big memory.

Beginners might use it without knowing the consequences.

//////////////////////////////////////////////////////////////////

--->why map doesnt extend CollectionInterface

because they are incompatible“. Collection has a method add(Object o).

Map can not have such method because it need key-value pair. There are other

reasons also such as Map

supports keySet, valueSet etc. Collection classes does not have such views.

////////////////////////////////////////////////////////////////

----> Why List interface, which class implement it

```It is ordered,

Stores in indexed form

/--> classes that implement it ae

-Stack,Vector,,ArrayList,LinkedlIst

////////////////////////////////////////////////////////////////

----> Convert array to arrayList

//String array

String[] words = {"ace", "boom", "crew", "dog", "eon"};

//Use Arrays utility class

List wordList = Arrays.asList(words);

//Now you can iterate over the list

it will return List of element of any type, of which the array is. e.g.

//String array

Integer[] nums = {1,2,3,4};

//Use Arrays utility class

List numsList = Arrays.asList(nums);

////////////////////////////////////////////////////////////////

----> Reverse a list

Collections.reverse(list);

------->reverse array

Collections.reverse(Arrays.asList(arr));

Arrays.toString (arr);

////////////////////////////////////////////////////////////////

------> Set interface

, it is not ordered collection. So no ordering

is preserved while adding or removing elements. The main feature it does provide

is “uniqueness of elements“. It does not support duplicate elements.

Set also adds a stronger contract on the behavior of the equals and hashCode operations,

allowing Set instances to be compared meaningfully even if their implementation types

differ. Two Set instances are equal if they contain the same elements.

Based on above reasons, it does not have operations based on indexes of elements

like List. It only has methods which are inherited by Collection interface.

Main classes implementing Set interface are : EnumSet, HashSet, LinkedHashSet,

TreeSet. Read more on related java documentation.

//////////////////////////////////////////////////////////////////////////////

-------> How HshSet works

````You must know that HashMap store key-value pairs, with

one condition i.e. keys will be unique. HashSet uses Map’s this feature

to ensure uniqueness of elements. In HashSet class, a map declaration is as below:

private transient HashMap<E,Object> map;

//This is added as value for each key

private static final Object PRESENT = new Object();

So when you store a element in HashSet, it stores the element as key

in map and “PRESENT” object as value. (See declaration above).

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

/////////////////////////////////////////////////////////////////////////////////////

------>Possible add nu;ll in hashset or treeset

As you see, There is no null check in add() method in previous

question. And HashMap also allows one null key, so one “null” is allowed in HashSet.

TreeSet uses the same concept as HashSet for internal logic, but uses

NavigableMap for storing the elements.

private transient NavigableMap<E,Object> m;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

NavigableMap is subtype of SortedMap which does not allow null keys.

So essentially, TreeSet also does not support null keys.

It will throw NullPointerException if you try to add null element in TreeSet.

/////////////////////////////////////////////////////////////////////////////////////

----> About hashMap

`Interface not extending collection,

stores in key value format

classes that extend are

: HashMap, Hashtable, EnumMap, IdentityHashMap, LinkedHashMap and Properties.

////////////////////////////////////////////////////////////////////////////////////////////////

--------->ConsurrentHashMAp and its working

////////////////////////////////////////////////////////////////////////////////////

------>hahsMAp vs concurrent hash map

1)HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.

HashMap performance is relatively high because it is non-synchronized in nature and any

number of threads can perform simultaneously. But ConcurrentHashMap performance is low

sometimes because sometimes Threads are

required to wait on ConcurrentHashMap.

While one thread is Iterating the HashMap object, if other thread try to add/modify

the contents of Object then we will get Run-time exception saying

ConcurrentModificationException.Whereas In ConcurrentHashMap we wont get any exception

while performing any modification at the time of Iteration.

```````Using HashMap

// Java program to illustrate

// HashMap drawbacks

import java.util.HashMap;

class HashMapDemo extends Thread

{

static HashMap<Integer,String> l=new HashMap<Integer,String>();

public void run()

{

// Child thread trying to add

// new element in the object

l.put(103,"D");

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

System.out.println("Child Thread going to add element");

}

}

public static void main(String[] args) throws InterruptedException

{

l.put(100,"A");

l.put(101,"B");

l.put(102,"C");

HashMapDemo t=new HashMapDemo();

t.start();

for (Object o : l.entrySet())

{

Object s=o;

System.out.println(s);

Thread.sleep(1000);

}

System.out.println(l);

}

}

Run on IDE

Output:

100=A

Exception in thread "main" java.util.ConcurrentModificationException

Using ConcurrentHashMap

// Java program to illustrate

// HashMap drawbacks

import java.util.HashMap;

import java.util.concurrent.\*;

class HashMapDemo extends Thread

{

static ConcurrentHashMap<Integer,String> l =

new ConcurrentHashMap<Integer,String>();

public void run()

{

// Child add new element in the object

l.put(103,"D");

try

{

Thread.sleep(2000);

}

catch(InterruptedException e)

{

System.out.println("Child Thread going to add element");

}

}

public static void main(String[] args) throws InterruptedException

{

l.put(100,"A");

l.put(101,"B");

l.put(102,"C");

HashMapDemo t=new HashMapDemo();

t.start();

for (Object o : l.entrySet())

{

Object s=o;

System.out.println(s);

Thread.sleep(1000);

}

System.out.println(l);

}

}

Run on IDE

Output:

100=A

101=B

102=C

103=D

{100=A, 101=B, 102=C, 103=D}

````n HashMap, null values are allowed for key and values, whereas

in ConcurrentHashMap null value is not allowed for key and value,

otherwise we will get Run-time exception saying NullPointerException.

```````Using HashMap

//Java Program to illustrate ConcurrentHashMap behaviour

import java.util.\*;

class ConcurrentHashMapDemo

{

public static void main(String[] args)

{

HashMap m=new HashMap();

m.put(100,"Hello");

m.put(101,"Geeks");

m.put(102,"Geeks");

m.put(null,"World");

System.out.println(m);

}

}

Run on IDE

output:

{null=World, 100=Hello, 101=Geeks, 102=Geeks}

Using ConcurrentHashMap

//Java Program to illustrate HashMap behaviour

import java.util.concurrent.\*;

class ConcurrentHashMapDemo

{

public static void main(String[] args)

{

ConcurrentHashMap m=new ConcurrentHashMap();

m.put(100,"Hello");

m.put(101,"Geeks");

m.put(102,"Geeks");

m.put(null,"World");

System.out.println(m);

}

}

Output:

Exception in thread "main" java.lang.NullPointerException

/////////////////////////////////////////////////////////////////////////////////////

/------------>What are different Collection views provided by Map interface?

Map interface provides 3 views of key-values pairs stored in it:

key set view

value set view

entry set view

////////////////////////////////////////////////////////////////////

------------------>When to use HashMap or TreeMap?

HashMap is well known class and all of us know that. So, I will

leave this part by saying that it is used to store key-value pairs and

allows to perform many operations on such collection of pairs.

TreeMap is special form of HashMap. It maintains the ordering of keys which

is missing in HashMap class. This ordering is by default “natural ordering”.

The default ordering can be override by providing an instance of Comparator class,

whose compare method will be used to maintain ordering of keys.

Please note that all keys inserted into the map must implement the Comparable

interface (this is necessary to decide the ordering). Furthermore, all such

keys must be mutually comparable: k1.compareTo(k2) must not throw a ClassCastException

for any keys k1 and k2 in the map. If the user attempts to put a key into the map that

violates this constraint (for example, the user attempts to put a string key into a map

whose keys are integers),

the put(Object key, Object value) call will throw a ClassCastException.

/////////////////////////////////////////////////////////////////////////////////////

Difference between HashMap and HashTable?

There are several differences between HashMap and Hashtable in Java:

Hashtable is synchronized, whereas HashMap is not.

Hashtable does not allow null keys or values. HashMap allows one null

key and any number of null values.

The third significant difference between HashMap vs Hashtable is that

Iterator in the HashMap is a

fail-fast iterator while the enumerator for the Hashtable is not.

////////////////////////////////////////////////////////////////////////////////

---------------->Fail fast vs Fail Safe iterator (Weakly Consistent)

```````Iterators from java.util package throw ConcurrentModificationException if

collection was modified by

collection's methods (add / remove) while iterating

Iterators from java.util.concurrent package typically iterate over

a snapshot and allow concurrent modifications but may not reflect collection

updates after the iterator was created.

if there is structural modification of the collection. Structural modification means adding,

removing or updating any element from collection while a thread is iterating over that

collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

Fail-Safe iterators don’t throw any exceptions if a collection is structurally

modified while iterating over it. This is because, they operate on the clone of the

collection, not on the original collection and that’s why they are called fail-safe iterators.

Iterator on CopyOnWriteArrayList,

ConcurrentHashMap classes are examples of fail-safe Iterator.

FAILFAST CODE

// Java code to illustrate

// Fail Fast Iterator in Java

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

public class FailFastExample {

public static void main(String[] args)

{

Map<String, String> cityCode = new HashMap<String, String>();

cityCode.put("Delhi", "India");

cityCode.put("Moscow", "Russia");

cityCode.put("New York", "USA");

Iterator iterator = cityCode.keySet().iterator();

while (iterator.hasNext()) {

System.out.println(cityCode.get(iterator.next()));

// adding an element to Map

// exception will be thrown on next call

// of next() method.

cityCode.put("Istanbul", "Turkey");

}

}

//using iteratot to remove will not throw exception

Iterator<Integer> itr = al.iterator();

while (itr.hasNext()) {

if (itr.next() == 2) {

// will not throw Exception

itr.remove();

}

}

System.out.println(al);

itr = al.iterator();

while (itr.hasNext()) {

if (itr.next() == 3) {

// will throw Exception on

// next call of next() method

al.remove(3);

}

}

}

Run on IDE

Output :

India

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.HashMap$HashIterator.nextNode(HashMap.java:1442)

at java.util.HashMap$KeyIterator.next(HashMap.java:1466)

at FailFastExample.main(FailFastExample.java:18)

---------------->FAILSAFE

Fail-safe iterators allow modifications of a collection while iterating over it.

These iterators don’t throw any Exception if a collection is modified while iterating over it.

They use copy of original collection to traverse over the elements of the collection.

These iterators require extra memory for cloning of collection. Ex : ConcurrentHashMap, CopyOnWriteArrayList

FAIL safe may or may not create Copy of Collection, but are capable of hadling mofdification

COPY CREATED IN COPYONWRITEARRAYLIST<>(new INteger{1.2.3.4});

Example of Fail Safe Iterator in Java:

// Java code to illustrate

// Fail Safe Iterator in Java

import java.util.concurrent.CopyOnWriteArrayList;

import java.util.Iterator;

class FailSafe {

public static void main(String args[])

{

CopyOnWriteArrayList<Integer> list

= new CopyOnWriteArrayList<Integer>(new Integer[] { 1, 3, 5, 8 });

Iterator itr = list.iterator();

while (itr.hasNext()) {

Integer no = (Integer)itr.next();

System.out.println(no);

if (no == 8)

// This will not print,

// hence it has created separate copy

list.add(14);

}

}

}

------------->CONCURRENTHASHMAP DOESNT CREATE COPY

Example of Fail-Safe Iterator which does not create separate copy

// Java program to illustrate

// Fail-Safe Iterator which

// does not create separate copy

import java.util.concurrent.ConcurrentHashMap;

import java.util.Iterator;

public class FailSafeItr {

public static void main(String[] args)

{

// Creating a ConcurrentHashMap

ConcurrentHashMap<String, Integer> map

= new ConcurrentHashMap<String, Integer>();

map.put("ONE", 1);

map.put("TWO", 2);

map.put("THREE", 3);

map.put("FOUR", 4);

// Getting an Iterator from map

Iterator it = map.keySet().iterator();

while (it.hasNext()) {

String key = (String)it.next();

System.out.println(key + " : " + map.get(key));

// This will reflect in iterator.

// Hence, it has not created separate copy

map.put("SEVEN", 7);

}

}

}

/////////////////////////////////////////////////////////////////////////////

------------------->Why Iterator .remov dont throw Exception while CollectionObj.remove does

1```````

Consider this example:

List<String> list = new ArrayList<>(Arrays.asList("a", "b", "c", "d"));

for (Iterator<String> iter = list.iterator(); iter.hasNext(); ) {

if (iter.next().equals("b")) {

// iter.remove(); // #1

// list.remove("b"); // #2

}

}

If you uncomment line #1, it will work fine. If you uncomment line #2

(but leave #1 commented) then it will cause the subsequent call to iter.next() to

throw ConcurrentModificationException.

The reason is that the iterator is a separate object that has some

references to the internal state of the underlying list. If you modify the

list while the iterator is in operation, it could cause the iterator to behave badly,

e.g. by skipping elements, repeating elements, indexing off the end of the array, etc.

It attempts to detect such modifications and so it throws ConcurrentModificationException

if it does.

Removing elements through the iterator works and does not cause exceptions, because this

updates the underlying list and the iterator's state that refers to the internals of the,

so everything can stay consistent.

However, there is nothing special about iterator.remove() that makes it work in all cases.

If there are multiple iterators iterating over the same list, modifications made by one will

cause problems for the others. Consider:

Iterator<String> i1 = list.iterator();

Iterator<String> i2 = list.iterator();

i1.remove();

i2.remove();

We now have two iterators pointing into the same list. If we modify the list using one

of them, it disrupts the operation of the second, so the call to i2.remove() will result

in ConcurrentModificationException.

///////

///////////////////////////////////////////////////////////////////////////

------------->OCCURance of number of word in a list

`````import java.util.\*;

public class HelloWorld{

public static void main(String []args){

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

String s= "this is list of words that will define occurance of ea";

String []arr= s.split(" ");

ls = Arrays.asList(arr);

for(String ss: ls){

int freq = Collections.frequency(ls, ss);

System.out.println(ss+" occured "+ freq+" times");

}

// System.out.println("Hello World");

}

}

///////////////////////////////////////////////////////////////////////////

-------->Difference between HashMap and HashTable?

There are several differences between HashMap and Hashtable in Java:

1)Hashtable is synchronized, whereas HashMap is not.

2)Hashtable does not allow null keys or values. HashMap allows one null key

and any number of null values.

3)The third significant difference between HashMap vs Hashtable is that

Iterator in the HashMap

is a fail-fast iterator while the enumerator for the Hashtable is not.

///////////////////////////////////////////////////////////////////////////

--------------->Difference between Vector and ArrayList?

Lets note down the differences:

All the methods of Vector is synchronized. But, the methods of ArrayList is not synchronized.

Vector is a Legacy class added in first release of JDK.

ArrayList was part of JDK 1.2, when collection framework was introduced in java.

By default, Vector doubles the size of its array when it is re-sized internally.

But, ArrayList increases by half of its size when it is re-sized.

///////////////////////////////////////////////////////////////////////////

---------> Itrator vs Enumerator

```````

Iterators differ from enumerations in three ways:

1)Iterators allow the caller to remove elements from the underlying collection during the iteration with its remove() method.

You can not add/remove elements from a collection when using enumerator.

2)Enumeration is available in legacy classes i.e Vector/Stack etc.

whereas Iterator is available in all modern collection classes.

3)Another minor difference is that Iterator has improved method names e.g.

Enumeration.hasMoreElement() has become Iterator.hasNext(), Enumeration.nextElement() has become Iterator.next() etc.

Enumerator example

package com.myjava.Enumeration;

import java.util.Enumeration;

import java.util.Vector;

public class MyEnumeration {

public static void main(String a[]){

Vector<String> lang = new Vector<String>();

Enumeration<String> en = null;

lang.add("JAVA");

lang.add("JSP");

lang.add("SERVLET");

lang.add("EJB");

lang.add("PHP");

lang.add("PERL");

en = lang.elements();

while(en.hasMoreElements()){

System.out.println(en.nextElement());

}

}

}

///////////////////////////////////////////////////////////////////////////

----------------------->Iterator vs ListIterator

`````````What is the difference between Iterator and ListIterator ?

Iterator:-

An iterator over a collection.

By using Iterator we can retrieve the elements from Collection Object in forward direction only.

We can use Iterator to traverse Set and List and also Map type of Objects.

Iterator iterator = Set.iterator();

Iterator iterator = List.iterator();

Methods in Iterator :

hashNext()

next()

remove()

ListIterator:-

An iterator for lists that allows the programmer to traverse the list in

either direction, modify the list during iteration, and obtain the iterator's current

position in the list.

A ListIterator has no current element; its cursor position always lies between

the element that would be returned by a call to previous() and the element that would be

returned by a call to next().

But List Iterator can be used to traverse for List type Objects, but not for Set

type of Objects.

ListIterator listIterator = List.listIterator();

i.e., we can't get List Iterator object from Set interface.

Which allows you to traverse in either directions. That is List Iterators traverse

two directions. So it has another methods hasPrevious() & previous() other than Iterator.

Methods in ListIterator

hashNext()

next()

previous()

hashPrevious()

remove()

nextIndex()

previousIndex()

/////////////////////////////////////////////////////////////////////////////////////////

---------------> Difference between ArrayList and LinkedList?

```````````

1)LinkedList store elements within a doubly-linked list data structure.

ArrayList store elements within a dynamically resizing array.

2)LinkedList allows for constant-time insertions or removals, but only sequential

access of elements. In other words, you can walk the list forwards or

backwards, but grabbing an element in the middle takes time proportional to the size

of the list. ArrayLists, on the other hand, allow random access, so you can grab any

element in constant time. But adding or removing from anywhere but the end requires shifting

all the latter elements over, either to make an opening or fill the gap.

3)LinkedList has more memory overhead than ArrayList because in ArrayList each index only

holds actual object (data) but in case of LinkedList each node holds both data and address

of next and previous node.

//////////////////////////////////////////////////////////////////////////////////////////////

------------->How to make a collection read only?

Use following methods:

Collections.unmodifiableList(list);

Collections.unmodifiableSet(set);

Collections.unmodifiableMap(map);

//////////////////////////////////////////////////////////////////////////////////////////////

------------> How to make a collection thread safe?

Use below methods:

Collections.synchronizedList(list);

Collections.synchronizedSet(set);

Collections.synchronizedMap(map);

//////////////////////////////////////////////////////////////////////////////////////////////

------------>Why there is not method like Iterator.add() to add elements to the collection?

The sole purpose of an Iterator is to enumerate through a collection.

All collections contain the add() method to serve your purpose. There would

be no point in adding to an Iterator because the collection may or may not be ordered

. And add()

method can not have same implementation for ordered and unordered collections.

//////////////////////////////////////////////////////////////////////////////////////

-----------> How to avoid ConcurrentModificationException while iterating a collection?

You should first try to find another alternative iterator which

are fail-safe. For example if you are using List and you can use ListIterator.

If it is legacy collection, you can use enumeration.

If above options are not possible then you can use one of three changes:

If you are using JDK1.5 or higher then you can use ConcurrentHashMap and

CopyOnWriteArrayList classes. It is the recommended approach.

You can convert the list to an array and then iterate on the array.

You can lock the list while iterating by putting it in a synchronized block.

////////////////////////////////////////////////////////////////////

-------------->What is UnsupportedOperationException?

This exception is thrown on invoked methods which are not supported by actual collection type.

For example, if you make a read-only list list using

“Collections.unmodifiableList(list)” and then call add() or remove() method,

what should happen. It should clearly throw UnsupportedOperationException.

////////////////////////////////////////////////////////////////////////////////////

----------> Blocking queue

A queue implementation which let calling thrrad to get blocked while queueing (When queue is full)

and dequieing when queue is empty

its iternal structur is

public class BlockingQueue {

private List queue = new LinkedList();

private int limit = 10;

public BlockingQueue(int limit){

this.limit = limit;

}

public synchronized void enqueue(Object item)

throws InterruptedException {

while(this.queue.size() == this.limit) {

wait();

}

if(this.queue.size() == 0) {

notifyAll();

}

this.queue.add(item);

}

public synchronized Object dequeue()

throws InterruptedException{

while(this.queue.size() == 0){

wait();

}

if(this.queue.size() == this.limit){

notifyAll();

}

return this.queue.remove(0);

}

}

////////////////////////////////////////////////////////////////////////////////////

---------> Comparable vs Comparatoor

What is Comparable and Comparator interface?

In java. all collection which have feature of automatic sorting, uses compare methods to ensure the correct sorting of elements. For example classes which use sorting are TreeSet, TreeMap etc.

To sort the data elements a class needs to implement Comparator or Comparable interface. That’s why all Wrapper classes like Integer,Double and String class implements Comparable interface.

Comparable helps in preserving default natural sorting, whereas Comparator helps in sorting the elements in some special required sorting pattern. The instance of comparator if passed usually as collection’s constructor argument in supporting collections.

Sorting an ArrayList in ascending order

// Java program to demonstrate working of Collections.sort()

import java.util.\*;

public class Collectionsorting

{

public static void main(String[] args)

{

// Create a list of strings

ArrayList<String> al = new ArrayList<String>();

al.add("Geeks For Geeks");

al.add("Friends");

al.add("Dear");

al.add("Is");

al.add("Superb");

/\* Collections.sort method is sorting the

elements of ArrayList in ascending order. \*/

Collections.sort(al);

// Let us print the sorted list

System.out.println("List after the use of" +

" Collection.sort() :\n" + al);

}

}

output

List after the use of Collection.sort() :

[Dear, Friends, Geeks For Geeks, Is, Superb]

Sorting an ArrayList in descending order

// Java program to demonstrate working of Collections.sort()

// to descending order.

import java.util.\*;

public class Collectionsorting

{

public static void main(String[] args)

{

// Create a list of strings

ArrayList<String> al = new ArrayList<String>();

al.add("Geeks For Geeks");

al.add("Friends");

al.add("Dear");

al.add("Is");

al.add("Superb");

/\* Collections.sort method is sorting the

elements of ArrayList in ascending order. \*/

Collections.sort(al, Collections.reverseOrder());

// Let us print the sorted list

System.out.println("List after the use of" +

" Collection.sort() :\n" + al);

}

}

Run on IDE

Output:

List after the use of Collection.sort() :

[Superb, Is, Geeks For Geeks, Friends, Dear

SINCE STRING CLASS IMPLEMEMNTS COMPARABLE NO REQUIREEMENT OF passing instance in colelction sort or explicitly defining interface

Sorting an ArrayList according to user defined criteria.

We can use Comparator Interface for this purpose.

// Java program to demonstrate working of Comparator

// interface and Collections.sort() to sort according

// to user defined criteria.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

// A class to represent a student.

class Student

{

int rollno;

String name, address;

// Constructor

public Student(int rollno, String name,

String address)

{

this.rollno = rollno;

this.name = name;

this.address = address;

}

// Used to print student details in main()

public String toString()

{

return this.rollno + " " + this.name +

" " + this.address;

}

}

class Sortbyroll implements Comparator<Student>

{

// Used for sorting in ascending order of

// roll number

public int compare(Student a, Student b)

{

return a.rollno - b.rollno;

}

}

// Driver class

class Main

{

public static void main (String[] args)

{

ArrayList<Student> ar = new ArrayList<Student>();

ar.add(new Student(111, "bbbb", "london"));

ar.add(new Student(131, "aaaa", "nyc"));

ar.add(new Student(121, "cccc", "jaipur"));

System.out.println("Unsorted");

for (int i=0; i<ar.size(); i++)

System.out.println(ar.get(i));

Collections.sort(ar, new Sortbyroll());

System.out.println("\nSorted by rollno");

for (int i=0; i<ar.size(); i++)

System.out.println(ar.get(i));

}

}

Run on IDE

Output :

Unsorted

111 bbbb london

131 aaaa nyc

121 cccc jaipur

Sorted by rollno

111 bbbb london

121 cccc jaipur

131 aaaa nyc

////////////////////////////////////////////////////////////////////////////////////---------------------->>Comparable interface

```````Implementing Comparable interface

Comparable interface provides one method compareTo(T o) to implement in any class so that two instances of that class can be compared. Signature of method is:

public int compareTo(T o);

Here, out of two instances to compare, one is instance itself on which method will be invoked, and other is passed as parameter o.

Lets see how our Employee class will look after implementing Comparable interface.

package corejava.compare;

public class Employee implements Comparable<Employee> {

private int id = -1;

private String firstName = null;

private String lastName = null;

private int age = -1;

public Employee(int id, String fName, String lName, int age) {

this.id = id;

this.firstName = fName;

this.lastName = lName;

this.age = age;

}

@Override

public int compareTo(Employee o) {

return this.id - o.id;

}

@Override

public String toString() {

return "Employee : " + id + " - " + firstName + " - " + lastName

+ " - " + age + "n";

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

// Other accessor methods

}

Default way to sort a list of employees, in our case, is by their id. Whatever, your default sorting order is, use in compare() method.

In implemented compare() method, we have simply returned the difference in employee ids of two instances. Two equal employee ids will return zero, indicating same object.

//////////////////////////////////////

-----------> Polymorphism

``````Ability by which we can create function and reference variable,which behave

differently in different programming context

"polymorphism in the context of object-oriented programming, is the ability to create a variable,

a function, or an object that has more than one form.”

An example of polymorphism is referring the instance of subclass, with reference variable of super-class. e.g.

Object o = new Object(); //o can hold the reference of any subtype

Object o = new String();

Object o = new Integer();

Compile timee Polymorphism--

program written in such a way that flow of control is decided in

compile time.

It is achieved using method overloading.

In method overloading, an object can have two or more methods with same name, BUT, with their method parameters different. These parameters may be different on two bases:

Parameters type

Type of method parameters can be different. e.g. java.util.Math.max() function comes with following versions:

public static double Math.max(double a, double b){..}

public static float Math.max(float a, float b){..}

public static int Math.max(int a, int b){..}

public static long Math.max(long a, long b){..}

-------------------------------> Runtime Polymorphism

`````````Runtime Polymorphism (dynamic binding or method overriding)

Runtime polymorphism is essentially referred as method overriding. Method overriding

is a feature which you get when you implement inheritance in your program.

A simple example can be from real world e.g. animals. An application can have Animal class,

and its specialized sub classes like Cat and Dog. These subclasses will override the

default behavior provided by Animal class + some of its own specific behavior.

public class Animal {

public void makeNoise()

{

System.out.println("Some sound");

}

}

class Dog extends Animal{

public void makeNoise()

{

System.out.println("Bark");

}

}

class Cat extends Animal{

public void makeNoise()

{

System.out.println("Meawoo");

}

}

public class Demo

{

public static void main(String[] args) {

Animal a1 = new Cat();

a1.makeNoise(); //Prints Meowoo

Animal a2 = new Dog();

a2.makeNoise(); //Prints Bark

}

}

Another term operator overloading is also there, e.g. “+” operator can be used to

add two integers as well as concat two sub-strings. Well, this is the only available support

for operator overloading in java,

and you can not have your own custom defined operator overloading in java.

Java dont have Operator overloading C# has

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

----------> Abstraction

``````Hiding up of inner detail is abstraction,

Like large arithmetic expression, you care of output not of which of operatior execute first.

types of abstrction

Data abstraction

Data abstraction is the way to create complex data types and exposing only meaningful operations to

interact with data type,

where as hiding all the implementation details from outside works.

Benefit of this approach involves capability of improving the implementation over time e.g.

solving performance issues is any. The idea is that such changes are not supposed to have any impact

on client code, since they involve no difference in the abstract behavior.

Control abstraction

A software is essentially a collection of numerous statements written in any programming language.

Most of the times, statement are similar and repeated over places multiple times.

Control abstraction is the process of identifying all such statements and expose them as a unit of work.

We normally use this feature when we create a function to perform any work.

ie same functionality statement in same block..

`````````````Use of Abstraction in application

--As abstraction is one of core principles of Object oriented programming practices, and

Java following all OOPs principles, abstraction is one of major building block of java language.

Data abstraction spans from creating simple data objects to complex collection

implementations such as HashMap or HashSet. Similarly, control abstraction can be seen

from defining simple function calls to complete open sourc

e frameworks. control abstraction is main force behind structured programming.

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

------------------>Encapsulation

````Wrapping data and methods within classes in combination with implementation

hiding (through access control) is often called encapsulation. The result is a data type with characteristics and behaviors. Encapsulation

essentially has both i.e. information hiding and implementation hiding.

----> mean wrapoing of data and method using implementation hiding(using access control) resulting into datatype with

behaviour

--->

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

----------->encapsulation vs Abstraction

Going forward, i will take example of our well known class HashMap. This class is responsible for storing key-value pair, searching based on key and do more things. From outside, client code only knows the method names and their behavior.

It calls these methods and live happily. This is actually what abstraction guidelines are.

Abstraction says that client code should call a method to add key-value pair, a method to retrieve

value based on key and so on. How it should be done? is not business of abstraction.

And here comes encapsulation, when you start writing actual code. You write HashMap.Entry class and create

variable table of type Entry[]. Then you declare all such things private and give public access to only put()

and get() methods etc. This is actually encapsulation. A realization of your desired abstraction.

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

------------------------>Abstraction vs Interface

An abstract method, is a method which is not implemented in place. An abstract method adds the

incompleteness to class, thus compiler wants to declare whole class abstract.

Only way to use an abstract class in your application is to extend this class. Its subclasses

if not declared abstract again, can be instantiated. The feature that subclass inherits the behavior

of super class, and super class can hold the

reference of subclass increases importance of abstract classes many fold.

'

Interface define contracts, which implementing classes

need to honor. These contracts are essentially unimplemented methods.

Java already has a keyword for unimplemented methods i.e. abstract. Java has

provision where any class can implement any interface, so all the methods declared in

interfaces need to be public only.

Are “public” and “public final” redundant for interface methods?

Yes.

All methods in an interface are implicitly public and abstract (but not final).

All fields in an interface are implicitly public, static and final.

The JLS states this. It also states that these modifiers can be left out.

Lets see an quick picture, how interface syntax looks like:

public interface TestInterface

{

void implementMe();

}

There is only one scenario, when you implement an interface and do

not override the method i.e. declare the implement class itself abstract.

public abstract class TestMain implements TestInterface

{

//No need to override implement Me

}

Otherwise, you must implement the method implementMe() in you class without any other exception.

public class TestMain implements TestInterface

{

@Override

public void implementMe() {

// TODO Auto-generated method stub

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Let’s note down differences between abstract classes and interfaces for quick review:

Interfaces have all methods inherently public and abstract. You can not

override this behavior by trying to reduce accessibility of methods. You

can not even declare the static methods. Only public and abstract.

On other side, abstract classes are flexible in declaring the methods.

You can define abstract methods with protected accessibility also.

Additionally, you can define static methods as well, provided they are not abstract.

Non-abstract static methods are allowed.

Interfaces can’t have fully defined methods. By definition,

interfaces are meant to provide only contract.

Abstract classes can have non-abstract methods without any limitation.

You can use any keyword with non-abstract methods as you will do in any other class.

Any class which want to use abstract class can extend abstract class using keyword

extends, whereas for implementing interfaces keyword used is implements.

A class can extends only one class, but can implement any number of interfaces.

This property is often referred as simulation of multiple inheritance in java.

Interface is absolutely abstract and cannot be instantiated; A Java abstract

class also cannot be instantiated, but can be invoked if a main() exists.

////////////////////////////////////////////////////////////////////////////////////-----------------> Member varuable of interface is staticgfinal by default

interface inter{

int a=10; // It becomes final static by default

public void interFunc();

}

class cls implements inter{

public void interFunc(){

System.out.println("In Class Method WITH a's Value as --> "+a);

}

}

class Test{

public static void main(String[] args){

inter in= new cls();

in.interFunc();

}

}

o/p ==== 10

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

--------------->Compile Time Means

```````The operations performed at compile time usually include syntax analysis,

various kinds of semantic analysis (e.g., type checks and instantiation of template) and code generation.

Programming language definitions usually specify compile time requirements that

source code must meet to be successfully compiled. For example, languages may stipulate

that the amount of storage required by types and variables can be deduced.

Properties of a program that can be reasoned about at compile time include range-checks

(e.g., proving that an array index will not exceed the array bounds), deadlock freedom in

concurrent languages, or timings (e.g., proving that a sequence of code takes no more than an

allocated amount of time).

Compile time occurs before link time (when the output of one or more compiled

files are joined together) and runtime (when a program is executed). In some programming

languages it may be necessary for some compilation and linking to occur at runtime. There is

a trade-off between compile-time and link-time in that many compile time operations can be deferred

to link-time without incurring extra run-time.

"Compile time" can also refer to the amount of time required for compilation.

///////////////////////////////////////////////////////////////////////////////////-------------------> Enum Type

````````Enumerations (in general) are generally a set of related constants

enums are a special type of class that always extends java.lang.Enum.

A simple usage will look like this:

public enum DIRECTION {

EAST,

WEST,

NORTH,

SOUTH //optionally can end with ";"

}

Here EAST, WEST, NORTH and SOUTH are final static inner classes of Direction of type Direction extends java.lang.Enum.

public abstract class Enum<E extends Enum<E>>

extends Object

implements Comparable<E>, Serializable

This clearly means that enums are comparable and serializable implicitly. Also, all enum types in java are singleton by default. So, you can compare enum types using ‘==’ operator also.

This also means that all enums extends java.lang.Enum, so they can not extend any other class because java does not support multiple inheritance this way. But, enums can implement any number of interfaces.

----To access enum inside a class,( enum is static)

-->outerObj.enum.enumMember

example for-->public class TestOuter

{

enum Direction

{

EAST,

WEST,

NORTH,

SOUTH

}

}

Like for above enum, if declared inside TestOuter.java, we can access a direction using TestOuter.Direction.NORTH.

`````````````````````````````````We can define enum type constructor

`````Though, you can give define your own constructors to initialize the state of enum types.

enum Direction {

// Enum types

EAST(0), WEST(180), NORTH(90), SOUTH(270);

// Constructor

private Direction(final int angle) {

this.angle = angle;

}

// Internal state

private int angle;

public int getAngle() {

return angle;

}

}

``````````````Extend an enum

. Enum types are final by default and hence can

not be extended. Yet, you are free to implement

any number of interfaces as you like.

----------------->Template method in enum

``````` the enum is basically a special class type, and can have methods and fields just like any other class. So, you can define a template for enum creation also

package enumTest;

public enum Direction {

// Enum types

EAST(0) {

@Override

public void shout() {

System.out.println("Direction is East !!!");

}

},

WEST(180) {

@Override

public void shout() {

System.out.println("Direction is West !!!");

}

},

NORTH(90) {

@Override

public void shout() {

System.out.println("Direction is North !!!");

}

},

SOUTH(270) {

@Override

public void shout() {

System.out.println("Direction is South !!!");

}

};

// Constructor

private Direction(final int angle) {

this.angle = angle;

}

// Internal state

private int angle;

public int getAngle() {

return angle;

}

// Abstract method which need to be implemented

public abstract void shout();

}

////////////////////////////////////////////////////////////////////////////////////////////

---------------------->Serialization

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable.

Serializable interface guarantees i.e. ability to serialize the classes. This interface recommends you to use serialVersioUID

Recommendations for serialVersionUID

The serialVersionUID is a universal version identifier for a Serializable class. Deserialization uses this number to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, then an InvalidClassException is thrown.

Always include it as a field, for example: “private static final long serialVersionUID = 7526472295622776147L; ” include this field even in the first version of the class, as a reminder of its importance.

Do not change the value of this field in future versions, unless you are knowingly making changes to the class which will render it incompatible with old serialized objects. If needed, follow above given guidelines.

Other points to keep remember

Use javadoc’s @serial tag to denote Serializable fields.

The .ser extension is conventionally used for files representing serialized objects.

No static or transient fields undergo default serialization.

Extendable classes should not be Serializable, unless necessary.

Inner classes should rarely, if ever, implement Serializable.

Container classes should usually follow the style of Hashtable, which implements Serializable by storing keys and values, as opposed to a large hash table data structure.

Sample implementation class

package staticTest;

import java.io.Serializable;

import java.text.StringCharacterIterator;

import java.util.\*;

import java.io.\*;

public final class UserDetails implements Serializable {

/\*\*

\* This constructor requires all fields

\*

\* @param aFirstName

\* contains only letters, spaces, and apostrophes.

\* @param aLastName

\* contains only letters, spaces, and apostrophes.

\* @param aAccountNumber

\* is non-negative.

\* @param aDateOpened

\* has a non-negative number of milliseconds.

\*/

public UserDetails(String aFirstName, String aLastName, int aAccountNumber,

Date aDateOpened) {

super();

setFirstName(aFirstName);

setLastName(aLastName);

setAccountNumber(aAccountNumber);

setDateOpened(aDateOpened);

// there is no need here to call verifyUserDetails.

}

// The default constructor

public UserDetails() {

this("FirstName", "LastName", 0, new Date(System.currentTimeMillis()));

}

public final String getFirstName() {

return fFirstName;

}

public final String getLastName() {

return fLastName;

}

public final int getAccountNumber() {

return fAccountNumber;

}

/\*\*

\* Returns a defensive copy of the field so that no one can change this

\* field.

\*/

public final Date getDateOpened() {

return new Date(fDateOpened.getTime());

}

/\*\*

\* Names must contain only letters, spaces, and apostrophes. Validate before

\* setting field to new value.

\*

\* @throws IllegalArgumentException

\* if the new value is not acceptable.

\*/

public final void setFirstName(String aNewFirstName) {

verifyNameProperty(aNewFirstName);

fFirstName = aNewFirstName;

}

/\*\*

\* Names must contain only letters, spaces, and apostrophes. Validate before

\* setting field to new value.

\*

\* @throws IllegalArgumentException

\* if the new value is not acceptable.

\*/

public final void setLastName(String aNewLastName) {

verifyNameProperty(aNewLastName);

fLastName = aNewLastName;

}

/\*\*

\* Validate before setting field to new value.

\*

\* @throws IllegalArgumentException

\* if the new value is not acceptable.

\*/

public final void setAccountNumber(int aNewAccountNumber) {

validateAccountNumber(aNewAccountNumber);

fAccountNumber = aNewAccountNumber;

}

public final void setDateOpened(Date aNewDate) {

// make a defensive copy of the mutable date object

Date newDate = new Date(aNewDate.getTime());

validateAccountOpenDate(newDate);

fDateOpened = newDate;

}

/\*\*

\* The client's first name.

\*

\* @serial

\*/

private String fFirstName;

/\*\*

\* The client's last name.

\*

\* @serial

\*/

private String fLastName;

/\*\*

\* The client's account number.

\*

\* @serial

\*/

private int fAccountNumber;

/\*\*

\* The date the account was opened.

\*

\* @serial

\*/

private Date fDateOpened;

/\*\*

\* Determines if a de-serialized file is compatible with this class.

\* Included here as a reminder of its importance.

\*/

private static final long serialVersionUID = 7526471155622776147L;

/\*\*

\* Verify that all fields of this object take permissible values

\*

\* @throws IllegalArgumentException

\* if any field takes an unpermitted value.

\*/

private void verifyUserDetails() {

validateAccountNumber(fAccountNumber);

verifyNameProperty(fFirstName);

verifyNameProperty(fLastName);

validateAccountOpenDate(fDateOpened);

}

/\*\*

\* Ensure names contain only letters, spaces, and apostrophes.

\*

\* @throws IllegalArgumentException

\* if field takes an unpermitted value.

\*/

private void verifyNameProperty(String aName) {

boolean nameHasContent = (aName != null) && (!aName.equals(""));

if (!nameHasContent) {

throw new IllegalArgumentException(

"Names must be non-null and non-empty.");

}

StringCharacterIterator iterator = new StringCharacterIterator(aName);

char character = iterator.current();

while (character != StringCharacterIterator.DONE) {

boolean isValidChar = (Character.isLetter(character)

|| Character.isSpaceChar(character) || character == ''');

if (isValidChar) {

// do nothing

} else {

String message = "Names can contain only letters, spaces, and apostrophes.";

throw new IllegalArgumentException(message);

}

character = iterator.next();

}

}

/\*\*

\* AccountNumber must be non-negative.

\*

\* @throws IllegalArgumentException

\* if field takes an unpermitted value.

\*/

private void validateAccountNumber(int aAccountNumber) {

if (aAccountNumber < 0) {

String message = "Account Number must be greater than or equal to 0.";

throw new IllegalArgumentException(message);

}

}

/\*\*

\* DateOpened must be after 1970.

\*

\* @throws IllegalArgumentException

\* if field takes an unpermitted value.

\*/

private void validateAccountOpenDate(Date aDateOpened) {

if (aDateOpened.getTime() < 0) {

throw new IllegalArgumentException(

"Date Opened must be after 1970.");

}

}

/\*\*

\* Always treat de-serialization as a full-blown constructor, by validating

\* the final state of the de-serialized object.

\*/

private void readObject(ObjectInputStream aInputStream)

throws ClassNotFoundException, IOException {

// always perform the default de-serialization first

aInputStream.defaultReadObject();

// make defensive copy of the mutable Date field

fDateOpened = new Date(fDateOpened.getTime());

// ensure that object state has not been corrupted or tampered with

// malicious code

verifyUserDetails();

}

/\*\*

\* This is the default implementation of writeObject. Customise if

\* necessary.

\*/

private void writeObject(ObjectOutputStream aOutputStream)

throws IOException {

// perform the default serialization for all non-transient, non-static

// fields

aOutputStream.defaultWriteObject();

}

}

Lets see now how to do serialization and deserialization.

Serialization and deserialization of object

package serializationTest;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.util.Calendar;

import java.util.Date;

public class TestUserDetails {

public static void main(String[] args) {

// Create new UserDetails object

UserDetails myDetails = new UserDetails("Lokesh", "Gupta", 102825,

new Date(Calendar.getInstance().getTimeInMillis()));

// Serialization code

try {

FileOutputStream fileOut = new FileOutputStream("userDetails.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(myDetails);

out.close();

fileOut.close();

} catch (IOException i) {

i.printStackTrace();

}

// De-serialization code

@SuppressWarnings("unused")

UserDetails deserializedUserDetails = null;

try {

FileInputStream fileIn = new FileInputStream("userDetails.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

deserializedUserDetails = (UserDetails) in.readObject();

in.close();

fileIn.close();

// verify the object state

System.out.println(deserializedUserDetails.getFirstName());

System.out.println(deserializedUserDetails.getLastName());

System.out.println(deserializedUserDetails.getAccountNumber());

System.out.println(deserializedUserDetails.getDateOpened());

} catch (IOException ioe) {

ioe.printStackTrace();

} catch (ClassNotFoundException cnfe) {

cnfe.printStackTrace();

}

}

}

Output:

Lokesh

Gupta

102825

Wed Nov 21 15:06:34 GMT+05:30 2012

Example of serialization:

Let's say you have a class person like the following:  
  
public class Person implements java.io.Serializable {  
 /\*\*  
 \*   
 \*/  
 private static final long serialVersionUID = 1L;  
 public String firstName;  
 public String lastName;  
 public int age;  
 public String address;  
  
 public void play() {  
 System.out.println(String.format(  
 "If I win, send me the trophy to this address: %s", address));  
 }  
 @Override  
 public String toString() {  
 return String.format(".....Person......\nFirst Name = %s\nLast Name = %s", firstName, lastName);  
 }  
}  
and then you create an object like this:  
  
Person william = new Person();  
 william.firstName = "William";  
 william.lastName = "Kinaan";  
 william.age = 26;  
 william.address = "Lisbon, Portugal";  
You can serialise that object to many streams. I will do that to two streams:  
  
Serialization to standard output:  
  
public static void serializeToStandardOutput(Person person)  
 throws IOException {  
 OutputStream outStream = System.out;  
 ObjectOutputStream stdObjectOut = new ObjectOutputStream(outStream);  
 stdObjectOut.writeObject(person);  
 stdObjectOut.close();  
 outStream.close();  
 }  
Serialization to a file:  
  
public static void serializeToFile(Person person) throws IOException {  
 OutputStream outStream = new FileOutputStream("person.ser");  
 ObjectOutputStream fileObjectOut = new ObjectOutputStream(outStream);  
 fileObjectOut.writeObject(person);  
 fileObjectOut.close();  
 outStream.close();  
 }  
Then:  
  
Deserialize from file:  
  
public static void deserializeFromFile() throws IOException,  
 ClassNotFoundException {  
 InputStream inStream = new FileInputStream("person.ser");  
 ObjectInputStream fileObjectIn = new ObjectInputStream(inStream);  
 Person person = (Person) fileObjectIn.readObject();  
 System.out.println(person);  
 fileObjectIn.close();  
 inStream.close();  
 }

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

------------------->Cloning

``````Jav cloning makes an identical copy of object,

By defau;t java cloning is field by field copy, ie as the oBject class

dont have idea about structure of class on which clone method will be invoked,

so jvm when called for cloning does--:

1) if class has only primitive members then a completely new copy

of the object will be created and the reference to the

new object copy will be returned.

2) if the class containes member of anuy class type then only the object refeence

to those members are copied and hence member references in both original object

as well as cloned object to the same object

````````

To make a clone of class following infrastructure is is followed

->A) You must implement Cloneable interface.

->B) You must override clone() method from Object class.

/\*

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object.

The general intent is that, for any object x, the expression:

1) x.clone() != x will be true

2) x.clone().getClass() == x.getClass() will be true, but these are not absolute requirements.

3) x.clone().equals(x) will be true, this is not an absolute requirement.

\*/

protected native Object [More ...] clone() throws CloneNotSupportedException

--->First statement guarantees that cloned object will have separate memory address assignment.

seperate memory for both

-->Second statement suggest that original and cloned objects should have

same class type, but it is not mandatory.

same class type for both

-->Third statement suggest that original and cloned objects should have be equal using equals() method, but it is not mandatory.

both shld be equals but not mandatory for non primitive

java example---

public class Employee implements Cloneable{

private int empoyeeId;

private String employeeName;

private Department department;

public Employee(int id, String name, Department dept)

{

this.empoyeeId = id;

this.employeeName = name;

this.department = dept;

}

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

//Accessor/mutators methods will go there

}

Our Department class has two attributes. id and name.

public class Department

{

private int id;

private String name;

public Department(int id, String name)

{

this.id = id;

this.name = name;

}

//Accessor/mutators methods will go there

}

So, if we need to clone the Employee class, then we need to do something like this.

public class TestCloning {

public static void main(String[] args) throws CloneNotSupportedException

{

Department dept = new Department(1, "Human Resource");

Employee original = new Employee(1, "Admin", dept);

//Lets create a clone of original object

Employee cloned = (Employee) original.clone();

//Let verify using employee id, if cloning actually workded

System.out.println(cloned.getEmpoyeeId()); 1

//Verify JDK's rules

//Must be true and objects must have different memory addresses

System.out.println(original != cloned);

//As we are returning same class; so it should be true

System.out.println(original.getClass() == cloned.getClass());

//Default equals method checks for refernces so it should be false. If we want to make it true,

//we need to override equals method in Employee class.

System.out.println(original.equals(cloned));

}

}

Output:

1

true

true

false

Great, we successfully cloned the Employee object. But, remember we have two references of same object and now both will change the state of object in different parts of application. Want to see how?

Lets see:

public class TestCloning {

public static void main(String [] args) throws CloneNotSupportedException {

Department hr = new Department(1, "Human Resource");

Employee original = new Employee(1, "Admin", hr);

Employee cloned = (Employee) original.clone();

//Let change the department name in cloned object and we will verify in original object

cloned.getDepartment().setName("Finance");

System.out.println(original.getDepartment().getName());

}

}

Output: Finance

--------------->Shallow Cloning

This is default implementation in java. In overridden clone method, if you are not cloning all the object types (not primitives), then you are making a shallow copy.

All above examples are of shallow copy only, because we have not cloned the Department object on Employee class’s clone method. Now, i will move on to next section where we will see the deep cloning.

---------->>Deep cloning

It is the desired behavior in most the cases. We want a clone which is independent of original and making changes in clone should not affect original.

Let see how it can be done in our case.

//Modified clone() method in Employee class

@Override

protected Object clone() throws CloneNotSupportedException {

Employee cloned = (Employee)super.clone();

cloned.setDepartment((Department)cloned.getDepartment().clone());

return cloned;

}

I modified the Employee classes clone() method and added following clone method in Department class.

//Defined clone method in Department class.

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

---->Test deep cloning

public class TestCloning {

public static void main(String[] args) throws CloneNotSupportedException {

Department hr = new Department(1, "Human Resource");

Employee original = new Employee(1, "Admin", hr);

Employee cloned = (Employee) original.clone();

//Let change the department name in cloned object and we will verify in original object

cloned.getDepartment().setName("Finance");

System.out.println(original.getDepartment().getName());

}

}

Output:

Human Resource

Best practices

1) When you don’t know whether you can call the clone() method of a particular class as you are not sure if it is implemented in that class, you can check with checking if the class is instance of “Cloneable” interface as below.

if(obj1 instanceof Cloneable){

obj2 = obj1.clone();

}

//Dont do this. Cloneabe dont have any methods

obj2 = (Cloneable)obj1.clone();

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

----------------> Executor service

An advantage I see is in managing/scheduling several threads. With ExecutorService, you don't

have to write your own thread manager which can be plagued with bugs. This is especially useful if

your program needs to run several threads at once. For example you want to execute two threads at a time,

you can easily do it like this:

ExecutorService exec = Executors.newFixedThreadPool(2);

exec.execute(new Runnable() {

public void run() {

System.out.println("Hello world");

}

});

exec.shutdown();

The example may be trivial, but try to think that the "hello world"

line consists of a heavy operation and you want that operation to run in several

threads at a time in order to improve your program's performance. This is just one example,

there are still many cases that you want to schedule or run several threads and use ExecutorService

as your thread manager.

For running a single thread, I don't see any clear advantage of using ExecutorService.

``````````> we can seperte task creation and its execution using executor service

This is around the Executor interface, its sub-interface ExecutorService,

and the ThreadPoolExecutor class that implements both interfaces. This mechanism separates

the task creation and its execution. With an executor, you only

have to implement the Runnable objects and send them to the executor.

Sample java program------

1) Create a task to execute

Obviously, first step is to have a task which you would like to execute using Executors.

class Task implements Runnable

{

private String name;

public Task(String name)

{

this.name = name;

}

public String getName() {

return name;

}

@Override

public void run()

{

try

{

Long duration = (long) (Math.random() \* 10);

System.out.println("Doing a task during : " + name);

TimeUnit.SECONDS.sleep(duration);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

2) Execute tasks using Executors

Now all you have to do is to create an instance of ThreadPoolExecutor and pass the tasks to be executed into it’s execute() method.

package com.howtodoinjava.demo.multithreading;

import java.util.concurrent.Executors;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

public class BasicThreadPoolExecutorExample

{

public static void main(String[] args)

{

//Use the executor created by the newCachedThreadPool() method

//only when you have a reasonable number of threads

//or when they have a short duration.

ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();

for (int i = 0; i <= 5; i++)

{

Task task = new Task("Task " + i);

System.out.println("A new task has been added : " + task.getName());

executor.execute(task);

}

executor.shutdown();

}

}

Output:

A new task has been added : Task 0

A new task has been added : Task 1

A new task has been added : Task 2

A new task has been added : Task 3

A new task has been added : Task 4

A new task has been added : Task 5

Doing a task during : Task 5

Doing a task during : Task 0

Doing a task during : Task 2

Doing a task during : Task 1

Doing a task during : Task 4

Doing a task during : Task 3

--------------------->>Executor service using callable..

------public class CallableExample {

public static class WordLengthCallable

implements Callable {

private String word;

public WordLengthCallable(String word) {

this.word = word;

}

public Integer call() {

return Integer.valueOf(word.length());

}

}

public static void main(String args[]) throws Exception {

ExecutorService pool = Executors.newFixedThreadPool(3);

Set<Future<Integer>> set = new HashSet<Future?Integer>>();

for (String word: args) {

Callable<Integer> callable = new WordLengthCallable(word);

Future<Integer> future = pool.submit(callable);

set.add(future);

}

int sum = 0;

for (Future<Integer> future : set) {

sum += future.get();

}

System.out.printf("The sum of lengths is %s%n", sum);

System.exit(sum);

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////// SPRING FRAMEWORKS ///////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

----> Inversion Control

, inversion of control (IoC) is a programming technique in which object

coupling is bound at run time by an assembler

object and is typically not known at compile time using static analysis.

In Java, dependency injection may happen through 3 ways:

A constructor injection

A setter injection

An interface injection

IOC is accomplished using Dependency INjection

/////////////////////////////////////////////////////////////////////////////////////////

--------------->Dispatcher servlet

``````After receiving an HTTP request, DispatcherServlet consults the HandlerMapping (configuration files) to call the appropriate Controller. The Controller takes the request and calls the appropriate service methods and set model data and then returns view name to the DispatcherServlet. The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request. Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

<web-app>

<display-name>Archetype Created Web Application</display-name>

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

By default, DispatcherServlet loads its configuration file using <servlet\_name>-servlet.xml. E.g. with above web.xml file, DispatcherServlet will try to find spring-servlet.xml file in classpath.

ContextLoaderListener reads the spring configuration file (with value given against “contextConfigLocation” in web.xml), parse it and loads the beans defined in that config file. e.g.

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dispatcher servlet2

->The dispatcher-servlet.xml file contains all of your configuration for Spring MVC. So in it you will find beans such as ViewHandlerResolvers, ConverterFactories, Interceptors and so forth. All of these beans are part of Spring MVC which is a framework that structures how you handle web requests, providing useful features such as databinding, view resolution and request mapping.

The application-context.xml can optionally be included when using Spring MVC or any other framework for that matter. This gives you a container that may be used to configure other types of spring beans that provide support for things like data persistence. Basically, in this configuration file is where you pull in all of the other goodies Spring offers.

These configuration files are configured in the web.xml file as shown:

Dispatcher Config

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>WEB-INF/spring/servlet-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

Application Config

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/application-context.xml</param-value>

</context-param>

<!-- Creates the Spring Container shared by all Servlets and Filters -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

To configure controllers, annotate them with @Controller then include the following in the dispatcher-context.xml file:

<mvc:annotation-driven/>

<context:component-scan base-package="package.with.controllers.\*\*" />

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

----------->>Bean scopes

`````` scope of an object at the Jaava class level. Beans can be defined to be

deployed in one of

a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of

which three are available only if you are using a web-aware ApplicationContext).

Scope Description

```````````>singleton

--->Scopes a single bean definition to a single object instance

per Spring IoC container.

--> When scope is set as singlton only one

shared instance of shared instance of bean is managed

--> all future request to bean using that id will

result in id or ids matching that bean definition will result in that one

specific bean instance being returned by the Spring container.

-->Please be aware that Spring's concept of a singleton bean

is quite different from the Singleton pattern as defined in the seminal

Gang of Four (GoF) patterns book. The GoF Singleton hard codes the scope

of an object such that one and only one instance of a particular class will

ever be created per ClassLoader. The scope of the Spring singleton is best

described as per container and per bean. This means that if you define one

bean for a particular class in a single Spring container,

<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default); using spring-beans-2.0.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>

``````````>prototype

```>Bean with scope prototype, result in creation of new bean instance, everythime

request for that bean with id is made.via a programmatic getBean() method call on the container)

Use prototype bean for stateful, singleton for stateless beans..

this is how we can define prototypebean...

<!-- using spring-beans-2.0.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>

spring contaier does not manage complete protoype bean lifecycle

ie. for Prototype bean container instantiates, configures,

decorates and otherwise assembles a prototype object and hand to client.

no initializa or destruction callbakc are called.

-> client has responsibility to release prototype bean,

we can use custome bean post procesor, whih would hold reference to bean that need to be cleaned up

->------------\_\_---------\_\_\_-----------\_\_\_\_-----------

1)FOR REQUEST SESSION GLOBAL SESION requires some extra configuration

than singleton and prototype.

2) It require Aplication context only, not( XmlBeanFactory or ClassPathXmlApplicationContext,)

3) Dispatcher servlet is required

------------\_\_---------\_\_\_-----------\_\_\_\_-----------

Scopes a single bean definition to any number of object instances.

**request**

Scopes a single bean definition to the lifecycle of a single HTTP

request; that is each and every HTTP request will have

its own instance of a bean created off the back of a single bean definition.

Only valid in the context of a web-aware Spring ApplicationContext.

-><bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

Bean will be created for each bean definition for each and every HTTP request

-> HTTP request. That is, the 'loginAction' bean will be effectively scoped at the HTTP request level.

->all bean will be unaffected isolated for independent request

**session**

Scopes a single bean definition to the lifecycle of a HTTP Session.

Only valid in the context of a web-aware Spring ApplicationContext.

->

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

With the above bean definition in place, the Spring container

will create a brand new instance of the UserPreferences bean using the

'userPreferences' bean definition for the lifetime of a single HTTP Session.

In other words, the 'userPreferences' bean will be effectively scoped at the

HTTP Session level. Just like request-scoped beans, you can change the

internal state of the instance that is created as much as you want, safe in

the knowledge that other HTTP Session instances that are also using instances

created off the back of the same 'userPreferences' bean definition will not be

seeing these changes in state since they are particular to an individual HTTP Session. When the HTTP Session is eventually

discarded, the bean that is scoped to that particular HTTP Session will also

be discarded.

global session

Scopes a single bean definition to the lifecycle of a global HTTP Session.

Typically only valid when used in a portlet context. Only valid in the context

of a web-aware Spring ApplicationContext.

<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>

->GLobals session shared among alll portlets

->Both portlets and servlets receive an http

request and return a response, which is usally some HTML that can be rendered by a browser. A portlet is used in the context of a "Portal", the idea being that a single page seen by the user has lots of parts, think tiles, each coming from a different portlet.

Now, you can get that "tiled" effect

from normal servets (See Struts + Tiles for an example of how)

the extra bit from the portlets is that the portlets are in a richer

environment provided by the Portal, extra APIs are provided so that

what is displayed by any portlet can be configured by individual users

to their preferences, and the porlets can communicate with each other -

press a button in one, something happens in a another.

////////////////////////////////////////////////////////////////////////////

-------------->Stateful vs stateless bean

``````````For stateless session beans the server can maintain a variable amount of instances in a pool. Each time a client requests such a stateless bean (e.g. through a method) a random instance is chosen to serve that request. That means if the client does two subsequent requests it is possible that two different instances of the stateless bean serve the requests. In fact

there is no conversational state between the two requests.

Also if the client disappears, the stateless bean does not get

destroyed and can serve the next request from another client.

On the other hand a stateful session bean is closely connected to the

client. Each instance is created and bounded to a single client and serves

only requests from that particular client. So happens that if you do two

subsequent requests on a stateful bean, your request will be served always

from the same instance of the bean. That means you can maintain a conversational

state between the requests. At the end of the lifecyle the client calls a remove

method and the bean is being destroyed/ready for garbage collection.

**When to use stateless or stateful?**

That mainly depends on whether you want to maintain the conversational state.

For example if you have a method that adds up to numbers and return the result

you use a stateless bean because its a one time operation. If you call this method

a second time with other numbers you are not interested in the result of the previous addition anymore.

But if you want for example count the number of requests a client has done, you

have to use a stateful bean. In this scenario it is important to know how often

the client has requested the bean method before, so you have to maintain

conversational state in the bean (e.g. with a variable). If you would use a

stateless bean here the request of the client would be served each time from

a different bean what messes up your results.

EXAMPLE STATEFULL

## Counter

package org.superbiz.counter;  
  
import javax.ejb.Stateful;  
  
/\*\*  
 \* This is an EJB 3 style pojo stateful session bean  
 \* Every stateful session bean implementation must be annotated  
 \* using the annotation @Stateful  
 \* This EJB has 2 business interfaces: CounterRemote, a remote business  
 \* interface, and CounterLocal, a local business interface  
 \* <p/>  
 \* Per EJB3 rules when the @Remote or @Local annotation isn't present  
 \* in the bean class (this class), all interfaces are considered  
 \* local unless explicitly annotated otherwise. If you look  
 \* in the CounterRemote interface, you'll notice it uses the @Remote  
 \* annotation while the CounterLocal interface is not annotated relying  
 \* on the EJB3 default rules to make it a local interface.  
 \*/  
//START SNIPPET: code  
@Stateful  
public class Counter {  
  
 private int count = 0;  
  
 public int count() {  
 return count;  
 }  
  
 public int increment() {  
 return ++count;  
 }  
  
 public int reset() {  
 return (count = 0);  
 }  
}

## CounterTest

The Counter class is tested by obtaining a Context object and performing a JNDI lookup on it, to retrieve an instance of the Counter bean. After some state manipulation, a new instance is fetched from the container and we can see that it's a new instance.

package org.superbiz.counter;  
  
import junit.framework.TestCase;  
  
import javax.ejb.embeddable.EJBContainer;  
import javax.naming.Context;  
  
public class CounterTest extends TestCase {  
  
 //START SNIPPET: local  
 public void test() throws Exception {  
  
 final Context context = EJBContainer.createEJBContainer().getContext();  
  
 Counter counterA = (Counter) context.lookup("java:global/simple-stateful/Counter");  
  
 assertEquals(0, counterA.count());  
 assertEquals(0, counterA.reset());  
 assertEquals(1, counterA.increment());  
 assertEquals(2, counterA.increment());  
 assertEquals(0, counterA.reset());  
  
 counterA.increment();  
 counterA.increment();  
 counterA.increment();  
 counterA.increment();  
  
 assertEquals(4, counterA.count());  
  
 // Get a new counter  
 Counter counterB = (Counter) context.lookup("java:global/simple-stateful/Counter");  
  
 // The new bean instance starts out at 0  
 assertEquals(0, counterB.count());  
 }  
 //END SNIPPET: local  
}

# Running

-------------------------------------------------------  
 T E S T S  
-------------------------------------------------------  
Running org.superbiz.counter.CounterTest  
Apache OpenEJB 4.0.0-beta-1 build: 20111002-04:06  
http://tomee.apache.org/  
INFO - openejb.home = /Users/dblevins/examples/simple-stateful  
INFO - openejb.base = /Users/dblevins/examples/simple-stateful  
INFO - Using 'javax.ejb.embeddable.EJBContainer=true'  
INFO - Configuring Service(id=Default Security Service, type=SecurityService, provider-id=Default Security Service)  
INFO - Configuring Service(id=Default Transaction Manager, type=TransactionManager, provider-id=Default Transaction Manager)  
INFO - Found EjbModule in classpath: /Users/dblevins/examples/simple-stateful/target/classes  
INFO - Beginning load: /Users/dblevins/examples/simple-stateful/target/classes  
INFO - Configuring enterprise application: /Users/dblevins/examples/simple-stateful  
INFO - Configuring Service(id=Default Stateful Container, type=Container, provider-id=Default Stateful Container)  
INFO - Auto-creating a container for bean Counter: Container(type=STATEFUL, id=Default Stateful Container)  
INFO - Configuring Service(id=Default Managed Container, type=Container, provider-id=Default Managed Container)  
INFO - Auto-creating a container for bean org.superbiz.counter.CounterTest: Container(type=MANAGED, id=Default Managed Container)  
INFO - Enterprise application "/Users/dblevins/examples/simple-stateful" loaded.  
INFO - Assembling app: /Users/dblevins/examples/simple-stateful  
INFO - Jndi(name="java:global/simple-stateful/Counter!org.superbiz.counter.Counter")  
INFO - Jndi(name="java:global/simple-stateful/Counter")  
INFO - Jndi(name="java:global/EjbModule309142400/org.superbiz.counter.CounterTest!org.superbiz.counter.CounterTest")  
INFO - Jndi(name="java:global/EjbModule309142400/org.superbiz.counter.CounterTest")  
INFO - Created Ejb(deployment-id=Counter, ejb-name=Counter, container=Default Stateful Container)  
INFO - Created Ejb(deployment-id=org.superbiz.counter.CounterTest, ejb-name=org.superbiz.counter.CounterTest, container=Default Managed Container)  
INFO - Started Ejb(deployment-id=Counter, ejb-name=Counter, container=Default Stateful Container)  
INFO - Started Ejb(deployment-id=org.superbiz.counter.CounterTest, ejb-name=org.superbiz.counter.CounterTest, container=Default Managed Container)  
INFO - Deployed Application(path=/Users/dblevins/examples/simple-stateful)  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.098 sec  
  
Results :  
  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

////////////////////////////////////////////v//////////////////////////////////////////////////////////////////

---------------->> we can have mmultiple application

context/spring contaiers that manage spring bean with different config

````````>ApplicationContext (as an interface, and by the direct

implementation flavours) is the mean of implementing this IoC container,

as opposed to the BeanFactory, which is now (a sparsely used and) more direct

way of managing beans, which by the way provides the base implementation features

for the ApplicationContext.

As per your second question, you can have multiple

instances of ApplicationContexts, in that case, they

will be completely isolated, each with its own configuration.

/////////////////////////////////////////////////////////////////////////////////////////////////////

---------> What is dispatcher serlet

`````> Simply put, in the Front Controller design pattern,

a single controller is responsible for directing incoming HttpRequests

to all of an application’s other controllers and handlers.

he job of the DispatcherServlet is to take an incoming URI and find the right combination of handlers (generally methods on Controller classes) and views (generally JSPs) that combine to form the page or resource that's supposed to be found at that location.

I might have

a file /WEB-INF/jsp/pages/Home.jsp

and a method on a class

@RequestMapping(value="/pages/Home.html")

private ModelMap buildHome() {

return somestuff;

}

///////////////////////////////////////////////////////////////////////////////////

----------------> Simple spring app

`````Now add DispatcherServlet entry in web.xml file so that all incoming requests come though DispatcherServlet only.

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

Now add below entries in spring configuration file.

<beans>

<!-- Scan all classes in this path for spring specific annotations -->

<context:component-scan base-package="com.howtodoinjava.demo" />

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

<!-- Vierw resolver configuration -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

Add controller code.

@Controller

@RequestMapping("/employee-module")

public class EmployeeController

{

@Autowired

EmployeeManager manager;

@RequestMapping(value = "/getAllEmployees", method = RequestMethod.GET)

public String getAllEmployees(Model model)

{

model.addAttribute("employees", manager.getAllEmployees());

return "employeesListDisplay";

}

}

Additionally you should add manager and dao layer classes as well. Finally you add the jsp file to display the view.

///////////////////////////////////////////////////////////////////////////////////////////

Spring MVC Interview Questions with Answers

March 2, 2015 by Lokesh Gupta

These Spring MVC interview questions and answers have been written to help you prepare for the interviews and quickly revise the concepts in general. I will strongly suggest you to go deeper into each concept if you have extra time. The more you know, more you are confident.

What is Spring MVC framework?

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between model, view and controller parts of application. Spring framework provides lots of advantages over other MVC frameworks e.g.

Clear separation of roles – controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.

Powerful and straightforward configuration of both framework and application classes as JavaBeans.

Reusable business code – no need for duplication. You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.

Customizable binding and validation

Customizable handler mapping and view resolution

Customizable locale and theme resolution

A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier. etc.

**What is DispatcherServlet and ContextLoaderListener?**

Spring’s web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central Servlet that handles all the HTTP requests and responses. Spring’s DispatcherServlet however, does more than just that. It is completely integrated with the Spring IoC container so it allows you to use every feature that Spring has.

After receiving an HTTP request, DispatcherServlet consults the HandlerMapping (configuration files) to call the appropriate Controller. The Controller takes the request and calls the appropriate service methods and set model data and then returns view name to the DispatcherServlet. The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request. Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

<web-app>

<display-name>Archetype Created Web Application</display-name>

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

By default, DispatcherServlet loads its configuration file using <servlet\_name>-servlet.xml. E.g. with above web.xml file, DispatcherServlet will try to find spring-servlet.xml file in classpath.

**ContextLoaderListener reads the spring** configuration file (with value given against “contextConfigLocation” in web.xml), parse it and loads the beans defined in that config file. e.g.

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

**What is the front controller class of Spring MVC?**

A front controller is defined as “a controller which handles all requests for a Web Application.” DispatcherServlet (actually a servlet) is the front controller in Spring MVC that intercepts every request and then dispatches/forwards requests to an appropriate controller.

When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring’s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.

How to use Java based configuration?

To configure java based MVC application, first add required dependencies.

<!-- Spring MVC support -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<!-- Tag libs support for view layer -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>taglibs</groupId>

<artifactId>standard</artifactId>

<version>1.1.2</version>

<scope>runtime</scope>

</dependency>

Now add DispatcherServlet entry in web.xml file so that all incoming requests come though DispatcherServlet only.

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

Now add below entries in spring configuration file.

<beans>

<!-- Scan all classes in this path for spring specific annotations -->

<context:component-scan base-package="com.howtodoinjava.demo" />

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

<!-- Vierw resolver configuration -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

Add controller code.

@Controller

@RequestMapping("/employee-module")

public class EmployeeController

{

@Autowired

EmployeeManager manager;

@RequestMapping(value = "/getAllEmployees", method = RequestMethod.GET)

public String getAllEmployees(Model model)

{

model.addAttribute("employees", manager.getAllEmployees());

return "employeesListDisplay";

}

}

Additionally you should add manager and dao layer classes as well. Finally you add the jsp file to display the view.

I will suggest to read above linked tutorial for complete understanding.

Read More : Spring MVC Hello World Example

How can we use Spring to create Restful Web Service returning JSON response?

For adding JSON support to your spring application, you will need to add Jackson dependency in first step.

<!-- Jackson JSON Processor -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.4.1</version>

</dependency>

Now you are ready to return JSON response from your MVC controller. All you have to do is return JAXB annotated object from method and use @ResponseBody annotation on this return type.

@Controller

public class EmployeeRESTController

{

@RequestMapping(value = "/employees")

public @ResponseBody EmployeeListVO getAllEmployees()

{

EmployeeListVO employees = new EmployeeListVO();

//Add employees

return employees;

}

}

Alternatively, you can use @RestController annotation in place of @Controller annotation. This will remove the need to using @ResponseBody.

@RestController = @Controller + @ResponseBody

So you can write the above controller as below.

@RestController

public class EmployeeRESTController

{

@RequestMapping(value = "/employees")

public EmployeeListVO getAllEmployees()

{

EmployeeListVO employees = new EmployeeListVO();

//Add employees

return employees;

}

}

Read More : Spring REST Hello World JSON Example

**Can we have multiple Spring configuration files?**

YES. You can have multiple spring context files. There are two ways to make spring read and configure them.

a) Specify all files in web.xml file using contextConfigLocation init parameter.

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>

WEB-INF/spring-dao-hibernate.xml,

WEB-INF/spring-services.xml,

WEB-INF/spring-security.xml

</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

b) OR, you can import them into existing configuration file you have already configured.

<beans>

<import resource="spring-dao-hibernate.xml"/>

<import resource="spring-services.xml"/>

<import resource="spring-security.xml"/>

... //Other configuration stuff

</beans>

///////////////////////////////////////////////////////////////////////////////////////////////////v//////////////////////////////////////////////////////////////////

-------------->Difference between <context:annotation-config> vs <context:component-scan>?

1) First big difference between both tags is that <context:annotation-config>

is used to activate applied annotations in already registered beans in application

context. Note that it simply does not matter whether bean was registered by which mechanism e.g. using <context:component-scan>

or it was defined in application-context.xml file itself.

2) Second difference is driven from first difference itself. It registers the beans defined in config file into context + it also scans the annotations inside beans and activate them. So <context:component-scan> does what <context:annotation-config> does, but additionally it scan the packages and register the beans in application context.

////////////////////////////////////////////////////////////////////////////////////

------------------->Difference between @Component, @Controller, @Repository & @Service annotations?

1) The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context. To use this annotation, apply it over class as below:

@Component

public class EmployeeDAOImpl implements EmployeeDAO {

...

}

2) The @Repository annotation is a specialization of the @Component annotation

with similar use and functionality. In addition to importing the DAOs into

the DI container, it also makes the unchecked exceptions (thrown from DAO methods)

eligible for translation into Spring DataAccessException.

3) The @Service annotation is also a specialization of the component annotation.

It doesn’t currently provide any additional behavior over the @Component annotation,

but it’s a good idea to use @Service over @Component in service-layer classes because it specifies intent better.

4) @Controller annotation marks a class as a Spring Web MVC controller. It too

is a @Component specialization, so beans marked with it are automatically

imported into the DI container. When you add the @Controller annotation to a class,

you can use another annotation i.e. @RequestMapping;

to map URLs to instance methods of a class.

-->COMPONENT VS BEAN ANNOTATION

Component (and @Service and @Repository) are used to auto-detect and

auto-configure beans using classpath scanning. There's an implicit one-to-one

mapping between the annotated class and the bean (i.e. one bean per class).

Control of wiring is quite limited with this approach, since it's purely declarative.

@Bean is used to explicitly declare a single bean, rather than letting

Spring do it automatically as above. It decouples the declaration of the bean

from the class definition, and lets you create and configure beans exactly how you

choose.

//////////////////////////////////////////////////////////////////////////////////////////////

--------> What is inner bean in spring

`````` whenever a bean is used for only one particular property,

it’s advise to declare it as an inner bean. And the inner bean is

supported both in setter injection ‘property‘ and constructor injection

‘constructor-arg‘.

For example, let’s say we one Customer class having

reference of Person class. In our application, we will be

creating only one instance of Person class, and use it inside Customer.

public class Customer

{

private Person person;

//Setters and Getters

}

public class Person

{

private String name;

private String address;

private int age;

//Setters and Getters

}

Now inner bean declaration will look like this:

<bean id="CustomerBean" class="com.howtodoinjava.common.Customer">

<property name="person">

<!-- This is inner bean -->

<bean class="com.howtodoinjava.common.Person">

<property name="name" value="adminis"></property>

<property name="address" value="India"></property>

<property name="age" value="34"></property>

</bean>

</property>

</bean>

////////////////////////////////////////////////////////////////////////////////////////////////

--------------->Are Singleton beans thread safe in Spring Framework?

```````Spring framework does not do anything under the hood concerning

the multi-threaded behavior of a singleton bean. It is the developer’s

responsibility to deal with concurrency issue and thread safety of the

singleton bean.

While practically, most spring beans have no mutable state

(e.g. Service and DAO clases), and as such are trivially thread safe.

But if your bean has mutable state (e.g. View Model Objects), so you need to

ensure thread safety. The most easy and obvious solution for this problem

is to change bean scope of mutable beans from “singleton” to “prototype“.

///////////////////////////////////////////////////////////////////////////////////////////////////v//////////////////////////////////////////////////////////////////

-------------->What is autowiring

``````Spring comtainer is capable of autowiring collaboratng bean

This means that it is possible to automatically let Spring resolve

collaborators (other beans) for your bean by inspecting the contents of

the BeanFactory. Autowiring is specified per bean and can thus be enabled for

some beans, while other beans will not be autowired.

The following excerpt from the XML configuration file shows a bean being autowired

by name.

<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />

Apart from the autowiring modes provided in bean configuration file, autowiring can be specified in bean classes also using @Autowired annotation. To use @Autowired annotation in bean classes, you must first enable the annotation in spring application using below configuration.

<context:annotation-config />

Same can be achieved using AutowiredAnnotationBeanPostProcessor bean definition

in configuration file.

<bean class ="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

Now, when annotation configuration has been enables, you are free to a

utowire bean dependencies using @Autowired, the way you like.

@Autowired

public EmployeeDAOImpl ( EmployeeManager manager ) {

this.manager = manager;

}

/////////////////////////////////////////////////////////////////////////////////////////

------------->Explain different modes of bean autowiring?

``````````There are five auto wiring modes in spring framework. Lets discuss them one by one.

1)no: This option is default for spring framework and it means that autowiring

is OFF. You have to explicitly set the dependencies using tags in bean definitions.

2)byName: This option enables the dependency injection based on bean names.

When autowiring a property in bean, property name is used for searching a

injected in property. If no such bean is found, a error is raised.

3)byType: This option enables the dependency injection based on bean types. When autowiring a property in bean, property’s class type is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised.

constructor: Autowiring by constructor is similar to byType, but applies to constructor arguments. In autowire enabled bean, it will look for class type of constructor arguments, and then do a autowire by type on all constructor arguments. Please note that if there isn’t exactly one bean of the

4)constructor argument type in the container, a fatal error is raised.

5)autodetect: Autowiring by autodetect uses either of two modes

i.e. constructor or byType modes. First it will try to look

for valid constructor with arguments, If found the constructor

mode is chosen. If there is no constructor defined in bean, or

explicit default no-args constructor is present, the autowire byType mode

is chosen.

////////////////////////////////////////////////////////////////////////////////

``````````````````````>RequiredAnnotation

Explain @Required annotation with example?

In a production-scale application, there may be hundreds or

thousands of beans declared in the IoC container, and the

dependencies between them are often very complicated.

One of the shortcomings of setter injection is that it’s very

hard for you to check if all required properties have been set or not

. To overcome this problem, you can set “dependency-check”

attribute of <bean> and set one of four attributes i.e.

none, simple, objects or all (none is default option).

In real life application, you will not be interested in

checking all the bean properties configured in

your context files. Rather you would like to

check if particular set of properties have been

set or not in some specific beans only.

Spring’s dependency checking feature using

“dependency-check” attribute, will not able to help you i

n this case. So solve this problem, you can use @Required annotation.

To Use the @Required annotation over setter method of bean property

in class file as below:

public class EmployeeFactoryBean extends AbstractFactoryBean<Object>

{

private String designation;

public String getDesignation() {

return designation;

}

@Required

public void setDesignation(String designation) {

this.designation = designation;

}

//more code here

}

RequiredAnnotationBeanPostProcessor is a spring bean post processor that

checks if all the bean properties

with the @Required annotation have been set. To enable this bean post

processor for property checking, you must register it in the Spring IoC container.

<bean class="org.springframework.beans.factory.annotation.

RequiredAnnotationBeanPostProcessor" />

If any properties with @Required have not been set, a

BeanInitializationException will be thrown by this bean post processor.

/////////////////////////////////////////////////////////////////////////////////////////////////

-----------> Explain @Qualifier annotation with example?

@Qualifier means, which bean is qualify to autowired on a field. The qualifier annotation helps disambiguate bean references when Spring would otherwise not be able to do so.

See below example, it will autowired a “person” bean into customer’s person property.

public class Customer

{

@Autowired

private Person person;

}

And we have two bean definitions for Person class.

<bean id="customer" class="com.howtodoinjava.common.Customer" />

<bean id="personA" class="com.howtodoinjava.common.Person" >

<property name="name" value="lokesh" />

</bean>

<bean id="personB" class="com.howtodoinjava.common.Person" >

<property name="name" value="alex" />

</bean>

Will Spring know which person bean should autowired? NO. When you run above example, it hits below exception :

Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:

No unique bean of type [com.howtodoinjava.common.Person] is defined:

expected single matching bean but found 2: [personA, personB]

To fix above problem, you need @Quanlifier to tell Spring about which bean should autowired.

public class Customer

{

@Autowired

@Qualifier("personA")

private Person person;

}

///////////////////////////////////////////////////////////////////////////////////////////////////////////////

----------->Setter vs Constructor injection

````````Difference between constructor injection and setter injection?

Please find below the noticeable differences:

In Setter Injection, partial injection of dependencies can possible, means if

we have 3 dependencies like int, string, long, then its not necessary to inject

all values if we use setter injection. If you are not inject it will takes

default values for those primitives. In constructor injection, partial injection

of dependencies is not possible,

because for calling constructor we must pass all the arguments right, if

not so we may get error.

Setter Injection will overrides the constructor injection value, provided if

we write setter and constructor injection for the same property. But, constructor

injection cannot overrides the setter injected values. It’s obvious because

constructors are called to first to create the instance.

Using setter injection you can not guarantee that certain dependency is injected

or not, which means you may have an object with incomplete dependency. On other hand

constructor Injection does not allow you to construct object, until your dependencies are ready.

In constructor injection, if Object A and B are dependent each other i.e A is depends on B and vice-versa, Spring throws ObjectCurrentlyInCreationException while creating objects of A and B because A object cannot be cre ated until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection because Objects are constructed before setter methods invoked.

EXAMPLEOF SEETTER VS CONSTRUCTOR INJECTION

* It consists of the following folders:
* /src/main/java folder, that contains source files for the dynamic content of the application,
* /src/test/java folder contains all source files for unit tests,
* /src/main/resources folder contains configurations files,
* /target folder contains the compiled and packaged deliverables,
* the pom.xml is the project object model (POM) file. The single file that contains all project related configuration.

## 2. Add Spring 3.2.3 dependency

* Locate the “Properties” section at the “Overview” page of the POM editor and perform the following changes:
* Create a new property with name **org.springframework.version** and value **3.2.3.RELEASE**.
* Navigate to the “Dependencies” page of the POM editor and create the following dependencies (you should fill the “GroupId”, “Artifact Id” and “Version” fields of the “Dependency Details” section at that page):
* Group Id : **org.springframework** Artifact Id : **spring-web** Version : **${org.springframework.version}**

Alternatively, you can add the Spring dependencies in Maven’s pom.xml file, by directly editing it at the “Pom.xml” page of the POM editor, as shown below:

*pom.xml:*

|  |  |
| --- | --- |
| 01 | <project xmlns="<http://maven.apache.org/POM/4.0.0>"; xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" |

|  |  |
| --- | --- |
| 02 | xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0><http://maven.apache.org/xsd/maven-4.0.0.xsd>"> |

|  |  |
| --- | --- |
| 03 | <modelVersion>4.0.0</modelVersion> |

|  |  |
| --- | --- |
| 04 | <groupId>com.javacodegeeks.snippets.enterprise</groupId> |

|  |  |
| --- | --- |
| 05 | <artifactId>springexample</artifactId> |

|  |  |
| --- | --- |
| 06 | <version>0.0.1-SNAPSHOT</version> |

|  |  |
| --- | --- |
| 07 |  |

|  |  |
| --- | --- |
| 08 | <dependencies> |

|  |  |
| --- | --- |
| 09 | <dependency> |

|  |  |
| --- | --- |
| 10 | <groupId>org.springframework</groupId> |

|  |  |
| --- | --- |
| 11 | <artifactId>spring-core</artifactId> |

|  |  |
| --- | --- |
| 12 | <version>${spring.version}</version> |

|  |  |
| --- | --- |
| 13 | </dependency> |

|  |  |
| --- | --- |
| 14 | <dependency> |

|  |  |
| --- | --- |
| 15 | <groupId>org.springframework</groupId> |

|  |  |
| --- | --- |
| 16 | <artifactId>spring-context</artifactId> |

|  |  |
| --- | --- |
| 17 | <version>${spring.version}</version> |

|  |  |
| --- | --- |
| 18 | </dependency> |

|  |  |
| --- | --- |
| 19 | </dependencies> |

|  |  |
| --- | --- |
| 20 |  |

|  |  |
| --- | --- |
| 21 | <properties> |

|  |  |
| --- | --- |
| 22 | <spring.version>3.2.3.RELEASE</spring.version> |

|  |  |
| --- | --- |
| 23 | </properties> |

|  |  |
| --- | --- |
| 24 | </project> |

As you can see Maven manages library dependencies declaratively. A local repository is created (by default under {user\_home}/.m2 folder) and all required libraries are downloaded and placed there from public repositories. Furthermore intra – library dependencies are automatically resolved and manipulated.

## 3. Constructor-based dependency injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

### 3.1 Create simple Spring beans

We create a simple Spring bean, HelloWorld and add a dependency to another bean, Foo. In this case, another bean is referenced in the bean, so the type is known, and matching can occur.

*HelloWorld.java:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.snippets.enterprise.services; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class HelloWorld { |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | /\*\* Dependency on Foo class. \*/ |

|  |  |
| --- | --- |
| 06 | private Foo foo; |

|  |  |
| --- | --- |
| 07 |  |

|  |  |
| --- | --- |
| 08 | /\*\* a constructor so that the Spring container can 'inject' a Foo\*/ |

|  |  |
| --- | --- |
| 09 | public HelloWorld(Foo foo){ |

|  |  |
| --- | --- |
| 10 | this.foo = foo; |

|  |  |
| --- | --- |
| 11 | } |

|  |  |
| --- | --- |
| 12 | public String toString(){ |

|  |  |
| --- | --- |
| 13 | return " HelloWorld! foo : \n " + foo; |

|  |  |
| --- | --- |
| 14 | } |

|  |  |
| --- | --- |
| 15 | } |

We also add three new dependencies to Foo bean of simple types and add two new constructors.

*Foo.java:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.snippets.enterprise.services; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class Foo { |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | private String name; |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | private String telephoneNumber; |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | private int age; |

|  |  |
| --- | --- |
| 10 |  |

|  |  |
| --- | --- |
| 11 | public Foo(String name, String telephoneNumber, int age){ |

|  |  |
| --- | --- |
| 12 | this.name = name; |

|  |  |
| --- | --- |
| 13 | this.telephoneNumber = telephoneNumber; |

|  |  |
| --- | --- |
| 14 | this.age = age; |

|  |  |
| --- | --- |
| 15 | } |

|  |  |
| --- | --- |
| 16 |  |

|  |  |
| --- | --- |
| 17 | public Foo(String name, int age, String telephoneNumber){ |

|  |  |
| --- | --- |
| 18 | this.name = name; |

|  |  |
| --- | --- |
| 19 | this.age = age; |

|  |  |
| --- | --- |
| 20 | this.telephoneNumber = telephoneNumber; |

|  |  |
| --- | --- |
| 21 | } |

|  |  |
| --- | --- |
| 22 |  |

|  |  |
| --- | --- |
| 23 | public String toString(){ |

|  |  |
| --- | --- |
| 24 | return " name : " + name+ " \n telephoneNumber : " + telephoneNumber + "\n age : "+age; |

|  |  |
| --- | --- |
| 25 | } |

|  |  |
| --- | --- |
| 26 | } |

### 3.2 Add xml configuration

The configuration xml file is set for the above beans as shown below:

*applicationContext.xml:*

|  |  |
| --- | --- |
| 01 | <beans xmlns="<http://www.springframework.org/schema/beans>"xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"xmlns:p="<http://www.springframework.org/schema/p>"xmlns:aop="<http://www.springframework.org/schema/aop>"xmlns:context="<http://www.springframework.org/schema/context>"xmlns:jee="<http://www.springframework.org/schema/jee>"xmlns:tx="<http://www.springframework.org/schema/tx>"xmlns:task="<http://www.springframework.org/schema/task>" xsi:schemaLocation=" <http://www.springframework.org/schema/aop> <http://www.springframework.org/schema/aop/spring-aop-3.2.xsd> <http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-3.2.xsd><http://www.springframework.org/schema/context><http://www.springframework.org/schema/context/spring-context-3.2.xsd><http://www.springframework.org/schema/jee> <http://www.springframework.org/schema/jee/spring-jee-3.2.xsd> <http://www.springframework.org/schema/tx><http://www.springframework.org/schema/tx/spring-tx-3.2.xsd><http://www.springframework.org/schema/task><http://www.springframework.org/schema/task/spring-task-3.2.xsd>"> |

|  |  |
| --- | --- |
| 02 | <bean id="helloWorldBean" |

|  |  |
| --- | --- |
| 03 | class="com.javacodegeeks.snippets.enterprise.services.HelloWorld"> |

|  |  |
| --- | --- |
| 04 | <constructor-arg ref="fooBean" /> |

|  |  |
| --- | --- |
| 05 | </bean> |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | <bean id="fooBean" class="com.javacodegeeks.snippets.enterprise.services.Foo"> |

|  |  |
| --- | --- |
| 08 | <constructor-arg> |

|  |  |
| --- | --- |
| 09 | <value>fooname</value> |

|  |  |
| --- | --- |
| 10 | </constructor-arg> |

|  |  |
| --- | --- |
| 11 | <constructor-arg> |

|  |  |
| --- | --- |
| 12 | <value>100</value> |

|  |  |
| --- | --- |
| 13 | </constructor-arg> |

|  |  |
| --- | --- |
| 14 | <constructor-arg> |

|  |  |
| --- | --- |
| 15 | <value>25</value> |

|  |  |
| --- | --- |
| 16 | </constructor-arg> |

|  |  |
| --- | --- |
| 17 | </bean> |

|  |  |
| --- | --- |
| 18 | </beans> |

When a simple type is used in the bean, such as int, Spring cannot determine the type of the value, and so cannot match by type without help. For example, in fooBean definition, the values set for the first constructor might be used from the second one as well, since the value 100 can be converted either to String or to int. In order to avoid such type ambiguities, we must always specify the exact data type for constructor, via type attribute.

*applicationContext.xml fooBean:*

|  |  |
| --- | --- |
| 01 | <bean id="fooBean" class="com.javacodegeeks.snippets.enterprise.services.Foo"> |

|  |  |
| --- | --- |
| 02 | <constructor-arg type="java.lang.String"> |

|  |  |
| --- | --- |
| 03 | <value>fooname</value> |

|  |  |
| --- | --- |
| 04 | </constructor-arg> |

|  |  |
| --- | --- |
| 05 | <constructor-arg type="java.lang.String"> |

|  |  |
| --- | --- |
| 06 | <value>100</value> |

|  |  |
| --- | --- |
| 07 | </constructor-arg> |

|  |  |
| --- | --- |
| 08 | <constructor-arg type="int"> |

|  |  |
| --- | --- |
| 09 | <value>25</value> |

|  |  |
| --- | --- |
| 10 | </constructor-arg> |

|  |  |
| --- | --- |
| 11 | </bean> |

### 3.3 Run the application

Through the ApplicationContext the beans are loaded to App.class.

*App.java:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.snippets.enterprise; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import org.springframework.context.ApplicationContext; |

|  |  |
| --- | --- |
| 04 | import org.springframework.context.support.ClassPathXmlApplicationContext; |

|  |  |
| --- | --- |
| 05 |  |

|  |  |
| --- | --- |
| 06 | import com.javacodegeeks.snippets.enterprise.services.HelloWorld; |

|  |  |
| --- | --- |
| 07 |  |

|  |  |
| --- | --- |
| 08 | public class App { |

|  |  |
| --- | --- |
| 09 |  |

|  |  |
| --- | --- |
| 10 | @SuppressWarnings("resource") |

|  |  |
| --- | --- |
| 11 | public static void main(String[] args) { |

|  |  |
| --- | --- |
| 12 |  |

|  |  |
| --- | --- |
| 13 | ApplicationContext context = newClassPathXmlApplicationContext("applicationContext.xml"); |

|  |  |
| --- | --- |
| 14 | HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorldBean"); |

|  |  |
| --- | --- |
| 15 | System.out.println(helloWorld); |

|  |  |
| --- | --- |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |

### 3.4 Output

When you execute the application you should see something like the output presented below:

HelloWorld! foo :   
 name : fooname   
 telephoneNumber : 100  
 age : 25

## 4. Setter-based dependency injection

Setter-based DI is accomplished by the container calling setter methods on the beans.

### 4.1 Create a simple Spring bean

We create a simple Spring bean, Bar and add a dependency to Foo. In this case, the Foo bean is injected via the setter method.

*Bar.java:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.snippets.enterprise.services; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class Bar { |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | private Foo foo; |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | public void setFoo(Foo foo){ |

|  |  |
| --- | --- |
| 08 | this.foo = foo; |

|  |  |
| --- | --- |
| 09 | } |

|  |  |
| --- | --- |
| 10 |  |

|  |  |
| --- | --- |
| 11 | public String toString(){ |

|  |  |
| --- | --- |
| 12 | return "Bar! Foo : \n" + foo; |

|  |  |
| --- | --- |
| 13 | } |

|  |  |
| --- | --- |
| 14 | } |

### 4.2 Add xml configuration

In applicationContext.xml the bean definition must be added.

*applicationContext.xml barBean:*

|  |  |
| --- | --- |
| 1 | <bean id="barBean" class="com.javacodegeeks.snippets.enterprise.services.Bar"> |

|  |  |
| --- | --- |
| 2 | <property name="foo"> |

|  |  |
| --- | --- |
| 3 | <ref bean="fooBean" /> |

|  |  |
| --- | --- |
| 4 | </property> |

|  |  |
| --- | --- |
| 5 | </bean> |

Another way to accomplish the Dependency Injection via the setter is using the @Autowired annotation. Thus, we can get rid of the <property> element in applicationContext.xml. The annotation is added to the setter of the injected bean. When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

*Bar.class:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.snippets.enterprise.services; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import org.springframework.beans.factory.annotation.Autowired; |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | public class Bar { |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | private Foo foo; |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | @Autowired |

|  |  |
| --- | --- |
| 10 | public void setFoo(Foo foo){ |

|  |  |
| --- | --- |
| 11 | this.foo = foo; |

|  |  |
| --- | --- |
| 12 | } |

|  |  |
| --- | --- |
| 13 |  |

|  |  |
| --- | --- |
| 14 | public String toString(){ |

|  |  |
| --- | --- |
| 15 | return "Bar! Foo : \n" + foo; |

|  |  |
| --- | --- |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |

In applicationContext.xml <context:annotation-config/> attribute is defined. It looks for annotations on beans in the same application context it is defined in.

*applicationContext.xml:*

|  |  |
| --- | --- |
| 01 | <beans xmlns="<http://www.springframework.org/schema/beans>" |

|  |  |
| --- | --- |
| 02 | xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"xmlns:p="<http://www.springframework.org/schema/p>" |

|  |  |
| --- | --- |
| 03 | xmlns:aop="<http://www.springframework.org/schema/aop>"xmlns:context="<http://www.springframework.org/schema/context>" |

|  |  |
| --- | --- |
| 04 | xmlns:jee="<http://www.springframework.org/schema/jee>"xmlns:tx="<http://www.springframework.org/schema/tx>" |

|  |  |
| --- | --- |
| 05 | xmlns:task="<http://www.springframework.org/schema/task>"xsi:schemaLocation="<http://www.springframework.org/schema/aop><http://www.springframework.org/schema/aop/spring-aop-3.2.xsd><http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-3.2.xsd><http://www.springframework.org/schema/context><http://www.springframework.org/schema/context/spring-context-3.2.xsd><http://www.springframework.org/schema/jee> <http://www.springframework.org/schema/jee/spring-jee-3.2.xsd> <http://www.springframework.org/schema/tx><http://www.springframework.org/schema/tx/spring-tx-3.2.xsd><http://www.springframework.org/schema/task><http://www.springframework.org/schema/task/spring-task-3.2.xsd>"> |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | <context:annotation-config/> |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | <bean id="helloWorldBean" |

|  |  |
| --- | --- |
| 10 | class="com.javacodegeeks.snippets.enterprise.services.HelloWorld"> |

|  |  |
| --- | --- |
| 11 | <constructor-arg ref="fooBean" /> |

|  |  |
| --- | --- |
| 12 | </bean> |

|  |  |
| --- | --- |
| 13 |  |

|  |  |
| --- | --- |
| 14 | <bean id="fooBean" class="com.javacodegeeks.snippets.enterprise.services.Foo"> |

|  |  |
| --- | --- |
| 15 | <constructor-arg type="java.lang.String"> |

|  |  |
| --- | --- |
| 16 | <value>fooname</value> |

|  |  |
| --- | --- |
| 17 | </constructor-arg> |

|  |  |
| --- | --- |
| 18 | <constructor-arg type="java.lang.String"> |

|  |  |
| --- | --- |
| 19 | <value>100</value> |

|  |  |
| --- | --- |
| 20 | </constructor-arg> |

|  |  |
| --- | --- |
| 21 | <constructor-arg type="int"> |

|  |  |
| --- | --- |
| 22 | <value>25</value> |

|  |  |
| --- | --- |
| 23 | </constructor-arg> |

|  |  |
| --- | --- |
| 24 | </bean> |

|  |  |
| --- | --- |
| 25 | <bean id="barBean" class="com.javacodegeeks.snippets.enterprise.services.Bar"> |

|  |  |
| --- | --- |
| 26 | </bean> |

|  |  |
| --- | --- |
| 27 | </beans> |

### 4.3 Run the application

*App2.class:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.snippets.enterprise; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import org.springframework.context.ApplicationContext; |

|  |  |
| --- | --- |
| 04 | import org.springframework.context.support.ClassPathXmlApplicationContext; |

|  |  |
| --- | --- |
| 05 |  |

|  |  |
| --- | --- |
| 06 | import com.javacodegeeks.snippets.enterprise.services.Bar; |

|  |  |
| --- | --- |
| 07 |  |

|  |  |
| --- | --- |
| 08 | public class App2 { |

|  |  |
| --- | --- |
| 09 |  |

|  |  |
| --- | --- |
| 10 | @SuppressWarnings("resource") |

|  |  |
| --- | --- |
| 11 | public static void main(String[] args) { |

|  |  |
| --- | --- |
| 12 |  |

|  |  |
| --- | --- |
| 13 | ApplicationContext context = newClassPathXmlApplicationContext("applicationContext.xml"); |

|  |  |
| --- | --- |
| 14 | Bar bar = (Bar) context.getBean("barBean"); |

|  |  |
| --- | --- |
| 15 | System.out.println(bar); |

|  |  |
| --- | --- |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |

### 4.4 Output

The output of the setter-based dependency injection example is the one below:

Bar! Foo :   
 name : fooname   
 telephoneNumber : 100  
 age : 25

This was an example of how to use Dependency Injection via a Constructor and a Setter in Spring 3.2.3.

///////////////////////////////////////////////////////////////////////

------------->Difference between FileSystemResource and ClassPathResource?

In FileSystemResource you need to give path of spring-config.xml (Spring Configuration) file relative

to your project or the absolute location of the file.

In ClassPathResource spring looks for the file using ClassPath so

spring-config.xml should be included in classpath. If spring-config.xml

is in “src” so we can give just its name because src is in classpath path by default.

In one sentence, ClassPathResource looks in the class path and FileSystemResource

looks in the file system.

/.////////////////.//////////////////.//////////////////.//////////////////.//

Spring to JAX-RS Cheat Sheet  
This is not an exhaustive list, but it does include the most common annotations.  
  
Spring Annotation JAX-RS Annotation  
@RequestMapping(path = "/troopers" @Path("/troopers")  
@RequestMapping(method = RequestMethod.POST) @POST  
@RequestMapping(method = RequestMethod.GET) @GET  
@RequestMapping(method = RequestMethod.DELETE) @DELETE  
@ResponseBody N/A  
@RequestBody N/A  
@PathVariable("id") @PathParam("id")  
@RequestParam("xyz") @QueryParam('xyz")  
@RequestParam(value="xyz" @FormParam(“xyz”)  
@RequestMapping(produces = {"application/json"}) @Produces("application/json")  
@RequestMapping(consumes = {"application/json"}) @Consumes("application

/////////////////

--------------->Design pattern in spirng framework

```````Name some of the design patterns used in Spring Framework?

There are loads of different design patterns used, but there are a few obvious ones:

Proxy – used heavily in AOP, and remoting.

Singleton – beans defined in spring config files are singletons by default.

Template method – used extensively to deal with boilerplate repeated code e.g.

RestTemplate, JmsTemplate, JpaTemplate.

Front Controller – Spring provides DispatcherServlet to ensure an incoming

request gets dispatched to your controllers.

View Helper – Spring has a number of custom JSP tags, and velocity macros,

to assist in separating code from presentation in views.

Dependency injection – Center to the whole BeanFactory / ApplicationContext

concepts.

Factory pattern – BeanFactory for creating instance of an object.

///////////////////v//////////////////////////////////////////////////////////////////