Resttemplate using spring-boot

Wednesday, May 1, 2024

8:33 PM

Using RestTemplate to consume web services

Spring Boot Module - Rest Template Requests

This page details using RestTemplate to make web requests from a Java/Spring Application. The example code assumes the use of Spring Boot, but RestTemplate is available to all Java/Spring applications.

References

* [RestTemplate](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html)

NOTE: RestTemplate will be deprecated in Spring Framework 5.

RestTemplate

In Spring RestTemplate acts as a web client to make requests to web services. RestTemplate offers 3 types of methods for exchanging data with web service:

|  |  |  |
| --- | --- | --- |
| **API** | **Purpose** | **Example** |
| \*ForEntity | Request for the native exchange format returned by the server (JSON, XML, Text) | getForEntity(...) |
| \*ForObject | Deserialize the response into a Java pojo | getForObject(...) |
| exchange | Generic API for making web requests | exchange(...) |

RequestTemplate Usage

getForEntity

**RestTemplate**rest =**newRestTemplate**();  
**String**resourceUrl ="<http://localhost:8080/myservice/data>";  
**ResponseEntity**<**String**>response =rest.**getForEntity**(resourceUrl +"/1", **String**.**class**);

The above code creates and instance of a RestTemplate and makes a request to the resourceUrl. The method used for the request here is one of the overloadedgetForEntity methods. Here the request is being made for the native data format being sent back by the server. The response is encapsulated in a ResponseEntity object with which we can retrived the response status, body, and headers.

getForObject

**publicclassDataModel**{  
 ...  
 **privateString**value;  
}

**RestTemplate**rest =**newRestTemplate**();  
**String**resourceUrl ="<http://localhost:8080/myservice/data>";  
**DataModel**model =rest.**getForObject**(resourceUrl +"/1", **DataModel**.**class**);

The above code is similar to the previouse example, however this time the code uses getForObject. RestTemplate will attempt to deserialize the response body using a message converter bean based on the Content-Type header in the response.

exchange API

The exchange API offers a more generic manner of make web request through RestTemplate.

**RestTemplate**rest =**newRestTemplate**();  
**HttpEntity**<**Data**>request =**newHttpEntity<>**(**newDataModel**("value"));  
**String**resourceUrl ="<http://localhost:8080/myservice/data>";  
**ResponseEntity**<**DataModel**>response =rest.**exchange**(resourceUrl, **HttpMethod**.POST, request, **DataModel**.**class**);

The above code uses the exchange API to perform an POST request. The major differences in the signature are the HttpMethod argument and the request argument. The HttpMethod argument configures which type of request to make, and the request argument will become the serialized body of the request.

**RestTemplate**rest =**newRestTemplate**();  
**HttpEntity**<**Data**>request =**newHttpEntity<>**(**newDataModel**("value"));  
**String**resourceUrl ="<http://localhost:8080/myservice/data>";  
**ResponseEntity**<**DataModel**>response =rest.**postForLocation**(resourceUrl, request);

This above code is similar to the previous example, but uses postForLocation. REST spec states that a Post request can return an empty body with the Location header set to the new resource url. postForLocation will return the value of the Location header in the response.

Form Data

**RestTemplate**rest =**newRestTemplate**();

**HttpHeaders**headers =**newHttpHeaders**();  
headers.**setContentType**(**MediaType**.APPLICATION\_FORM\_URLENCODED);

**MultiValueMap**<**String**, **String**>map =**newLinkedMultiValueMap<>**();  
map.**add**("value":"test");

**HttpEntity**<**MultiValueMap**<**String**, **String**>>request =**newHttpEntity<>**(map, headers);

**String**resourceUrl ="<http://localhost:8080/myservice/data>";  
**ResponseEntity**<**DataModel**>response =rest.**postForLocation**(resourceUrl, request);

From <[*https://app.revature.com/curriculum/7431/batch/1014/viewCalendar*](https://app.revature.com/curriculum/7431/batch/1014/viewCalendar)>

Class Loaders

Thursday, May 2, 2024

1:29 PM

Class loaders are organized in a hierarchical structure. The class loader hierarchy is responsible for loading classes into the Java Virtual Machine (JVM).

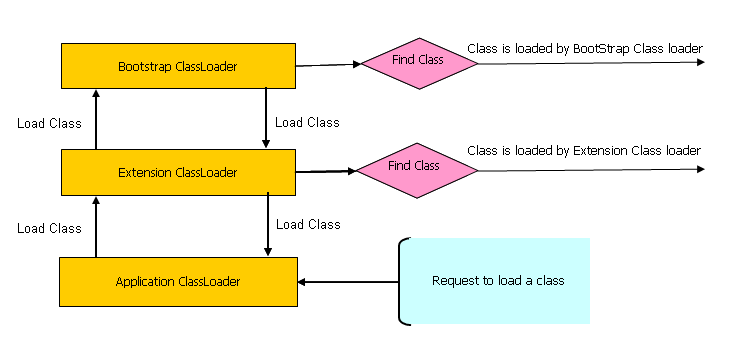


image source:javarevisited.blogspot.com

At the top of the hierarchy is the bootstrap class loader, which is responsible for loading the core Java classes, such as the java.lang package. These classes are typically stored in the JRE’s lib/rt.jar file.

Below the bootstrap class loader is the extension class loader, which loads classes from the JRE’s lib/ext directory or any other directory specified in the java.ext.dirs system property. These classes typically include extensions to the core Java classes.

The third level in the hierarchy is the system class loader, also known as the application class loader. It loads classes from the classpath, which is specified by the CLASSPATH environment variable or the -classpath command-line option.

When the JVM needs to load a class, it will first delegate the request to the topmost class loader in the hierarchy, the **bootstrap class loader**. If the class is not found, the request is passed to the next class loader in the hierarchy, the **extension class loader**. If the class is still not found, the request is passed to the **system class loader**. If the class is not found at any level, a `**ClassNotFoundException`** is thrown.

Custom class loaders can also be added to the hierarchy by extending the ClassLoader class and registering them with the parent class loader.

From <[*https://medium.com/@fullstacktips/what-are-class-loaders-and-different-types-of-class-loaders-in-java-e12f05821bc2*](https://medium.com/@fullstacktips/what-are-class-loaders-and-different-types-of-class-loaders-in-java-e12f05821bc2)>

Spring boot

Monday, May 27, 2024

9:06 PM

can you show me the implementation of the @Autowired, @Qualifier @required, @resource @Inject @Named annonation in java spring

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

Certainly! Each of these annotations serves a specific purpose within the Spring framework. Here's a brief explanation of each along with examples of their usage:

* 1. @Autowired: This annotation is used to automatically wire beans by type. Spring will attempt to find and inject the appropriate bean based on the type of the annotated field, constructor, or method parameter.  
       
     java  
     public class SomeClass { @Autowired private SomeDependency dependency; }

* 1. @Qualifier: When multiple beans of the same type are available, @Qualifier is used along with @Autowired to specify the exact bean to be injected by its name.  
       
     java  
     public class SomeClass {

@Autowired @Qualifier("specificBean")

private SomeDependency dependency;

}

* 1. @Required: This annotation is used to indicate that the annotated property must be populated by Spring's dependency injection mechanism; otherwise, an exception will be thrown at runtime.  
       
     java  
     public class SomeClass { @Autowired @Required private SomeDependency dependency; }

* 1. @Resource: This annotation is used for dependency injection by name. It can be applied to fields, setter methods, and constructors.  
       
     java  
     public class SomeClass { @Resource(name = "specificBean") private SomeDependency dependency; }

* 1. @Inject: Similar to @Autowired, @Inject is used for dependency injection. It's a standard annotation in Java EE, and it can be used interchangeably with @Autowired in Spring applications.  
       
     java  
     public class SomeClass { @Inject private SomeDependency dependency; }

* 1. @Named: This annotation is used to provide a name to a bean. It can be used in conjunction with @Inject for specifying the bean name.  
       
     java  
     Copy code  
     @Named("specificBean") public class SomeDependency { // class implementation }

Remember, the choice of annotation depends on your specific use case and project requirements. While @Autowired is the most commonly used annotation for dependency injection in Spring, the other annotations provide additional functionalities and options for fine-tuning the injection process.

In Spring Boot, @Resource can be used alongside @Qualifier to inject beans by name.

Multithreading & Executor Framework

Tuesday, December 27, 2022

4:16 PM

Thread Scheduler in Java:->

Component of java that decides which thread to run or execute and which thread to wait is called a thread schedular in java.

In java thread is only chosen by a thread scheduler if it is in the runnable state, it is upto the scheduler to pick one of the threads and ignore the other ones.

There are two factors for scheduling a thread : Priority and Time of arrival.

Priority: Priority of each thread lies between 1 to 10. if there is a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

MIN\_PRIORITY is 1 , MAX\_PRIORITY is 10, by default each thread is given NORM\_PRIORITY 5.

Time Of Arrival: If two threads of the same priority enter the runnable state, then priority can't be the factor to pick a thread from these two threads.

In such case, arrival time of thread is considered by the thread scheduler.

Thread that arrived first gets preference over the other threads.

Thread Scheduler works of following algorithms:

1. First come first serve -> it can be understood by name.

First come first serve algorithm is non-preemptive which is bad as it may lead to infinite blocking. To avoid that:->

1. Time slicing scheduling algorithm is used, in which threads are provided some time-slicing after that time another thread will occupy the CPU for that particular time slicing.
2. Preemptive-priority Scheduling: By name it is understood that, this algo is related to the priority of the threads.

There are multiple threads available in the runnable state. The thread scheduler picks the thread with highest priority and provides some time slices to it to avoid starvation, after that time slice if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

If we have multiple threads then thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If the thread is already in the runnable state and the another thread which higher priority than the current thread in runnable state reaches, in that case current thread is pre-empted from the processer, and the thread with higher priority gets the CPU time.

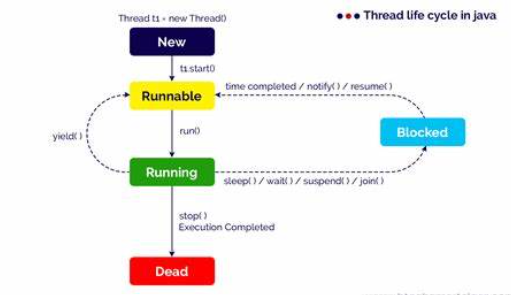
When two threads having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS (first come first serve). Thus the thread that arrives first gets the opportunity to execute first.

Thread Pool Methods

newFixedThreadPool(int s): This method creates a thread pool of the fixed size s.

newCachedThreadPool(): The method creates a new thread pool that creates the threads when needed but will still use the previously created thread whenever they are available to use.

newSingleThreadExecutor(): This method creates a new thread.



Daemon Thread in java:->

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads, when all the user threads dies, jvm terminates this thread automatically.

Daemon threads in Java provide services to user threads primarily in two ways:

1. **Supporting tasks in the background**: Daemon threads are typically used for tasks that can be safely abandoned when the application exits or when all user threads have completed their execution. For example, garbage collection, finalization, and other background tasks in the JVM are often implemented using daemon threads. These threads provide essential services that support the execution of user threads without interfering with the application's termination process.
2. **Resource management and cleanup**: Daemon threads can be responsible for managing and cleaning up resources that are shared among user threads. For instance, a daemon thread might monitor and clean up temporary files, manage connections to external services or databases, or perform periodic maintenance tasks. By running as daemon threads, these tasks can operate in the background without blocking the application's termination, ensuring that resources are properly released and managed even if user threads have completed execution.

In summary, daemon threads in Java play a critical role in providing background services and managing resources for user threads, helping to ensure the efficient and reliable operation of Java applications.

# Join() method:->

It's provided by java.lang package that permits one thread to wait until the other thread to finish it's execution.

If I have a test class and I have made it's thread object-> obj, so using obj.join() will make sure that obj has finished it's execution before the program does the execution of the next statement.

When we use join() method with multiple threads, it leads to overloading on the join() method that permits the developer to maintain the waiting period.

Just like sleep(), join() is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters.

When the join method is called the current thread stops it's execution and goes into the waiting state and remains there until that method with join method is dead or in wait state for the specified time frame is over. If interruption of the thread occurs then InterruptedException is thrown.

**join() method**:

* The join() method is used to wait for a thread to complete its execution before the current thread proceeds further.
* When a thread calls join() on another thread, it effectively waits until that other thread finishes executing or until a specified timeout period elapses.
* This method is particularly useful when one thread needs the results of another thread's computation before continuing its own execution.
* join() is a method of the Thread class in Java.

If system executing the thread has so much load than sleeping time of the thread is more than the time provided In the arguments of the sleep method, if the thread executing system has less load than thread sleeps for the time provided in the argument.

At a time only one thread is executed and if that thread is put to sleep than thread scheduler picks another thread.

After starting a thread it can never be started again. If you does so , an IllegalThreadStateException is thrown

What if we Call java run() method directly instead start() method:->

Each thread starts a separate call stack when we call the run() method using start() method.

Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than starting the new call stack.

**Executor Framework :->**

Database and web servers like server programs execute request from various clients every day and the approach through which we can build new threads whenever a request arrives but for every request ,our server spends to much time and resources, there are too many threads actively running this will result in memory being out of reach so to overcome this problem we have to limit the number of threads actively running.

Solution to resource out of memory is Thread Pool.

Thread Pool :-> it can reuse previously created threads to execute the current thread and offer a solution to problem of thread cycle overhead and resource thrashing.

Java provides Executor interface. It's sub interface is executor service and it's class is thread\_pool\_executor class , class which implements both the interfaces executor and executor\_service interface.

ThreadPoolExecutor class seperates a task creation and it's execution. With threadPoolExecutor, you can implement the runnable objects and send them to an executor. After that it is responsible for there execution, instantiation and running with the necessary threads.

When you send the task to the executor, it tries to use the pooled thread to execute the task to avoid continues spawning(producing) of threads.

Executor Thread Pool method in java:->

newFixedThreadPool(int) -> to generate fixed size thread pool. If the threads are not free then tasks wait in a queue.

newCachedThreadPool(int) -> to generate pool that creates a new thread.

newSingleThreadExecutor() -> when available, creates a single thread.

Interrupt() method :->

Interrupts this thread, causing it to continue if it was blocked for any reason.

dumpStack() method :->

Prints the stack trace for the currently running thread, which is useful when debugging the a multithreaded application.

Synchronization:

When multiple threads try to access the same resource and they can produce unforeseen result due to concurrency( multiple threads problem) issue.

When multiple threads try to write within the same file then they may corrupt the data because one of the threads can override the data or while one thread is trying to open the same file another thread is trying to close the same file at the same time.

Because of this there is a need to synchronize the action of multiple threads and make sure only one thread can access the resource at a given point in time.

This is implemented using the concept of monitors, each object in java is associated with a monitor, which a thread can lock or unlock, only one thread at a time may hold the lock on a monitor.

Inter-thread Communication:->

It's made using these 3 methods

Wait() : Causes the current thread to wait until another thread invokes the notify().

Notify() : wakes up a single thread that is waiting on this object's monitors.

Notifyall() : wakes up all the threads that called wait() on the same object.

Deadlock in java:->

When two types of threads are waiting for each other forever such type of infinite waiting is called deadlock in java.

Synchronized keyword is the only reason for dead lock in java.

So we have to take utmost care while using this keyword.

**How do we avoid dead lock situation:**

1. Using Thread.join() method: we can get dead lock situation when two threads are waiting for each other to finish indefinitely using thread join. Then our thread has to wait for another thread to finish, it is always best to use Thread.join() method with the maximum time you want to wait for the thread to finish.
2. Use lock ordering .
3. Avoiding unnecessary locks : one way to do this is to use a lock free data structure. For example- instead of using synchronized arrayList use the concurrentLinkedQueue.

Deadlock : It describes a situation where two or more threads are blocked forever, waiting for each other

It occurs when multiple threads needs same monitor but obtain it different order.

Concurrent HashMap:->

It is the advanced version of HashMap.

Combination of HashMap and Serialization interface.

Semaphore Class in java :->

It controls access to a shared resource through the use of counter. If the counter is greater than zero, then access is allowed. If it is zero access is denied.

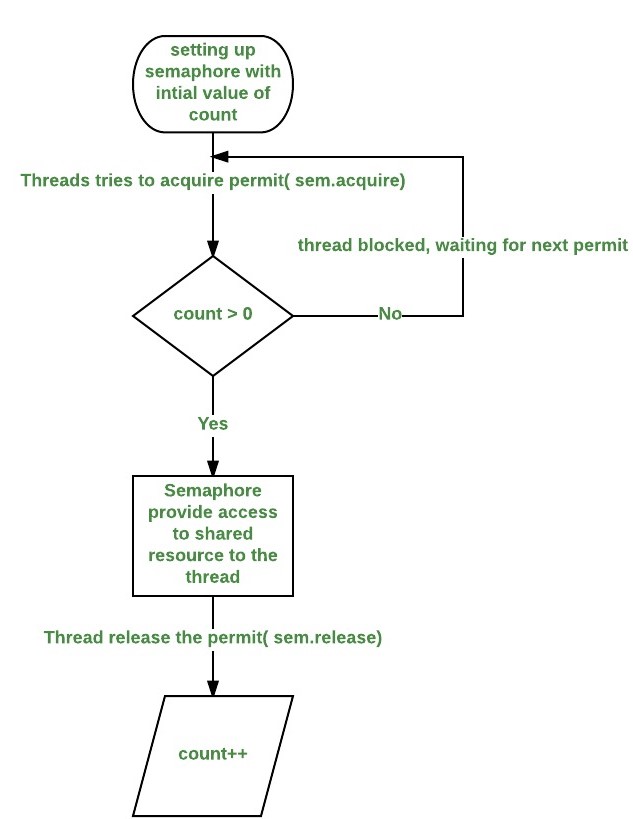
Counter counts the permits that allow access to shared resource, thus to access the resource thread must be granted a permit from the semaphore.

There are two constructors of the semaphore class.

Semaphore(int num) and semaphore(int num, boolean how).

Num specifies the initial permit count, It species the number of threads that can access the resource at any time.

By setting 'how' to true, we can ensure that waiting threads are granted a permit in the order in which they request access.



**Lock vs Synchronized keyword in java :->**

In java both synchronized keyword and lock interface from util.concurrent.lock package are used to handle concurrency and ensure thread safety , but they have some differences in terms of functionality, flexibility and performance.

Synchronized : we can use this to lock and object or method.

* We can use synchronized keyword either in the method declaration or inside the method in the synchronized block.

public synchronized void method() {

// code

}

public void method() {

synchronized (this) {

// code

}

}

* Synchronized keyword locks the monitor locks in an object.
* It is intrinsic and simpler to use for basic locking needs.
* Lock is automatically released when synchronized method or block exists, either normally or by throwing an exception.
* Each object has a single intrinsic lock.
* Synchronized keyword uses wait(), notify() , notifyAll() method for thread communication.

Lock Interface : This interface provides more flexibility with explicit lock and unlock methods.

* Example : using reentrantLock of Lock interface

private final Lock lock = new ReentrantLock();

public void method() {

lock.lock();

try {

// code

} finally {

lock.unlock();

}

}

* Lock provides advanced features like :
  + Ability to locking without blocking (tryLock()).
  + Interruptible locking (lockInterruptibly()).
  + Fair locking (ReentrantLock(boolean fair)).
* With lock you must release the lock manually in the finally block to ensure it gets released even if an exception occurs.
* Locks allow creating multiple conditions object for more complex thread communication.
* Performance Considerations -> lock can offer better performance and scalability in some scenarios due to its flexibility and advance features.

Example :

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class LockExample {

private final Lock lock = new ReentrantLock();

public void method() {

lock.lock();

try {

// critical section

} finally {

lock.unlock();

}

}

public void anotherMethod() {

if (lock.tryLock()) {

try {

// critical section

} finally {

lock.unlock();

public class SynchronizedExample {

public synchronized void method() {

// critical section

}}}

public void anotherMethod() {

synchronized (this) {

// critical section

}}}

Heap Data Structure

Monday, January 29, 2024

4:48 PM

To insert a value in max heap :

public class main{

//delete and insert op in the min heap works in logn time, which is the height of the binary tree

//insert operation in the heap

void insert(int a[], int n ,int value){

int n=n+1; //increasing the array size to input an element in the last

A[n]=value; // putting the value at the last

while(i>1){

int parent=i/2;

if(a[parent]<a[i]){

swap(A,parent,i);

i=parent;

}else{

return ;

}

}

}

//delete operation in the heap

//we will always delete the root node in max/min heap

//remove the root node and put the last node in place of root node compare it with it's children

void delete(int a[], int n){

a[1]=a[n]; //assigning the last element to the root of the tree

n=n-1; //reducing the size of the bst

i=l; // starting to compare the tree from the

while(i<n){

int left=a[2\*i]; //handle the cases here if 2\*i <n -> below also

int right=a[2\*i+1];

int larger=left>right?2\*i:2\*i+1;

if(a[i] <a[larger]){

swap(a,i,larger);

i=larger;

}else{

return ;

}

}

}

}

Definition - A complete binary tree that satisfies a heap's property.

Two types : max heap & min heap

Max heap : parent node value will be larger than it's children node value.

Min heap : value of children nodes will be larger than the parent node value.

Some properties of complete binary tree are below :

For complete binary tree of n nodes, it's height will be log(n)

To know the parent node value of a child node of index i = Math.floor(i/2)

To know the left child of a node at index I = 2\*I

To know the right child of a node at index I = 2\*i+1

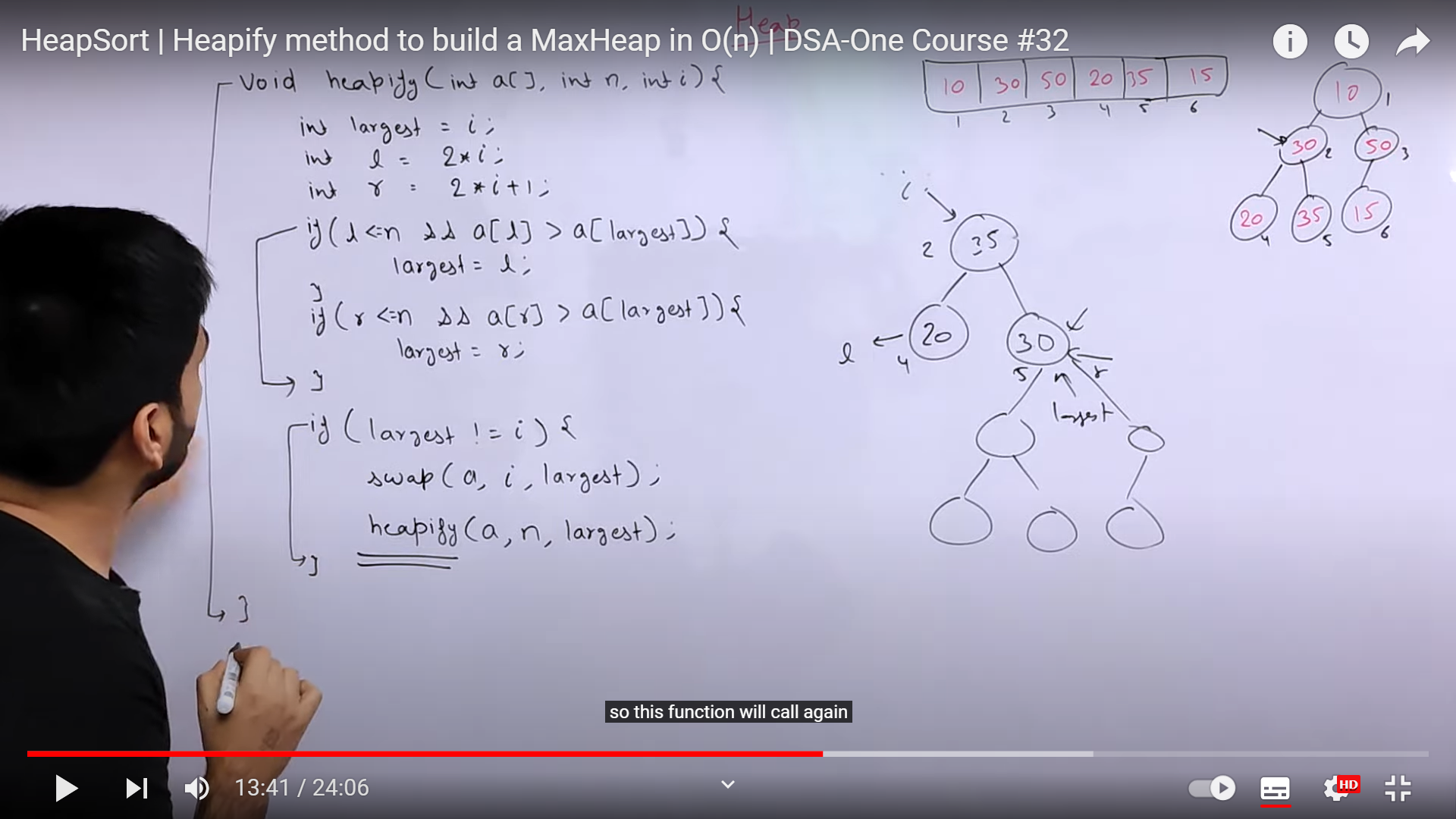
For n nodes there are n/2 leaf nodes

If there are n nodes in a bst than it's leaf node will start from Math.floor[n/2]+1+……n

Node which is just before the first node of the leaf node is -> n/2

Heapify -> converting the current node to heap node.

Heapify only works when the nodes below the node particular to heap is max/min heap tree



Heapify code to insert elements in O(n) time :

Void heapify(int a[], int n, int i){

int largest=I;

int l=2\*n;

int r=2\*n+1;

if(l<=n && a[l]>a[largest]){

largest=l;

}

if(r<=n && a[r]>a[largest]){

largest=r;

}

if(largest!=i){

swap(a,I,largest);

heapify(a,n,largest);

}

}

Public static void buildHeap(int a[],int n){

For(int i=n/2;i>0;i++){

Heapify(a,a.length,n);

}

}

Time complexity is -> O(n)

We can also sort the heap array using the heap sort, all we have to do is :

1. Remove the root element and push it at the back of the array.
2. Now when we delete a node from the max/min heap, we have to include the replace it with the last node and remove the last node from the structure.
3. Once the last node node is inserted at the the root position, call the heapify function such that it arranges the element in the proper order.
4. Again remove the root element and follow the same process from 1 to 3.

Code to sort the element :

Public void sort(int a[], int n){

//here we need to iterate all the nodes so length is n not n/2

For(int i=n;i>0;i++){

Swap(a,1,i); //swaping first and last element that's 1 at the middle

heapify(a,i-1,1); // we are passing the first 1st index because we wanna call the buildHeap function from the first index

}

}

Priority queue

Tuesday, January 30, 2024

1:45 AM

Priority queue implementation in java :

Public class main{

Public void pqueue(){

PriorityQueue pq=new PriorityQueue();//by default it is min heap

Pq.add(12);

Pq.add(23);

Pq.add(2);

Pq.add(323);

System.out.println(pq.peek());//to element(smallest one)

Syso(pq.size());

While(!pq.isEmpty()){

Syso(pq.peek());

}

}

}

To initialize the priority queue as a max heap in java we just need to pass Collections.reverseOrder() inside the construction of the priorityqueue while initializing it.

PriorityQueue pq=new PriorityQueue(Collections.reverseOrder());

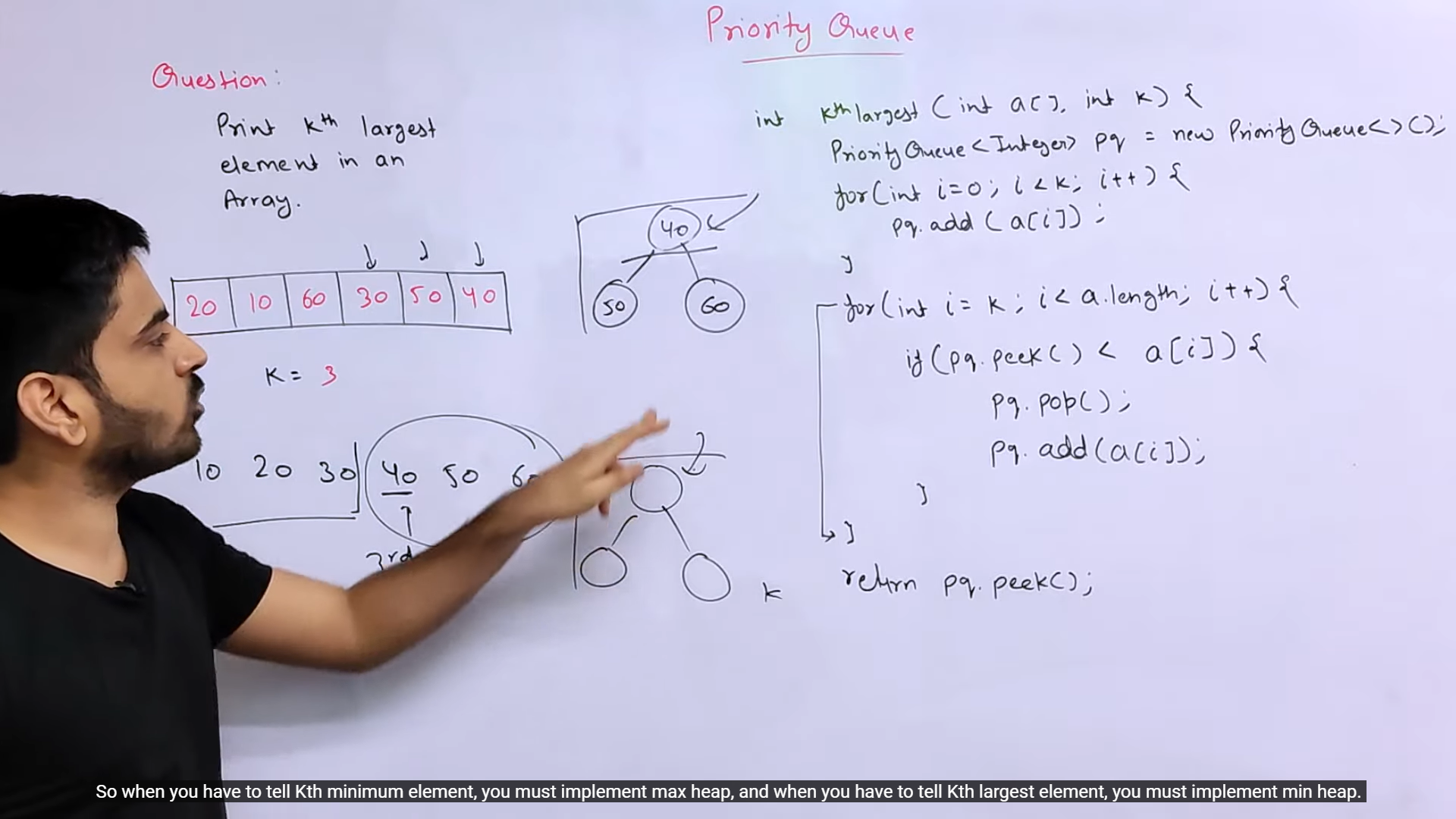
Here elements goes as min heap but comparator is returned as max order.

It works on the heap only, it is basically a queue but elements are inserted and removed on the based of a priority.

We can implement priority queue using a max heap.

in c++ it follows max heap by default when we make priority queue.

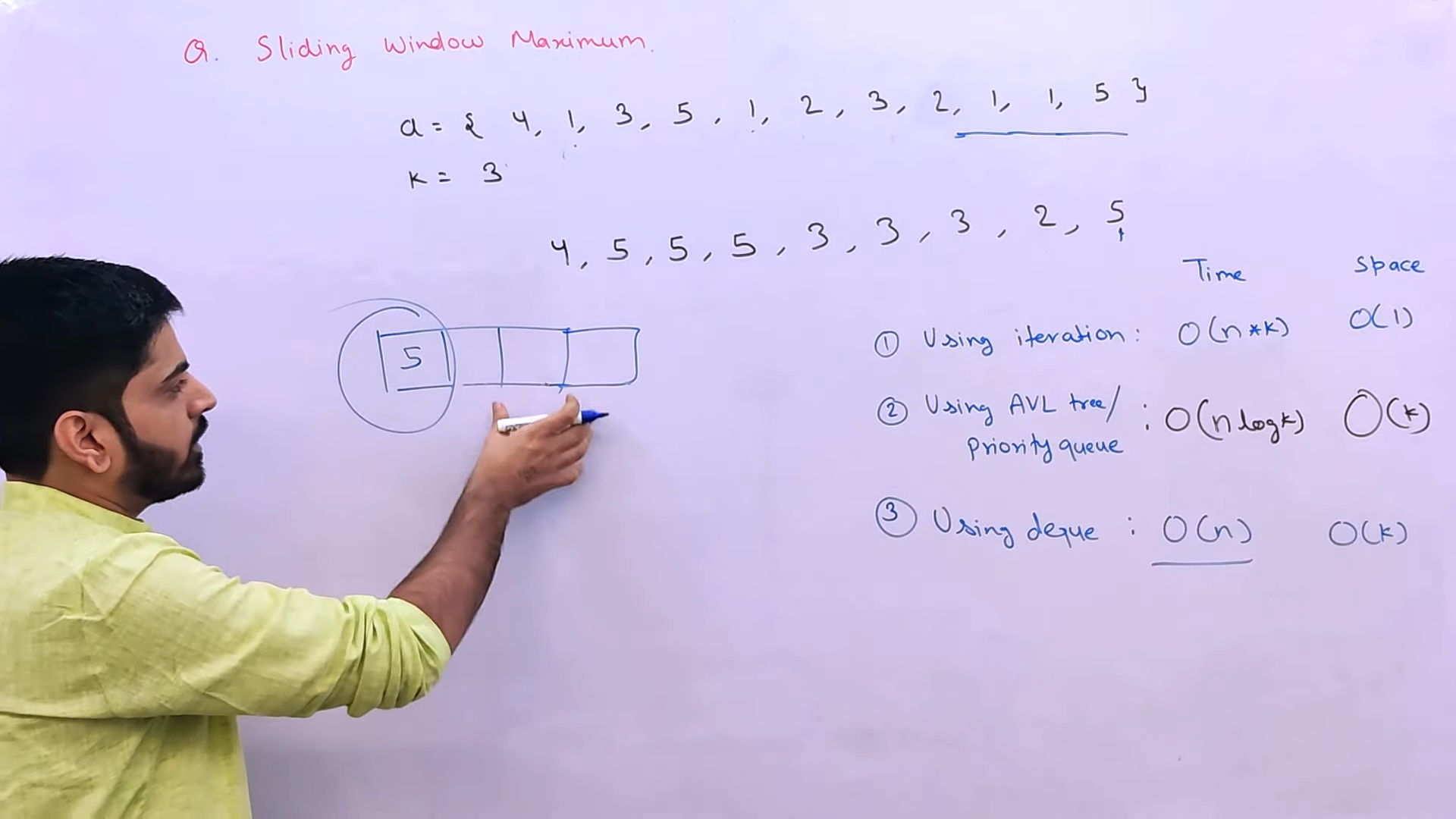
In java if follows min heap by default for priority queue.



Sliding window

Tuesday, January 30, 2024

2:08 AM



Longest nice subarray :

class Solution {

    public int longestNiceSubarray(int[] nums) {

        int head=0,tail=0,result=0,ans=0;

        for(;head<nums.length;head++){

            while((nums[head]&ans)>0){

                ans^=nums[tail];

                tail++;

            }

            ans|=nums[head];

            result=Math.max(result,head-tail+1);

        }

        return result

        ;

    }

}

MAVEN

Mvn compile - uses compiler plugin

Mvn install uses - Apache Maven install plugin

Maven pom is sort of build file and much more.

POM is where project identity and structure are declared, builds are configured and projects are related to one another.

The presence of a pom.xml file indicates that project uses Maven as project management tool.

At this state we use a bare minimum POM to define the project. The elements in pom.xml are :-

Project - the top level or root element of the pom.

* modelVersion - POM model to use and it should be set to 4.0.0
* groupId - projects group id and normally, it is organization name which owns the project such as org.apache, org.springsource etc.,
* artifactId - name of the project such as commons - lang, junit etc,.
* Version - artifact version.

The trio - groupId, artifactId and version- is known as Maven Coordinates. The coordinate provides an unique identity to a maven project and every project should have them.

The coordinates should be unique and no two maven projects can have same coordinates.

**Ellipsis** :-

As we progress through maven, the pom.xml becomes lengthy and it would be difficult to present the complete listing each and every time.

To shorten the listing, we use Ellipses (. . .) in the listing.

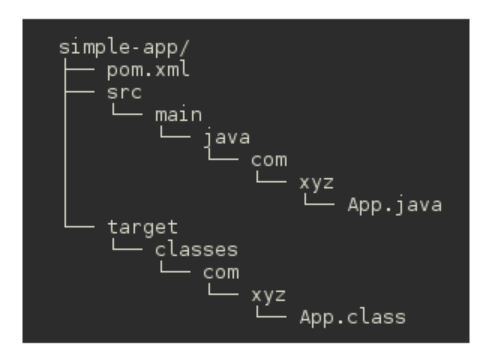


**Build Project :-**

It is used to run maven and also to compile our project.

Maven downloads core plugins during initial run. So we have to enable internet connection, and execute the maven from the top folder of the project where pom.xml exists i.e. from simple-app folder (in the below example our project name is simple-app).

Plugin - it is a software component that adds a specific features to an existing program.



$ cd simple-app

$ mvn compile

Maven downloads some of core plugins and then compile the project. It creates the build directory called target in the top folder & compiled class files goes into it.

Maven installed in simple projects is under 10 mb and reason for such a small core is, Maven delegates the functionality such as compile, test, etc., to plugins. On first run of command mvn compile, Maven downloads Apache Resource Plugin & Apache Compiler Plugin from its central repository and saves it in the local repository and then uses the compiler plugin to compile the project. On subsequent invocation of the command, it finds the compiler plugin in the local repository and uses it to execute the command.

MAVEN CLI :-

The command line interface of maven is "mvn", used to build manage and also interact with maven.

Everything is done through mvn and it's always executed from the top folder of the project that contains the pom.xml file.

$ mvn package : if we are executing this command for the first time on our project, then maven downloads maven jar plugin and some others from central repository and then, it completes the packaging of the project by creating a jar file named simple-app-1.0.jar (with reference to the image above) in target folder.

To check if our build is proper we will run :

$ java -cp target/simple-app-1.0.jar com.xyz.App : it should run the project and give the output of Main class file in java, If our project is properly packaged

$ mvn clean : maven has to delegate the cleaning task to a plugin called Apache Clean Plugin and to do that, it downloads the plugin and then, clean plugin deletes the build directory i.e. target and its subdirectories.

Now we can tryout compile, package again to rebuild the project and this time, we can do so without internet connection as all required plugins are there in the local directory.

RESOURCE ENCODING :-

In the build output, you might have observed a following warning ->

[WARNING] File encoding has not been set, using platform encoding UTF-8,

I.e. build is platform dependent!

To overcome the issue and make build platform independent, we modify pom.xml and set project.build.sourceEncoding property to UTF-8 as shown in the listing:



Maven Repository :-

When we execute any maven commands for the first time, it downloads the core plugins from maven central repository and store or caches them in a local repository. Apart from plugins, Maven also stores dependencies in the local repository.

Ex- if our project depends on Apache Commons Lang and when project is built, Maven downloads artifacts of Commons Lang from the central repository and caches them in the local repository. For windows local repo is located in settings{username}/.m2 folder.

Monday, January 2, 2023

12:54 PM

Apache maven is the project management tool which takes care of following aspects of project management:-

1. Builds the project including preprocessing, compilation, packaging, testing , and distribution.
2. Automates the project dependency management.
3. Generate project reports, documentation and project web site.

Dependency Management :

Maven manages both internal and external dependencies.

For java external dependencies might be libraries such as Junit, Apache Commons, Log4J etc.,

Internal Dependencies are modules of a related project .

Dependencies are added under <dependencies> tag.

Dependencies can contain multiple <dependency> elements.



When we run "mvn compile" command, maven downloads commons-lang version 2.4 from the central repository and save it in local repository and then compiles the project. Next time javac finds the library in build path and compilation goes through without any error.

Still we can't run the app directly using the java command. Java throws NoClassDefFoundError as it is unable to find commons-lang in runtime classpath.

There are multiple ways to handle this :

Brute force method is add commons-lang.jar to classpath. Maven caches the commons-lang.jar in the local repository and we find its location and add the location to the runtime classpath.

$ java -cp target/simple-app-1.0.jar:$HOME/.m2/repository/commons-lang/commons-lang/2.4/commons-lang-2.4.jar com.xyz.App

But, the better and sugested approach is to use maven to run the app.

$ mvn exec:java -Dexec.mainClass ="com.xyz.App"

We use Maven Exec Plugin to run a class

Above command maven-exec-plugin to execute the class com.xyz.App.

Plugin builds the runtime classpath from the dependencies listed in pom.xml, by adding the location of commons-lang-2.4 jar, before executing the app.

UNIT TESTS & JUNIT DEPENDENCY :

In maven java project, the test source files are placed in src/test/java directory.

Main directory is replaced with test directory. Next we will add a file AppTest.java (App.java is main class file).



We use @Test annotation for the test case.

assertEquals(expected,actual) -> here if our expected and actual results are equal than our test will pass otherwise fail.

To run the tests, we will use mvn test command but initially build will fail as maven requires Junit 4 in the build classpath to compile the test files.

We edit the pom.xml and modify <dependencies> Element and add the junit dependency.



Apart from the coordinates of the Junit(groupId, artifactId, version), the dependency contains a new item -scope. It tells maven to add Junit dependency to classpath only during source compile. Scope controls which dependencies are available in which classpath and which are included with an application.

Maven Provides five dependencies scopes :

Compile, provided, runtime, test and system.

To run tests we use mvn test. Doing so maven compiles test files and java source files, run the tests.

Maven uses Maven Surefire Plugin to handle tests and to output the test results.

GET MAVEN DEPENDENCY COORDINATES :

HOW DO WE FIND THE COORDINATES OF LIBRARY OR MODULE WHICH WE WANT TO ADD TO pom.xml.

When an organization deploys the module to maven central repository they define the coordinates and from the central repository we can easily find the coordinates of any library or module.

We can go to the maven central repository and there, type any plugin name and find the groupId ,artifactId and version of it.

Somehow for commons-lang dependency groupId is mentioned as commons-lang whereas it should be apache.org. for some reason, while deploying the module to the repository, developer has mentioned it as commons-lang instead of apache.org and so it is continuing like that.



From version 3 onwards groupId is corrected as apache.org

Lifecycle Phases :

Mvn command structure is : mvn [options] [<goal (s)>] [<phase (s)>]

Apart from options mvn command accepts goal and phase

Maven Life cycle and phases :

To understand the concept of lifecycle and phases , let's recall how we used to build java projects by hand :

Copy resources such as conf file to some folder.

Compile the source file.

Copy dependency jars to build folder.

Run unit tests.

Package the build folder as jar.

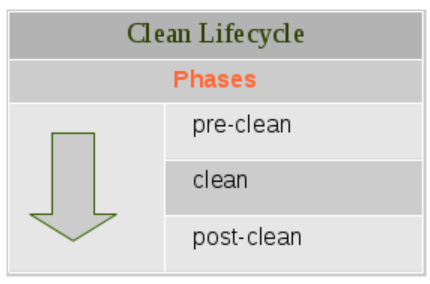
The build task consisted of series of steps which are run one after another in predefined order. Maven call the tasks as lifecycles and steps as phases.

By default maven defines three lifecycles - default, clean and site and life lifecycle consists of predefined phases.

We will understand the concept In detail :-

Clean lifecycle :

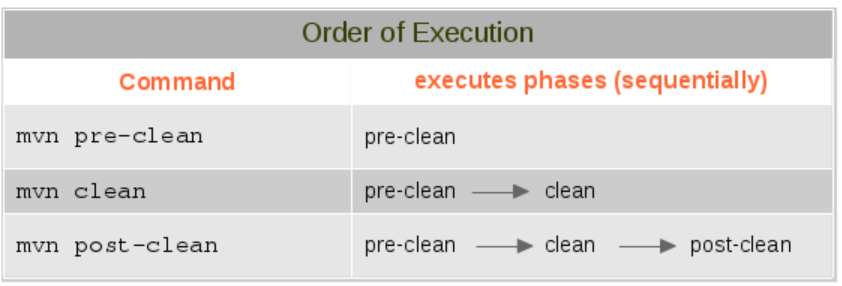
To clean the project we use command **mvn clean** which deletes the build directory named target and its contents. **The clean argument which we pass to mvn command is a phase in lifecycle again named clean.**



The clean lifecycle contains three phases : pre clean, clean, post clean

When we give the command mvn clean, maven loads clean lifecycle and executes all it's phases in the order above.

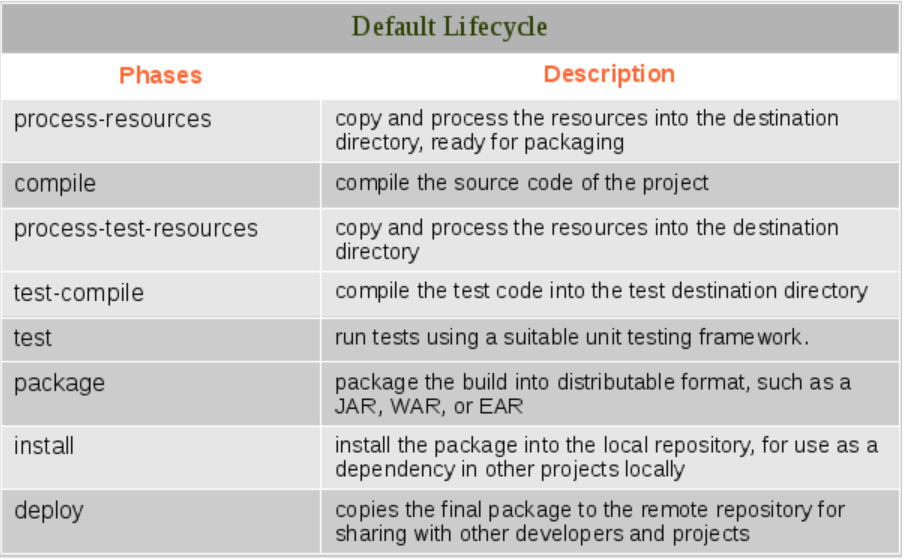
We can also invole any of these phases and all preceding phases up to and including the invoked phase are executed sequentially.



Lifecycle phases can't do anything by themselves. For example , phase clean by itself doesn't have ability or functionality to delete the build directory. It delegates the task to a plugin named **maven-clean-plugin**. So, the phases are just predefined steps which Maven invokes sequentially one after another and actual work is delegated to a plugin.

Default Lifecycle :

The most used of the three lifecycles is the default lifecycle. Maven uses default lifecycle to build, test and distribute the project. Default lifecycle contains 21 phases. Out of this big list, some of the frequently used ones are:



A project may contains resources such as properties, XML configuration files etc., The phases **process-resource** and **process-test-resource** copy and process such resources files.

The phases **compile** and **test-compile** compiles the source java files and test files respectively.

The phases **package**, install and deploy are used to distribute the project. The **package** phase creates JAR files of classes and resource files for distribution.

The phase **install**, installs the artifacts of the project i.e., jar and pom.xml to the local repository at $home/.m2 folder so the other projects can use it them as dependencies.

The phase **deploy** installs the artifacts of the project to a remote repository (probable on internet) so that a wider group of projects can use it as dependency.

Now we lets get back to default phase of maven as focus of this chapter is lifecycle phases.

As we have seen in clean lifecycle, when a phase is invoked using mvn command, all preceding phases up to and including the invoked phase are executed sequentially.

For ex :-

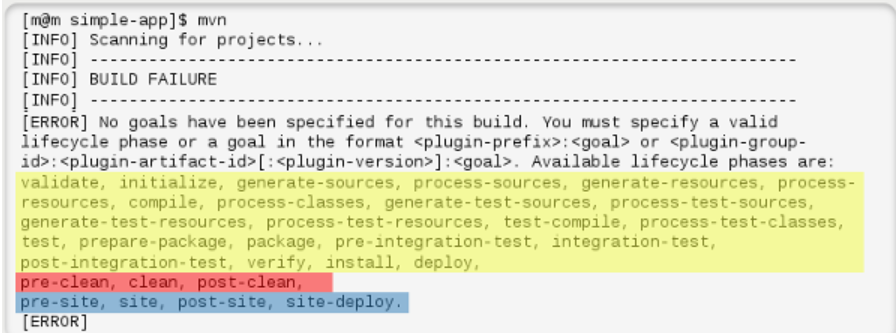
1. Mvn compile - runs phases process-resource and then compile.
2. Mvn test - runs phases process-resource, compile , process-test-resource, test-compile and finally test.
3. Mvn install - runs phases process-resource, compile, process-test-resource, test-compile, test and finally install.

In the clean lifecycle we explained that lifecycle phases by themselves don't have capabilities to accomplish any work. Here too same is true.

For example, compile phase by itself can't do anything but, it delegates compilation job to a plugin named **maven-compiler-plugin**.

**List Of Phases :**

We can always obtain the list of phases by running the following command **mvn** without any argument from your project folder. It triggers build failure and maven shows the complete list of lifecycle phases.



Default lifecycle phases are marked in yellow , clean in red and site in blue.

**Site Lifecycle** :

Site lifecycle generates project documentation and reports about the project. It contains four phases - pre-site, post-site, site-deploy. We can generate a site from a Maven project by running command - **mvn site**

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

The maven command mvn can accept only Lifecycle phase or plugin goal as argument.

Maven comes with three lifecycles - default, clean and site.

Each lifecycle is made up of phases and in all, there are 28 phases, - 21 default , 3 clean, and site4.

When a lifecycle phase is invoked with mvn, all preceding phases up to and including the invoked phases are executed sequentially one after another.

Lifecycle phase by themselves don't have any capabilities to accomplish anything and they rely on plugins to do any work.

MAVEN ARTIFACTS :

In general software terms, an artifact is something produced by the software development process, whether it be an executable file or documentation.

In maven terminology, anything built and deployed by maven to its repository is an artifact.

Some example of maven artifacts :

**Dependencies** - libraries such as commons-io, commons-lang, Junit, Log4j etc.

**Maven plugins** - even the JAR of maven plugins such as maven-clean-plugin, maven-compile-plugin, etc., are also artifacts since they, too, build and deployed by Maven.

**Documentation jars** - javadoc archive of the projects

**Source jars** - source code archive of the projects.

Below screenshot the layout of the local repository that contains artifacts of two projects - Apache commons Exec and Maven Jar Plugin. As we can see each contains a jar (the artifact), POM file and SHA key files.



To use an artifact (either as dependency or plugin) in a maven project, we refer it by its coordinates. The maven coordinates is combination of <groupId> : <artifactId> : <version>.

1. groupId - name of the organization which owns the project and usually a reversed domain name, for example org.example.foo.
2. artifactId - name of the project , library, plugin or a software component.
3. version - a version string.

Maven uses the coordinates to construct the layout repository and also, to locate the artifacts in the repository. For example, the coordinates of commons-exec is org.apache.commons (groupId), commons-exec (artifactId), 1.3 (version) and in the repository, it lands in org/apache/commons/commons-exec/1.3 folder.

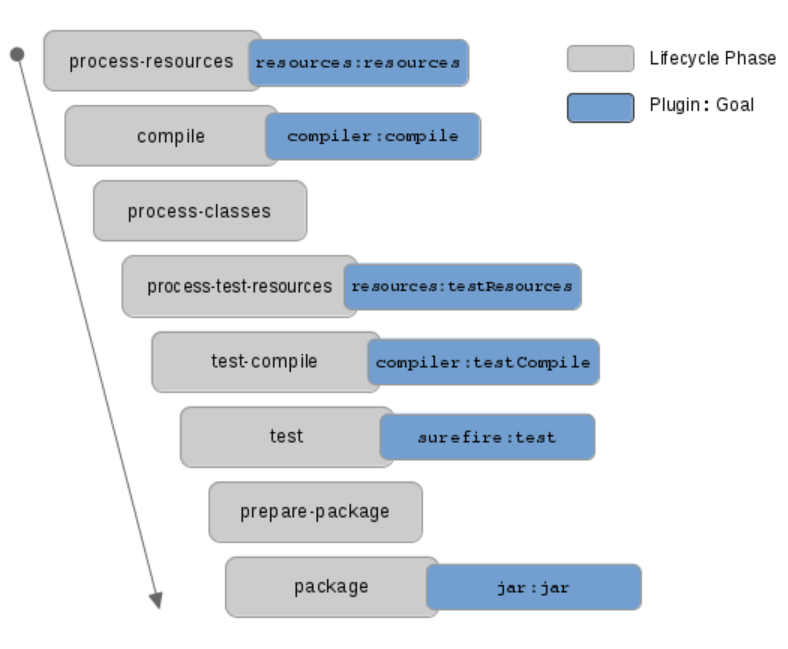
Maven Lifecycle Phases & Goals :

Phases and Plugin Goals :

Lifecycle phases don't come with any functionality, to carry out a task they rely on plugins.

For example: the compile phase doesn't actually compile the java source, instead it delegated the job to compile goal of maven-compile-plugin which invokes the rusty old javac to compile the project.

When a lifecycle phase is executed Maven binds specific plugin goals to lifecycle phases depending on the project and packaging type. For example, when we run **mvn package** in java project, the plugins that are plugged into phases are shown below :



To compile phase, maven binds compile goal of **maven-compiler-plugin**. Similarly, to test phase, it binds **test** goal of **maven-surefire-plugin**.

In a java project maven binds **jar** goal of **maven-jar-plugin**. When we run the same command in a webapp project, maven binds war goal of maven-war-plugin.

Which plugin and goal binds to phases depends on project and also package (jar,war etc.,) type.

The command mvn help:describe -Dcmd=<phase> is a useful command not only to list the lifecycle phases but also to know the bounded goal and plugin version.

**Run the following command :**

$ mvn help:describe -Dcmd=clean

'clean' is a lifecycle with the following phases :

Pre-clean : Not defined

Clean : org.apache.maven.plugins.: maven-clean-plugin:2.5:clean

Post-clean : Not defined

The help : describe load the appropriate lifecycle based on phase, outputs the list of phases that will be executed for that lifecycle. It also shows which plugin: goal is bounded to each phase.

As we can see in the screenshot, no goals are bounded to pre-clean and post-clean phases . For clean phase, clean goal of maven-clean-plugin(plugin version 2.5) is attached.

Now we will be running the same command for deploy phase. As deploy phase is part of default life

cycle, maven loads the lifecycle and outputs its phases and attached plugin:goal. The output list is trimmed for brevity.

**$mvn help:describe -Dcmd=deploy**

It is a part of the lifecycle for the POM packaging 'jar'. This **lifecycle includes** the following phases :

Generate-resource : Not defined

Process-resources: maven-resources-plugin:2.6:resources

Compile : maven-compiler-plugin:3.1:compile

Process-classes : Not defined

Process-test-resources : maven-resources-plugin:2.6:testResources

Test-compile : maven-compile-plugin:3.1: testCompile

Test: maven-surefire-plugin:2.12.4:test

Package: maven-jar-plugin:2.4:jar

Verify: not defined

Install: maven-install-plugin:2.4:install

Deploy: maven-deploy-plugin:2.7:deploy

Invoke plugin goal :

The mvn command accepts, apart form the options, only two things -goal or phases.

We have already seen the usage of phase in commands such as **mvn compile** etc., it is also possible to directly call the plugin goal . Previously we called the mvn help:describe -Dcmd=deploy which is an example of directly calling a goal which executes **describe goal** of **maven-help-plugin**.

While both - calling a phase or goal -ultimately uses some plugins to carry out tasks, there is a subtle difference between them. When a phase is invoked using mvn command, all preceding phases are stepped through sequentially one after another and the attached plugin goals are executed, but when a goal is called only that goal is executed.

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

Maven comes with 3 lifecycles - default, clean and site.

Each lifecycle is made up of lifecycle phases and in all, there are 28 phases - default 21, clean 3 and site 4.

Lifecycle phases by themselves doesn't have any capabilities to accomplish some task and then rely on plugins to carryout the task.

When a lifecycle phase is invoked using mvn command, all preceding phases are executed sequentially on eafter another, but when a goal is called only that goal is executed.

Depending on project and packaging type, maven binds various plugin goals to lifecycle phases. The plugin goal carryout the task delegated to them.

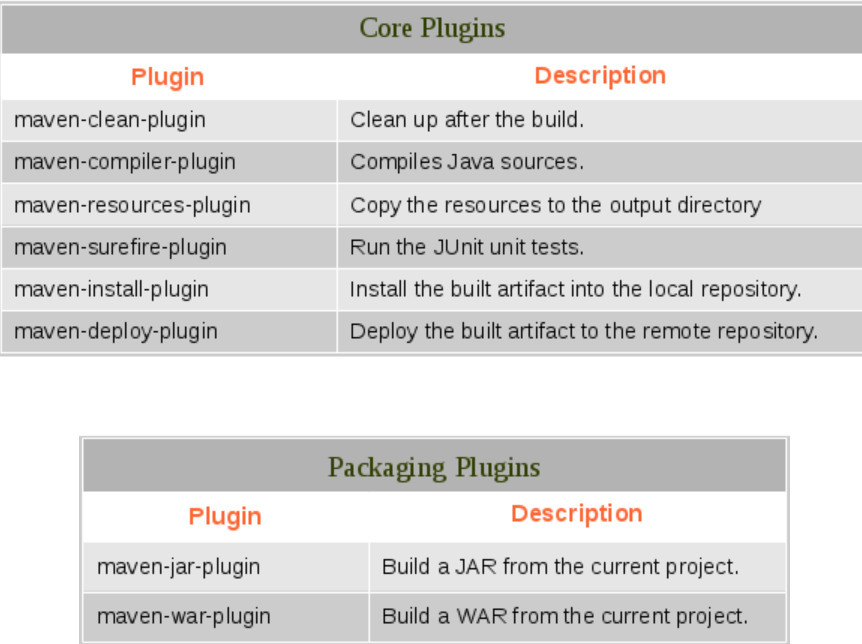
**Maven plugins :**

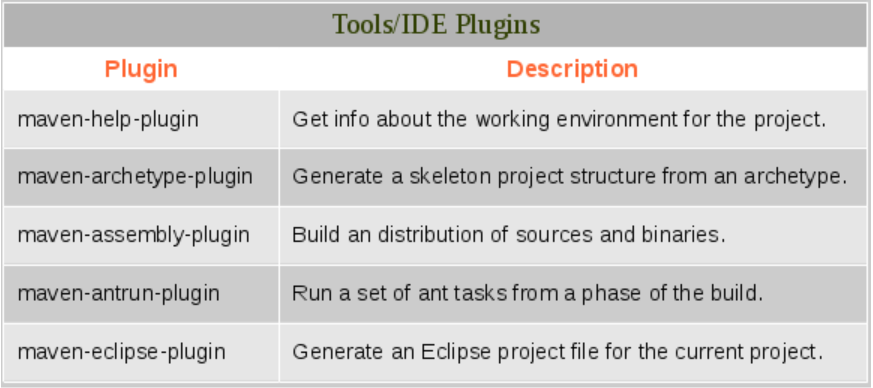
Lifecycle phases are just steps without any capability to carry out some task and they delegate the actual work to maven plugins.

Now we will study, how plugins bind with lifecycle phases and plugin configuration.

The maven core is small and it's just capable of parsing XML files such as pom.xml and manage lifecycle and phases. By itself, maven doesn't know how to compile the code or even to make a jar file. It is designed to delegate most of the functionality to maven plugins. In fact, the first time we ran something like **mvn install** with a new maven installation, it retrieves most of the core maven plugins from the central maven repository. **Maven is - at its heart - a plugin execution framework, all work is done by plugins.**

Plugins are broadly grouped as build plugins and reporting plugins. Apache maven project provides around fifty plugins out of which most frequently used ones are :



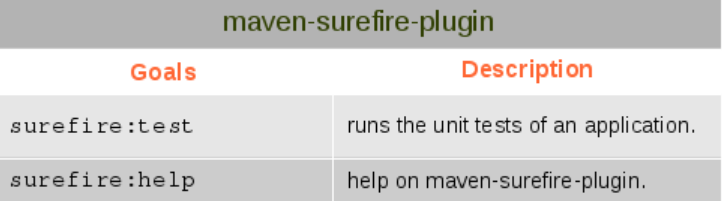


Apart from apache maven project, many projects such as hibernate, tomcat, jetty comes with plugin related to the project

**Maven Plugin Goals:**

Maven plugin is a collection of one or more goals which do some task or job, it is known as MOJO- maven plain old java object.

For example -> maven uses maven-surefire-plugin to run tests and it comes with two goals.



Similarly maven uses maven-compiler-plugin to compile the source code and test files and it provides three goals- compiler:compile, compiler:testCompile, compiler:help.

List plugin Goals :

We can use help goal to list plugin goal. To list goals of maven-compiler-plugin use following command.

$ mvn compile:help

Run plugin goal:

Normally we use lifecycle phase with maven command, but it can also execute plugin goals.

The usage description of maven is:

-> mvn [options] [goal<s>] [phase<s>]

For example, we can directly the source with following command

$ cd simple-app

$ mvn compiler:compile

To use a goal with mvn, use the format <plugin prefix>:goal. In the above example compiler:compile, the compiler is plugin prefix of maven-compiler-plugin and compile is the goal name.

When we invoke a goal directly, maven executes just that goal, whereas when we invoke a lifecycle phase all the phases upto that phase are executed.

For example :-

$ cd simple-app

$ mvn clean

$ mvn surefire:test

Here after cleaning the project, we try to run test with maven-surefire-plugin test goal. It reports that there are no tests to run as surefire test is unable to find any compiled test cases. So, we have to compile source files and test cases before hand.

$ mvn compiler:compile

$ mvn compiler:testCompiler

$ mvn surefire:test

Here we execute goals - compiler:compile and compiler:testCompile to compiler source files and test cases and once class are in place we can run surefire:test and test goes through successfully.

Above example was just to demonstrate the difference between running lifecycle phase and and plugin goal. But in real life we never do that instead we invoke lifecycle phase test as mvn test, which executes all the dependent goals for us explained in maven lifecycle phases.

In some cases on plugin goal is invoked directly , apart from help goal there are some specific plugins whose goals are invoked directly and two such examples are: maven-archetype-plugin and maven-help-plugin

**Maven Plugin Configuration :**

Till now we have used basic pom that has only the project coordinates and explored out of the box behaviour of maven core & plugins, now we cover the plugin configuration :-

To configure the plugins, we use project build element in pom.xml. The listing configures the top elements used to configure a plugin.



**The elements are:**

Build: defines the project build.

Plugins : parent element for one or more <plugin> element.

Plugin: the plugin to configure.

GroupId, artifactId and version : coordinates of the plugin to configure.

Configuration : holds the parameter and properties to be passed to the plugin.

Executions : parent element for one or more <execution> element.

Execution : configures the execution of a goal of the plugin.

**Configures Plugin Parameters:**

With <configuration> we can set plugin parameters. For example, by default maven-compiler-plugin set -source and -target arguments of java compiler to 1.5 . We can use source and target parameters of the compiler plugin to modify this behavior.

Modify the pom.xml of simple App project as shown in the next listing.

Normally the build block is placed after the project coordinates block and before the dependencies block.



The above listing configures the compiler plugin to use 1.6 for source and target. The configuration defined in <plugin> element is applied to all goals of the plugin. In other words, their scope is plugin and maven applies it to both compiler:compile and compiler:testCompile goals.

List Plugin Parameters:

Easiest way to know the available parameters for a goal is to run plugin help goal.

$ mvn compiler:help -Dgoal=compile -Ddetail

It will list the available parameters for compile goal of compiler.

But it will not show the default values of the parameters and to know the available parameters and also, the default value of each parameter, run help:describe goal of maven Help plugin (maven-help-plugin).

$ mvn help:describe -Dplugin=compiler -Dmojo=compile -Ddetail

Note that maven-help-pluign uses -Dmojo for goal instead of -Dgoal

To include or exclude certain files or directories in the project jar, configure maven jar plugin with <include> and <exclude> parameters.



Next one is clean plugin deletes the target directory by default and we may configure it to delete additional directories and files.



The above configuration forces the clean plugin to delete \*.java files in src/main/generated directory, but excludes the template.

For more plugin refer to this link -> <http://maven.apache.org/plugins/index.html>

**Maven Plugin Execution :**

Now we will cover the maven plugin execution configuration.

Configure Plugin execution :

The element <executions> </executions> allow you to configure the execution of a plugin goal, with it you can accomplish the following things :

Bind a plugin goal to a lifecycle phase.

Configure plugin parameters of a specific goal.

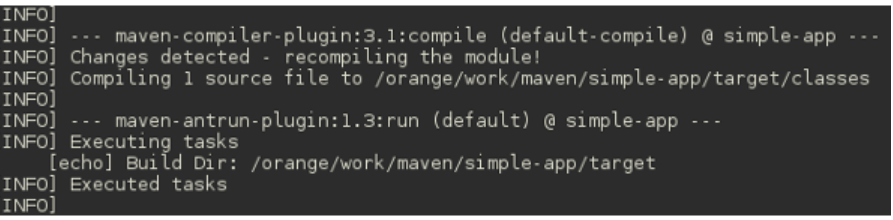
Configure plugin parameters of a specific goal such as compiler:compile, surefire:test etc., that are by default binds to a lifecycle phase.

In apache ant it's easy to output echo any property or message during the build and many of us frequently use it to understand the build flow or debug. But maven comes with no such feature and only way to echo any message is to use Ant within maven. The Apache AntRun plugin provides the ability to run Ant tasks within Maven.

Let's configure maven-antrun-plugin to output the message to console.



Build the project with mvn package and it echos the build directory name in compiler phase.



Lets go through the above snippet to understand the execution. In the <execution> element we indicate that it should be applied to antrun:run goal. We can apply execution to one or more goals. Next we bind execution to compile lifecycle phase and antrun:run goal binds the compile phase. Finally we add <configuration> to the execution. In configuration we set task and echo parameters of antrun:run goal to echo the build directory name.

In the above execution, we attached antrun:run goal to compile phase. But it can be any other phase and during execution of that phase ant will echo the output. Change the phase to test or verify phase and observe the output.

**Maven Properties :-**

In the maven configuration we used maven property ${property.build.directory}. Maven properties are value placeholders, like properties in ant and there values are accessible anywhere within the POM by using the annotation ${x}, where x is the property.

Maven allows five types of properties - Environment variables, Project properties, properties defined in settings files, java system properties and properties defined in POM.

Earlier we saw that we can apply one or more goals to an execution. It is also possible to use one goal in multiple execution. Suppose in the default jar we want to exclude certain packages and create a separate jar with those packages, we can use multiple execution declarations to achieve this.



In execution with id default -jar we exclude some package and then add one or more execution where jar:jar is again binds to package phase and in its configuration we include some package. The classifier parameter of jar:jar appends the string of the jar name. The above snippet, generates two jars -simple-app-1.0.jar and simple-app-1.0-extra.jar

Maven Resource plugin :

This chapter explains the maven Resources plugin which maven uses to handle the project resources.

Resources are non-source files of the project such as properties files, xml configuration files, shell script files and also binary files like images etc., Maven resources plugin (plugin prefix resources) copies resources from the source directories to output directory. Apart from the standard resources:help goal , the plugin has three goals.

Resources:resources :

* Copy resources from the main source codet to the main output directory .
* Always uses the project.build.resources element to specify the resources to copy.

Resources:test-resources :

* Copy resources from the test resources code to the test output directory.
* Always uses the project.build.testResources element to specify the resources to copy.

Resources:copy-resources :

* Copy resources of the configured plugin attribute resources.

The goal resources:resources binds to process -resources phase and resources:testResources to process test-resources.

Process Resources :

By default standard location of the resources is src/main/resoruces and test resources is src/test/resources. Let's add a database configuration file to the simple-app project and see how it is processed by maven. To hold resources, create directories

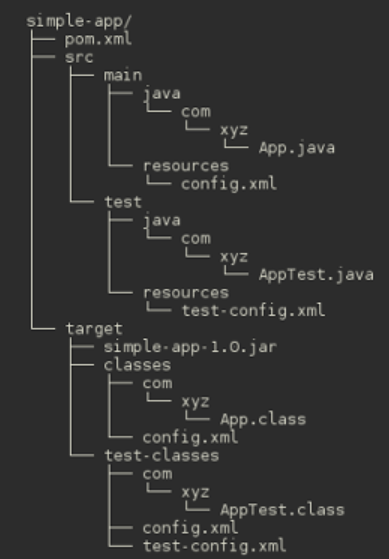
$ cd simple-app

$ mkdir -p src/main/resources

$mkdir -p src/test/resources

Next, add configuration file config.xml to src/main/resources and add test resource file test-config.xml to src/test/resources with following contents :





Finally run, mvn package

The directory structure after the build is shown in the screenshot. Life cycle phase process-resources executes resources:resources goal and it copies src/main/resources/config.xml to output folder target/classes. Similarly lifecycle phase process-test-resources executes resources:testResources goal and it copies src/test/resources/test-config.xml to output folder target/test-classes.

In package phase, the goal jar:jar includes the classes/config.xml in the simple-app-1.0.jar, the test resources are used only to run test resources are used only to run tests and not added to the jar.

Process Resources :

By default standard location of the source resources is src/main/resources and test resources is src/test/resources

**Maven Project Object Model : -**

Overview of POM elements and Structure :

The next listing shows the sections or blocks of maven pom and the top level elements within the project element.





Super Pom :

Maven installation comes with a pom.xml which is known as super POM. As in java where where every other class extends java.lang.Object, all maven projects extends super pom. The pom.xml we create in our project is child or decedent of the super pom.

The super pom declares location of the maven central repository, plugin repository, build element which sets the standard locations of the directories, plugin configuration and executions etc.,

Effective POM :

At the start of every build, maven internally merges project pom.xml with super POM and constructs a new POM which is known as effective Pom.

Earlier in Maven lifecycle and plugin goals, we learned that maven builds plugin goals to lifecycle phases. Actually this magic happens in effective Pom and it is highly instructive to go through effective pom to know what goes on under the hood.

Let's generate the effective pom and analyze it. To get a clear picture ,replace the simple the simple pom.xml with a basic pom, just project coordinates and project dependencies.



Now we will use Help Plugin to dump the effective pom to a file of investigation.

$ cd simple-app

$ mvn help:effective-pom -Doutput=target/efffective-pom.xml

It creates the effective file pom.xml in the target directory. If -Doutput is omitted then command outputs the effective POM to console.

We explained that maven merges project pom.xml with super pom and internally creates an effective pom which it uses to build the project. In the example pom.xml we defined project coordinates, a property related to resource encoding and project dependencies. The effective pom contains these items from pom.xml and also many more items from Super Pom.

The effective POM starts with coordinates of our project and then goes on to define maven central and plugin repositories. Next, in build section, it defines project and build directories.

**<build>**

<sourceDirectory>/home/m/work/simple-app/src/main/java</sourceDirectory>

<scriptSourceDirectory>/home/m/work/simple-app/src/main/scripts</scriptSourceDirectory>

 <testSourceDirectory>/home/m/work/simple-app/src/test/java</testSourceDirectory>

 <outputDirectory>/home/m/work/simple-app/target/classes</outputDirectory>

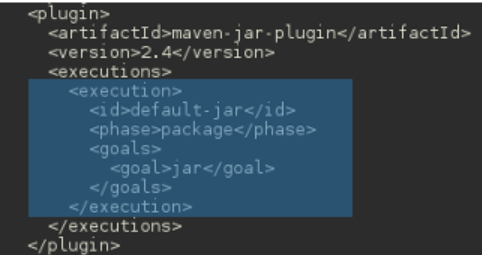
 <testOutputDirectory>/home/m/work/simple-app/target/test-classes</testOutputDirectory>  ....

**<directory>**/home/m/work/simple-app/target**</directory>**

**<finalName>**simple-app-1.0**</finalName>**

From <[*https://www.codetab.org/tutorial/apache-maven/plugins/maven-project-object-model/*](https://www.codetab.org/tutorial/apache-maven/plugins/maven-project-object-model/)>

In the plugin section, Maven builds default goals for lifecycle phases. Though we have not defined build section int the project pom.xml. Maven constructs the effective pom with all the sections which are essential to build the project.



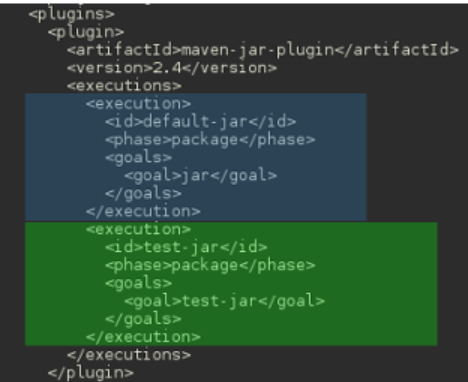
As sample, the next screenshot shows the plugin definition for maven jar plugin. It defines plugin execution with id default-jar (default-<goal-name>) which binds jar:jar goal to the package phase. If we go through the effective pom , we can find other default goals which binds to various lifecycle phases.

**Project POM and Effective POM :**

Lets add some configuration to pom.xml and see how they get merged into effective POM.

So far we saw many single goal executed in a lifecycle phase. However it is possible to execute multiple goals in a single phase. For example, the package phase by default, creates a jar application class and resource files using jar:jar goal. Suppose we also require a separate jar that contains only the test classes and resources. The maven jar plugin has another goal jar:test-jar which creates the jar of the test files and to bind it to package phase, add following snippet to the pom.xml.





Run mvn package and build, in package phase execute two goals , jar:jar and jar:test-jar and creates two jars -simple-app-1.0.jar and simple-app-1.0-test.jar - in target directory.

Generate effective pom with mvn help:effective-pom and in it, for maven jar plugin we can find two executions - default one defined by super pom another one defined by us In pom.xml

Validation In Spring

Monday, June 3, 2024

11:16 PM

@ModelAttribute -> bind the method parameter or return type to a pojo class. Binds the request paramter to a model object.

It adds the object to the model so it can be accessed in the view.

@RequestBody -> it binds the request body to a model object.

* It does not adds the object to the model. Not intended for view rendering.

They both Binding and validation using @valid annonation.

@Valid -> this annonation is used to validate the input against all the constraints.

DataBinder -> this class can be used with the validation such that all the validation error can be stored in this object.

Errors -> errors objects can be passed such that validators can report validation failures to this errors object.

1. We can also build our custom validator by implementing the Validator interface.
2. Doing so, we will need to override the method public boolean supports(class) and public void validate(Object, Errors) method.
3. We can use ValidatorUtils.rejectIfEmpty() or ValidatorUtils.rejectIfEmptyOrWhitespace() method if there are any errors found and these errors can be registered in the Errors object passed.

**Spring Errors :-**

Spring errors object interface stores and exposes information about data binding and validation errors for objects. This interface defines a static NESTED\_PATH\_SEPERATOR String, which details the nested path for properties on a model or bean.

* Example-> if there is an object of account type it's attribute will be represented by account.id or if it has a parameter which in turn is a class so it will be represented like account.paramterName.attributeName.

Hibernate also provides it's own validation like @NotNull @Size @Max @Min etc.

Transaction in spring

Monday, June 3, 2024

11:45 PM

@Transactional annonation is used to handle transaction and ensure acid properties.

It is best practice to use this annonation with @Service annonation.

Transaction is a unit of work so it can cause either success or failure, The side effect of transaction can be assumed to either commit or rollback therefore the model will be consistent, but the @Service will be required to react accordingly.

There are many attributes which can be passed with the transaction annonation like :

Isolation -> isolation level of transaction.

noRollbackFor -> indicates which exception types must not cause a transaction rollback.

noRollbackForClassName -> indicates which exception types must not cause a transaction rollback.

Propagation -> transaction propagation type.

readOnly-> setting this flag to true will only tell transaction to perform read-only operations only.

rollbackFor -> indicates which excpetion types must cause a transaction rollback.

rollbackForClassName -> indicates which exception types must cause a transaction rollback.

Timeout -> timeout for this transaction.

transactionManager -> qualifier value for the specified transaction.

We can also use this annonation in the method, overriding the default implementation.

Transation Propagation in spring =>

1. Mandatory -> support the current transaction, throw an exception if none exists.
2. Nested -> supports the current as well as nested transaction.
3. Required -> if no transaction exists, create a new one.
4. Not Supported -> execute non-transactionally, suspend the current transaction.
5. Never -> execute not-transactionally, throw an exception if a transaction exists.
6. Requires New -> create a new transaction, and suspend the current transaction if one exists.
7. Supports -> supports the current transaction, execute non-transactionally if none exists.

In Spring, the @Transactional annotation is used to manage transactions declaratively. The propagation attribute of the @Transactional annotation determines how the transactional context should behave when a transactional method is invoked. Here are examples for each of the transaction propagation types in Java:

1. REQUIRED

If there is an existing transaction, join it. If not, create a new one.

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class RequiredService { @Transactional(propagation = Propagation.REQUIRED) public void requiredMethod() { // Your transactional code here } }

2. REQUIRES\_NEW

Create a new transaction, suspending the current transaction if one exists.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class RequiresNewService { @Transactional(propagation = Propagation.REQUIRES\_NEW) public void requiresNewMethod() { // Your transactional code here } }

3. NESTED

Execute within a nested transaction if a current transaction exists, otherwise behave like REQUIRED.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class NestedService { @Transactional(propagation = Propagation.NESTED) public void nestedMethod() { // Your transactional code here } }

4. MANDATORY

Support a current transaction, throw an exception if none exists.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class MandatoryService { @Transactional(propagation = Propagation.MANDATORY) public void mandatoryMethod() { // Your transactional code here } }

5. SUPPORTS

Support a current transaction, execute non-transactionally if none exists.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class SupportsService { @Transactional(propagation = Propagation.SUPPORTS) public void supportsMethod() { // Your transactional code here } }

6. NOT\_SUPPORTED

Execute non-transactionally, suspend the current transaction if one exists.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class NotSupportedService { @Transactional(propagation = Propagation.NOT\_SUPPORTED) public void notSupportedMethod() { // Your transactional code here } }

7. NEVER

Execute non-transactionally, throw an exception if a transaction exists.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class NeverService { @Transactional(propagation = Propagation.NEVER) public void neverMethod() { // Your transactional code here } }

8. NEVER

Always execute non-transactionally.

java

Copy code

import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @Service public class NeverService { @Transactional(propagation = Propagation.NEVER) public void neverMethod() { // Your transactional code here } }

How to Use @Transactional

* 1. Annotating Service Methods:  
     Typically, the @Transactional annotation is used on service methods where transactional behavior is desired.
  2. Setting Propagation and Other Attributes:  
     You can specify other attributes such as isolation, timeout, readOnly, and rollbackFor alongside propagation.  
       
     java  
     Copy code  
     @Service public class SampleService { @Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ\_COMMITTED, timeout = 30, readOnly = false, rollbackFor = Exception.class) public void transactionalMethod() { // Your transactional code here } }
  3. Combining Propagation Types:  
     In a real-world application, you might have different methods with different propagation behaviors interacting with each other.  
       
     java  
     Copy code  
     @Service public class CombinedService { private final RequiredService requiredService; private final RequiresNewService requiresNewService; public CombinedService(RequiredService requiredService, RequiresNewService requiresNewService) { this.requiredService = requiredService; this.requiresNewService = requiresNewService; } @Transactional(propagation = Propagation.REQUIRED) public void combinedMethod() { requiredService.requiredMethod(); requiresNewService.requiresNewMethod(); } }

Summary

By properly configuring the @Transactional annotation and understanding the different propagation types, you can effectively manage transactions in your Spring Boot application to ensure data consistency and integrity.

From <[*https://chatgpt.com/c/4cb2fba9-b5e2-4b70-8df7-a39479ad59ab*](https://chatgpt.com/c/4cb2fba9-b5e2-4b70-8df7-a39479ad59ab)>

Spring Security

Tuesday, June 4, 2024

12:41 AM

@EnableSecurityConfig -> this will tell the spring, it is security config class.

Overriding the default securityfilterchain method and removing the @Order annonation will set the priority of the method more than the default one.

User defined filter ->

* Http.authorizedRequests().anyRequest().authenticated().and().addFilterBefore(jwtAuthFilter,UsernamePasswordAuthenticationFilter.class);
* Here we have removed the default http login form and provided the custom filter.
* UsernamePasswordAuthenticationFilter will be executed before build in filter.
* + **SecurityContextPersistenceFilter:** Stores the SecurityContext contents between HTTP requests. It also clears SecurityContextHolder when a request is finished.
  + **ConcurrentSessionFilter:**It is responsible for handling concurrent sessions. Its purpose is to refresh the last modified time of the request's session and to ensure the session hasn't expired.
  + **UsernamePasswordAuthenticationFilter:** It's the most popular authentication filter and is the one that's most often customized.
  + **ExceptionTranslationFilter:**This filter resides above FilterSecurityInterceptor in the security filter stack. Although it doesn't perform actual security enforcement, it handles exceptions thrown by the security interceptors and returns valid and suitable HTTP responses.
  + **FilterSecurityInterceptor:**It is responsible for securing HTTP resources (web URIs), and raising or throwing authentication and authorization exceptions when access is denied.

From <[*https://www.interviewbit.com/spring-security-interview-questions/*](https://www.interviewbit.com/spring-security-interview-questions/)>

|  |  |
| --- | --- |
| -> | Whenever a request arrives it's filtered by the JwtutilFilter, which checks if Bearer of request has any token & if it has they will fetch the user data from user Service. |
| -> | Token is of 7 chars. |
| -> | If token is valid the filter creates an usernamePasswordAuthenticationToken. |
| -> | UsernamePasswordAuthenticationToken stores the username and password provided by the user. |
| -> | First thing that gets executed when we create a spring boot application is securityFilterChain. |
| -> | JwtAuthentication filter is Once per request filter and checks everything, it has a role to check everything regarding the jwt token that we have. |

JWT Token Filter Steps :-

* At first jwt will check if any token is present or not, if not return 403 to user.
* If Jwt token is present, we'll start the validation process.
* JWT auth filter will fetch the user data from user service like H2/sql db.
* If all the checks are passed, we start the JWT validation process, jwt token was based on the specific user so we will validate it.
* Jwt validity service will take the user object and token and check the validity of the token like expiration time or validity. If token is not valid return 403 to the client.
* We store the user info like user name and password with it's token inorder to validate it in future.
* We will update the security context holder and set this user as connected, we will tell our user filter chain that this user is authenticated for request it's authentication will be approved.
* Once the security context holder will be updated, our request will be sent to the dispatcher servlet.

SecurityFilterChain is an interface and securityFilterChainConfiguration class has default implementation of defualtSecurityFilterChain() method which accepts HttpSecurity argument & returns an object of SecurityFilterChain.

* By default this line executed -: http.authorizeRequests().anyRequest().authenticated()

//this line tells the application to authenticate all the request.

* Http.formLogin() basic login form shown in browser just like the basic dialouge box prompting for the username and password.

Serialization & Deserialization in java

Friday, June 7, 2024

4:54 PM

BufferedReader :

* Buffered reader is primarily used for reading character data from an input source, such as file or an input stream.
* Buffering : buffering functionality which means it reads data from the underlying input source in chunks and stores it in an internal buffer. This improves performance by reducing the number of I/O operations.
* Buffer reader reads the data in larger chunks, it is more efficient than reading the data char by char directly from input source.
* It also provides convenient methods like readLine() for reading data line-by-line from a file or input stream.
* BufferedReader is sub-class of an abstract class reader.

FileInputStream :

* Used for reading raw byte data from a file on disk. It is specifically designed for reading character data.
* No buffering functionality.
* It is low-level input stream, provide methods like to read() single bytes from the input stream.
* It is subclass of InputStream which is an abstract class representing a stream of bytes.

Basically BufferedReader is used for efficient character-based input, especially for reading text files, while "FileInputStream" is used for low-level byte-based input, suitable for reading binary data from files.

ObjectInputStream :

* Object input stream establishes an association with the provided "INPUTSTREAM", this means that, it will read data from the input stream to deserialize objects.
* It first reads the special header from the input stream to verify that the stream contains serialized objects. This header typically contains meta data about the serialized data format.
* Once verified, it starts deserializing objects from the input stream. It reads bytes representing the serialized object's state from the input stream & reconstructs the object in memory.
* **Class Descriptor**: In Java serialization, each serialized object includes a class descriptor that contains metadata about the class structure. This metadata includes information such as the class name, superclass name, field names, and field types.
* It reads class descriptors from the input stream and uses reflection to locate & load the corresponding class definition.
* After resolving the class, it instantiates the object by invoking it's constructor and initializes its fields by reading data from the input stream.

If we do not use ObjectInputStream it reads the raw data and will not deserialize it into java objects, we will need to parse it.

It refers to a process of converting object into sequence of bytes which can then be saved to a file, sent over a network or saved to a database.

The reverse process where bytes are converted back into objects is called deserialization.

Any object that implements SERIALIZATION interface can be serialized. This interface works as a marker which tells java runtime that objects state can be written to a stream.

This interface is just a marker with no methods just to tell that this class can be converted to stream.

When object is serialized all the data along with metadata is converted into a byte stream. Metadata is included because it is used while deserializing the object for reconstructing.

Transient fields are not serialized.

Externalizable interface -> we can use this to have more control during serialization & deserialization process by allowing you to define custom read and write methods.

-> java provides build in classes for serializing and deserializing objects.

1. ObjectOutputStream.
2. ObjectInputStream.

Example :

import java.io.\*;

public class SerializationExample {

public static void main(String[] args) {

// Create an object to serialize

Student student = new Student("John", 25);

// Serialize the object to a file

try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("student.ser"))) {

out.writeObject(student);

System.out.println("Object serialized successfully.");

} catch (IOException e) {

e.printStackTrace();

}

// Deserialize the object from the file

try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("student.ser"))) {

Student deserializedStudent = (Student) in.readObject();

System.out.println("Object deserialized successfully: " + deserializedStudent);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

// A simple Serializable class

class Student implements Serializable {

private String name;

private int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Student{name='" + name + "', age=" + age + '}';

}

}

Bean Factory & Application Context

Saturday, June 8, 2024

8:49 PM

Spring is an umbrella term for family of frameworks which can be utilized to rapidly create loosely coupled java applications. These spring frameworks provide a comprehensive & configurable model for modern enterprise java applications. Which can easily migrated to any kind of deployment platform.

This ease is due to the core feature of spring framework, which focuses on dependency injection resulting from the framework acting as IOC container.

Spring utilizes pojo design pattern & applying enterprises services, as needed, non-invasively to those POJOs.

This loose coupling allows developers to focus on business logic for applications as spring handles the infrastructural needs.

Spring consists of close to 20 modules, each focusing on particular task or service.

These modules are grouped into the following layers:

1. Data Access/Integration :
   1. JDBC
   2. ORM
   3. OXM
   4. JMS
   5. Transactions
2. Web:
   1. Socket
   2. Servlet
   3. Web
   4. Portlet
3. AOP
4. Aspects
5. Instrumentation
6. Messaging
7. Core Container:
   1. Core
   2. Beans
   3. Spel
   4. Context
8. Test

It is worth noting that bean factory instantiates bean lazily and application context does it eagerly.

In spring IOC containers are responsible for Instantiating,configuring and assembly of objects knows as beans. It does this by getting information from the xml files & assembling the objects accordingly.

In spring there are two types of IOC containers :

* Bean Factory and application context, application context is build out of the bean factory which provides extra functionality, such as integration with spring AOP, event propagation, message resource handling & application layer specific context.
* Dependency Injection is a design pattern that removes dependencies of a program providing the configuration in a external source such as ML file.
* DI can occur through various methods: constructor, Field injection and setter injection.
* To use constructor injection in xml configuration we use the <constructor-arg> tag. And for the use of field injection we use <property tag>
* To use setter injection using xml configuration file we can make use of <property> tag.
* To set beans in annonation configuration we use @autowired annonation in case of setter as well as constructor injection.
* Same we do in java annonation.

BeanFactory interface is used for instantiation, configuration and assembly of bean and also for the eventual removal. BeanFactory provides the use of xml

Stack/Heap Memory

Monday, June 10, 2024

5:01 PM

In Java, both the stack and the heap are memory areas used for different purposes:

1. **Stack**:
   * **The stack is a region of memory that stores method invocations and local variables.**
   * **For each thread in a Java program, a separate stack is allocated.**
   * **When a method is called, a new frame is pushed onto the stack to store parameters, local variables, and intermediate results.**
   * **As method calls return, their frames are popped off the stack, and the memory is freed.**
   * **The stack follows a last-in, first-out (LIFO) structure.**
2. **Heap**:
   * **The heap is a region of memory that stores objects and arrays dynamically allocated by the Java Virtual Machine (JVM).**
   * **It's a shared resource accessible to all threads in a Java application.**
   * **Objects in the heap can be referenced by multiple parts of the program and persist beyond the lifetime of any individual method.**
   * **Memory in the heap is managed by the JVM's garbage collector, which automatically deallocates memory for objects that are no longer referenced.**
   * **The heap does not have a defined organization like the stack; memory is allocated and deallocated dynamically.**

Memory Management in java :

Heap Memory :

* Dynamic Memory Allocation : the heap is a region of memory where objects are dynamically allocated during runtime. It's large, shared pool of memory that all java threads have access to.
* Garbage Collection : Objects created on heap remain untill they are no longer referenced by any part of the program. Java GC scans for the heap to reclaim memory occupied by objects that are no longer in use.
* Storage for Objects : Most of the objects created in java programs, including those created with the new keyword are stored in the heap.
* Size management : The size of the heap can be controller using jvm command line options such as -Xms (initial heap size ) and -Xmx (maximum heap size).

Pool :

* The term "pool" in Java often refers to various specialized collections or resources that are managed in a pooled manner. For example, connection pools in database access frameworks or thread pools in concurrency utilities.
* These pools typically allocate resources (such as database connections or threads) from a pre-allocated pool, rather than dynamically creating and destroying them. This can improve performance by reducing overhead associated with frequent allocation and deallocation of resources.

Stack Memory :

* Stack is region of memory used for static memory allocation, primarily for storing method call frames & local variables.
* LIFO structure.
* Limited Size : it is relatively small compared to heap and is typically limited in size. Each thread in java has its own stack memory.

HashMap Working

Tuesday, June 11, 2024

10:20 AM

How HashMap works :

1. Internal Structure : it is implemented as a hash table, consisting of an array of buckets. Each bucket can store multiple entries (key-value pairs).
2. Hashing Function : when you insert a key-value pair into a hashmap, it computes the hash code of the key. The hash code is used to determine the index of the bucket in which the entry will be stored.
3. Handling collisions : since different keys may have same hash code (collision), HashMap uses a linked list or a binary tree( starting from java 8) to handle collisions within the same bucket.
4. Load Factor and Rahashing : when the number of entries exceeds the product of the load factor and the current capacity, the HashMap is resized, typically doubling the capacity to maintain performance.
5. Iteration: Iterating over a hashmap does not gurantee any particular order of the elements, as it depends on the hash codes and how they map to buckets.
6. Null Key : it allows one null key and multiple null values.

The default THREASHold value for the hashmap to convert linked list to binary tree is 8.

Binary Tree improves the worst-case performance.

Process of converting HashMap to Hash value :

Hashing function is used to convert an object into an integer hash code.

Hash code calculation : hashing function calculates the hash code based on the contents of the key object. The implementation of the hashing function can vary.

Once the hash value is calculated, it is used to determine which bucket in the hashmap internal array the key-value pair will be stored in. The bucket index is typically computed by the performing a modulo operation on the hash code with the size of the internal array.

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

What is Rehashing and Load Factor :-

In the context of HashMap in Java or similar data structures, rehashing and load factor are key concepts that relate to how the structure manages its internal storage and handles the distribution of elements. Let's break down each concept:

**Load Factor**

The **load factor** of a HashMap is a measure of how full the map is allowed to get before its capacity is automatically increased. It's defined as:

Load Factor=Number of elementsCurrent capacity of the HashMap\text{Load Factor} = \frac{\text{Number of elements}}{\text{Current capacity of the HashMap}}Load Factor=Current capacity of the HashMapNumber of elements​

In Java's HashMap, the default load factor is 0.75, which means the HashMap is resized (undergoes rehashing) when the number of elements reaches 75% of the current capacity.

**Rehashing**

**Rehashing** is the process of restructuring the internal storage of the HashMap when its load factor threshold is exceeded. This typically involves:

1. **Increasing Capacity**: Creating a new internal array (table) with a larger size to accommodate more elements. This new size is often a prime number to reduce collisions.
2. **Copying Elements**: Iterating through the existing elements in the old array and placing them into the new array according to their updated hash codes, which may change due to the new array size.
3. **Updating Hash Codes**: Recalculating hash codes of elements to fit into the new array slots.

**Why Rehashing and Load Factor Matter**

* **Performance**: By maintaining a reasonable load factor (default 0.75), HashMap balances between space efficiency and performance. It ensures that the map doesn't become too crowded, which would increase the likelihood of collisions (multiple elements having the same hash code).
* **Avoiding Collisions**: Collisions can degrade performance because elements with the same hash code must be stored in the same bucket and retrieved by iterating through them. Rehashing helps distribute elements more evenly across buckets, reducing collisions.
* **Dynamic Resizing**: HashMap dynamically resizes itself to maintain efficient operations. This resizing (rehashing) ensures that as elements are added, the map remains efficient in terms of lookup, insertion, and deletion times.

In summary, **rehashing** and **load factor** are mechanisms employed by HashMap (and similar data structures) to ensure efficient storage and retrieval of elements. Load factor dictates when rehashing occurs, and rehashing itself involves resizing the internal array and redistributing elements to maintain performance characteristics. These concepts are crucial for understanding how HashMap manages its capacity and handles collisions in a scalable way.

String/String Builder/Buffer

Tuesday, June 11, 2024

10:53 AM

String :

* Immutable, once created their values cannot be changed.
* Any operation that appears to modify string actually creates a new string object.
* Thread safe because of their immutability. Multiple threads can access and share string objects without the risk of data corruption.
* Performance : while strings are efficient for string manipulation operations that involve concatenation and comparison, repeated concatenation using the + operator can lead to performance issue due to creation of many temporary string objects.
* It uses string pool. If an object is created which is already in the pool it will point to the existing object rather than creating the new one.

StringBuffer :

* They are mutable, meaning their values can be modified after creation. They provide method to append, insert, delete and modify char in buffer.
* Thread-safe : it is designed to be thread safe, with synchronized methods for safe concurrent access. The synchronization comes with a performance cost.
* Suitable used in multithreaded env.

StirngBuilder :

* It is mutable and similar to string buffer.
* Not Thread safe like string buffer.
* Performance is better than string buffer in single threaded environment as it avoids the synchronization.
* Generally preferred in situations where thread safety is not a concern.

Central Exception Handler for Controller

Tuesday, June 11, 2024

8:55 PM

We can make use of the annonation @ControllerAdvice and @RestControllerAdvice

Annotate the class with these annonations and annotate the method with @ExceptionHandler class and inside that pass the exception name to be passed.

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleException(Exception ex) {

ex.printStackTrace();

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR)

.body("An unexpected error occurred");

}

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<String> handleResourceNotFoundException(ResourceNotFoundException ex) {

ex.printStackTrace();

return ResponseEntity.status(HttpStatus.NOT\_FOUND)

.body(ex.getMessage());

}

}

ukg

Tuesday, June 11, 2024

9:13 PM

What are inner beans.

Write a code to remove the duplicate elements from a list.

Write a code to limit the number of objects being created for a class in java.

How does hashset works internally.

Global exception handler for the controller class in spring.

How do you implement a class level lock.

Can we override the hash() function implemented in the hashmap.