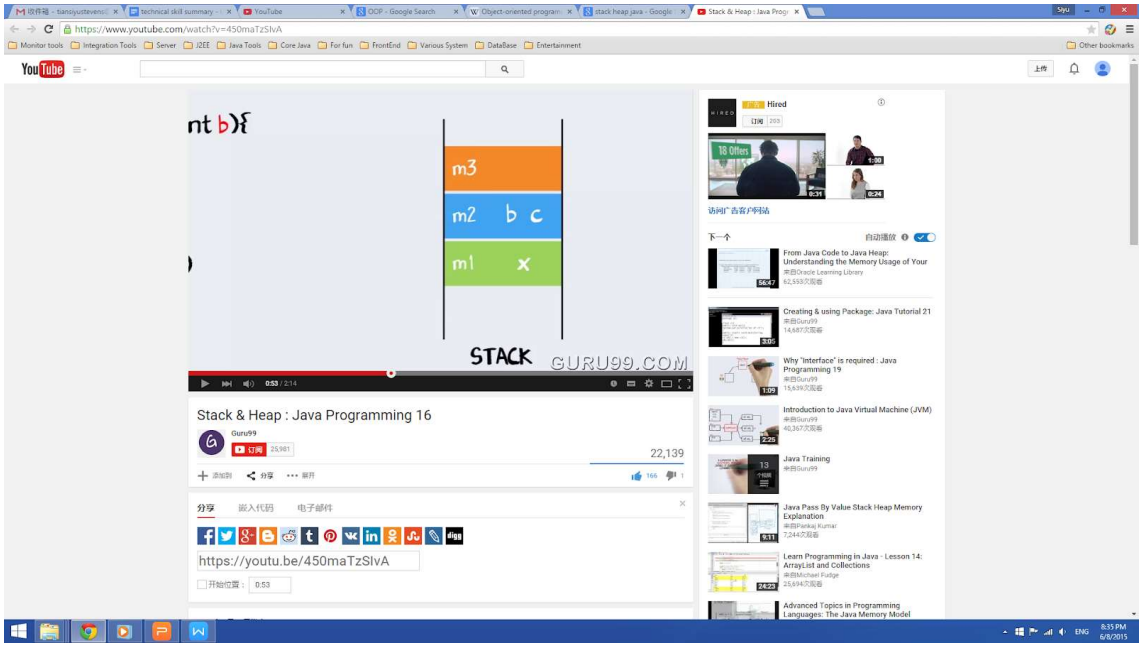
Basic concept

1. OOP Concept.
    1) encapsulation: internal private method and field, preventing from being accessed outside
    2) Inheritance: subclass will automatically inherit non-private method and fields
    3) Polymorphism: override and overload,
    4) Abstraction: some class is the abstraction of other class.
2. Stack vs heap



https://www.youtube.com/watch?v=450maTzSIvA

Java Heap Memory

Heap memory is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.
Stack memory size is very less compared to Heap memory.

3. String vs StringBuilder vs StringBuffer

String	*StringBuffer*	*StringBuilder*	
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable)	Yes(mutable)	Yes(mutable)
Thread Safe	Yes	Yes	No
Performance	Fast	Very slow	Fast

4. Boxing vs Autoboxing

Important point about Autoboxing and Unboxing in Java

1) Compiler uses valueOf() method to convert primitive to Object and uses intValue(), doubleValue() etc to get primitive value from Object.

2) During autoboxing boolean is converted to Boolean, byte to Byte, char converted to Character, float changes to Float, int goes to Integer, long goes to Long and short converts to Short, while in unboxing opposite happens like Float to float.

5. Downcast vs Upcast

when a variable of the base class (parent class) has a value of the derived class (child class), downcasting is possible.

```
Cat c1 = new Cat();
Animal a = c1; //automatic upcasting to Animal
Cat c2 = (Cat) a; //manual downcasting back to a Cat
```

Supercasting is always allowed, but subcasting involves a type check and can throw a ClassCastException.

6. Clone vs Cloneable

The **Cloneable** interface defines no members. It is used to indicate that a class allows a bitwise copy of an object (that is, a *clone*) to be made. If you try to call **clone( )** on a class that does not implement**Cloneable**, a **CloneNotSupportedException** is thrown. When a clone is made, the constructor for the object being cloned is *not* called. A clone is simply an exact copy of the original.

Cloning is a potentially dangerous action, because it can cause unintended side effects. For example, if the object being cloned contains a reference variable called *obRef,* then when the clone is made,*obRef* in the clone will refer to the same object as does *obRef* in the original. If the clone makes a change to the contents of the object referred to by *obRef,* then it will be changed for the original object, too. Here is another example. If an object opens an I/O stream and is then cloned, two objects will be capable of operating on the same stream. Further, if one of these objects closes the stream, the other object might still attempt to write to it, causing an error.

Because cloning can cause problems, **clone( )** is declared as **protected** inside **Object**. This means that it must either be called from within a method defined by the class that implements **Cloneable**, or it must be explicitly overridden by that class so that it is public. Let's look at an example of each approach.

7. Abstract class vs Interface

Abstract class	Interface
Partial implementation	No implementation
Single inheritance	Multiple inheritance

Any field	By default public static final fields and must be assigned immediately
Any method	[public abstract] method

8. static vs non-static
9. final, finally, finalize

final

final can be used to mark a variable "unchangeable"

```
private final String name = "foo";  //the reference name can never change
```

final can also make a method not "overrideable"

```
public final String toString() {  return "NULL"; }
```

final can also make a class not "inheritable". i.e. the class can not be subclasses.

```
public final class finalClass {...}public class classNotAllowed extends finalClass {...} // Not allowed
```

finally

finally is used in a try/catch statement to execute code "always"

```
lock.lock();try {  //do stuff} catch (SomeException se) {  //handle se} finally {  lock.unlock(); //always executed, even if Exception or Error or se}
```

Java 7 has a new try with resources statement that you can use to automatically close resources that explicitly or implicitly implement java.io.Closeable or java.lang.AutoCloseable

finalize

finalize is called when an object is garbage collected. You rarely need to override it. An example:

```
public void finalize() {  //free resources (e.g. unallocate memory)  super.finalize();}
```

10. Reflection:

The name reflection is used to describe code which is able to inspect other code in the same system (or itself).

For example, say you have an object of an unknown type in Java, and you would like to call a 'doSomething' method on it if one exists. Java's static typing system isn't really designed to support this unless the object conforms to a known interface, but using reflection, your code can look at the object and find out if it has a method called 'doSomething' and then call it if you want to.

So, to give you a code example of this in Java (imagine the object in question is foo) :

```
Method method = foo.getClass().getMethod("doSomething", null);method.invoke(foo, null);
```

> One very common use case in Java is the usage with annotations. JUnit 4, for example, will use reflection to look through your classes for methods tagged with the @Test annotation, and will then call them when running the unit test.

11. Access

	public	protected	(default)	private
Access within package	Yes	**Yes**	Yes	No

Subclass access within package	Yes	Yes	Yes	No
Access out of package	Yes	No	No	No
Subclass access out of package	Yes	Yes	No	No

12.Inner class
Static inner class can be instantiated without instantiating the outer class
Non-static inner class can be instantiated only after the outer class is instantiated.
Anonymous inner class
13. Garbage collection
Java has no destructor like C++. Garbage collection is applied to release memory occupied by objects that lose reference and are never used.
Garbage collection can be called explicitly by System.gc().
Garbage collection is a separate thread with the lowest priority.
When an object is removed by garbage collection, its finalize method is called to release its memory.
finalize method can be overridden and a lost reference might be obtained back.
Q: How to create an object that can not be garbage collected? A: create a static reference

Collection

1. Difference between List and Set

List is an ordered collection that keeps the insertion order. So it has get method to random access to an element at a specific index. However, Set is not ordered and can not have duplicated elements. (LinkedHashSet is a special subclass of a Set that can keep the insertion order.). To check whether the two objects a and b are duplicates, we use the boolean expression (a.hashCode()==b.hashCode() && (a==b || a.equals(b))) to justify.

2. Difference between ArrayList and Vector

Vector is thread-safe, while ArrayList is not. So ArrayList has better performance than Vector. When the capacity is reached, ArrayList will increase 50% of its capacity, while Vector will increase 100% of its capacity. To make an ArrayList thread-safe, we can call Collections.synchronizedList method.

3. Difference between ArrayList and LinkedList

ArrayList has better performance in random access, while LinkedList has better performance in insertion and deletion. Moreover, LinkedList does not have resizing issue while ArrayList has.

4. Difference between HashSet and TreeSet

TreeSet is sorted and it uses red-black tree algorithm, while HashSet is not sorted.

5. Difference between HashMap and TreeMap

Map has a method keyset to return a set of keys. HashMap's key set is a HashSet and TreeMap's key set is a TreeSet. Thus TreeSet is sorted by its keys, while HashMap is not.

6. Difference between HashMap, Hashtable and ConcurrentHashMap

	HashMap	Hashtable	ConcurrentHashMap
Thread-safe	No	Yes	Yes
Performance	Best	Worst	Good
Allow null as key	Yes	No	No
Fail-fast iterator	Yes	Yes	No

7. Difference between Comparator and Comparable interfaces

Comparator has one method compare that contains two parameters, while Comparable has one method compareTo that contains one parameter. If the objects of a class needs to be added to a TreeSet the class should implements Comparable interface. Comparator provides more flexible that a class can use multiple ways of sorting.

8. Difference between Iterator and Enumeration interfaces

Iterator has three methods next, hasNext and remove, while Enumeration has two methods nextElement, hasMoreElements. So Enumeration is read-only. Iterator has more simple methods and easy to use. Collection interface provide iterator() method and it means each collection can get its iterator.

9. How to make an iterator go back and forth?

ListIterator is a sub-interface of Iterator. It provides previous method and thus can make an iterator go back and forth.

10. What is fail-fast iterator

If a Collection gets its iterator, then the collection can not be modified (inserted or deleted). Otherwise, ConcurrentModificationException will be thrown during the runtime. So this iterator is called fail-fast iterator. Not all iterators are fail-fast. For example, the iterator of a ConcurrentHashMap may not be fail-fast.

11. How to make a List, Set or a Map thread-safe

Collections.synchronizedList,
Collections.synchronizedSet,
Collections.synchronizedMap.

12. How to make a List, Set or a Map read-only

Collections.unmodifiableList,
Collections.unmodifiableSet,
Collections.unmodifiableMap.

13. Which data structure can be used for mostly reading and not many writing operations?

CopyOnWriteArrayList

14. Which data structure has constant lookup time?

Hashtable, HashSet, HashMap, or Dictionary

Exception Handling:

1. Difference between checked and unchecked Exceptions

All the checked exceptions inherit from Exception class, but not RuntimeException class. They are also call compile time exception because they must be handled during the compile time. We can use try-catch-finally block to handle it, or use throws clause to bubble the exception to the caller. All the unchecked exceptions are subclasses of RuntimeException. We can also use try-catch block to catch them, but it is not necessary. They happened during the runtime and thus is also called as runtime exceptions.

2. About try-catch-finally block, can we use multiple catch block?

Yes. For multiple catch block, either the two exceptions have no relation, or the exception in the previous catch block is the subclass of the exception in the latter catch block.

3. In which case that finally block is not executed

System.exit(1) is executed in try or catch block.

4. Can we use try-finally block?

Yes.

I/O system

1. What is a Serializable object?

If a class implements Serializable interface, its object is a Serializable object. It means that this object can be converted into a binary stream.

2. If a class implements Serializable interface, which fields can not be serialized?

Two fields can not be serialized: transient field and static field

3. Difference between Serializable and Externalizable interfaces?

Three differences:

- Externalizable is a sub-interface of Serializable. It contains two methods readExternal and writeExternal.
- Externalizable interface can be used to control the serialization
- When an object is de-serialized, a Serializable object will be recovered directly, while for externalizable object, an empty object will be created first, and then the readExternal methods will be called to fill the fields.

You need to **define serialVersionUID in case of Serializable** and if it is not explicitly defined it will be generated automatically and it is based on all the fields, methods etc of the class and it changes every time you do the changes in the class. You if current id does not match with generated id you will not be able to recover the previously stored data.

Multi-Threading

1. How to create a thread?

   Two ways: extends Thread class, or implements Runnable interface. Actually, Thread class itself also implements Runnable interface. The latter way is better, because if a class implements Runnable interface, it can also inherit other classes.

2. Thread lifecycle

   Four phases: Ready, Running, Waiting and Dead

   When a start() method is called, the thread is in ready status; when CPU is free, the thread will go to running status; when it is finished, the thread is dead. There is a

special status called waiting. A running thread can go back to ready status by calling yield or join method; it can go to waiting status by calling sleep or wait method; A waiting thread can go to ready status by calling notify or notifyAll methods.

3. Different hierarchies or levels of synchronization
   Class level and object level. If a thread accesses a synchronized static method, it will get the class lock; if a thread accesses a synchronized non-static method, it will get the object lock.
4. Difference between wait and sleep method
   wait method will release the lock and other threads should call notify or notifyAll method to wake up the thread. However, sleep method is in the control and it will wake up automatically. Thus wait, notify and notifyAll methods must be in a synchronized method or synchronized block, otherwise a runtime exception called IllegalMonitorStateException will be thrown during the runtime.
5. Difference between yield and join method
   Both methods can change a thread from running status back to ready status. However, join() thread is like a stop sign, the caller will wait until the callee is finished; while yield() method can not guarantee that the caller is the next thread to be running.
6. What is a Daemon thread?
   Daemon threads will be stopped by OS after all other user threads are finished. We can use the method setDaemon(true) to change a user thread into a Daemon thread.
7. What is a volatile object?
   Multiple threads can read a volatile object at the same time. However, only one thread can update a volatile object at one time.
8. Difference between Runnable and Callable interfaces
   Runnable interface has run() method, while Callable interface has call() method. The call() method can throw exception and return value.
9. How to stop a thread? Does interrupt method stop a thread?
   There is no way to stop a thread. The interrupt method only releases the dead status resulted by join, sleep and wait methods.

- executor.shutDownNow() will invoke .interupt() method on all the threads it hold.
- you can use executor.submit(? extends Runnable) to return a Future<?> object to get the context of this Thread, and then invoke future.cancel(true) so that this thread get interrupted.
- A thread will go into blocked state when it runs into Thread.sleep(), I/O block or synchronized lock. However, only Thread.sleep() will be interrupted and throw InterruptedException.
- You may close the I/O resource to release the lock and keep the thread running.
- You can use ReentrantLock to make the synchronized lock interruptable.
- How to exit Thread with loop:
  - 1) judge on Thread.interrupted() so that when the thread is not blocked, it will jump out of the loop
  - 2) try catch InterruptedException

JDBC

1. Difference between Statement, PreparedStatement and CallableStatement?

Two differences between Statement and PreparedStatement
PreparedStatement can run a query with parameters and thus can prevent inline SQL problem; PreparedStatement is precompiled and has better performance.
Difference between PreparedStatement and CallableStatement
Both of them can call stored procedures, but CallableStatement can call a stored procedure with both input and output parameters, while PreparedStatement can call a stored procedure with only input parameters.

2. How to make a transaction in JDBC?

After we get a JDBC connection variable, say conn, then

```
conn.setAutoCommit(false);
// run all queries
conn.commit();
```

3. How to find the schema in a ResultSet?

A ResultSet has a method called getMetaData() and it returns an object of ResultSetMetadata. This object has methods getColumnCount, getColumnName and getColumnType method to retrieve the schema from a ResultSet.

4. Four types of JDBC drivers
    - Type 1: JDBC-ODBC Bridge driver (Bridge)
    - Type 2: Native-API/partly Java driver (Native)
    - Type 3: All Java/Net-protocol driver (Middleware)
    - Type 4: All Java/Native-protocol driver (Pure)

We are now using Type 4 JDBC driver.

Spring Security

When using servlet filters, you obviously need to declare them in your web.xml, or they will be ignored by the servlet container. In Spring Security, the filter classes are also Spring beans defined in the application context and thus able to take advantage of Spring's rich dependency-injection facilities and lifecycle interfaces**. Spring's DelegatingFilterProxy provides the link between web.xml and the application context.**
When using DelegatingFilterProxy, you will see something like this in the web.xml file:

```
<filter>
  <filter-name>myFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
  <filter-name>myFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

What`DelegatingFilterProxy` does is delegate the `Filter`'s methods through to a bean which is obtained from the Spring application context**.**

When we looked at how to set up web security using namespace configuration, we used a `DelegatingFilterProxy` with the name "springSecurityFilterChain". You should now be able to see that this is the name of the `FilterChainProxy` which is created by the namespace.
`FilterSecurityInterceptor` is responsible for handling the security of HTTP resources. It requires a reference to an`AuthenticationManager` and an `AccessDecisionManager`.

- delegatingFilterProxy
- filterChainProxy

  Spring Security's web infrastructure should only be used by delegating to an instance of FilterChainProxy. The security filters should not be used by themselves. In theory you could declare each Spring Security filter bean that you require in your application context file and add a corresponding DelegatingFilterProxy entry to web.xml for each filter, making sure that they are ordered correctly, but this would be cumbersome and would clutter up the web.xml file quickly if you have a lot of filters. FilterChainProxy lets us add a single entry to web.xml and deal entirely with the application context file for managing our web security beans. It is wired using a`DelegatingFilterProxy`, just like in the example above, but with the filter-name set to the bean name "filterChainProxy". The filter chain is then declared in the application context with the same bean name. Here's an example:

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

Core component or concept:

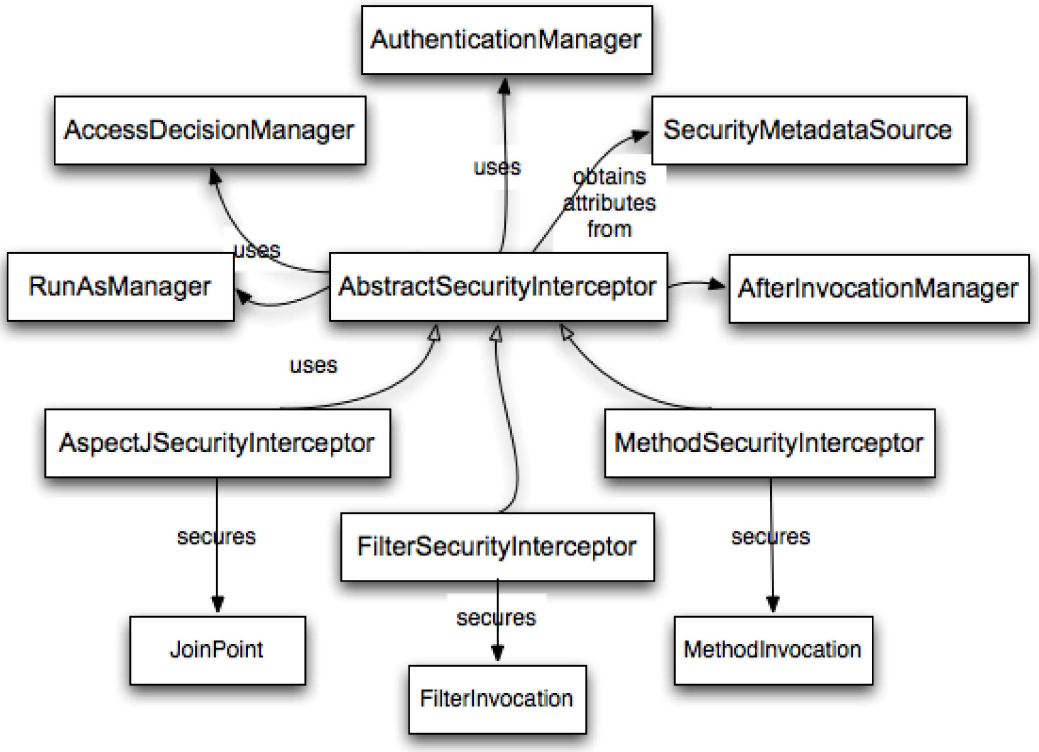
- SecurityContextHolder, to provide access to the SecurityContext.
- SecurityContext, to hold the Authentication and possibly request-specific security information.
- Authentication, to represent the principal in a Spring Security-specific manner.
- GrantedAuthority, to reflect the application-wide permissions granted to a principal.
- UserDetails, to provide the necessary information to build an Authentication object from your application's DAOs or other source of security data.
- UserDetailsService, to create a UserDetails when passed in a String-based username (or certificate ID or the like).
- AuthenticationManager: to authenticate the autheication token and return a fully populated authentication upon successful authentication.
- AuthenticationProvider:

  The AuthenticationManager is just an interface, so the implementation can be anything we choose, but how does it work in practice? What if we need to check multiple authentication databases or a combination of different authentication services such as a database and an LDAP server?
  The default implementation in Spring Security is called ProviderManager and rather than handling the authentication request itself, it delegates to a list of configured AuthenticationProvider s, each of which is queried in turn to see if it can perform the authentication. Each provider will either throw an exception or return a fully populated Authentication object.

- AbstractSercurityInterceptor:

  AbstractSecurityInterceptor provides a consistent workflow for handling secure object requests, typically:

  1. Look up the "configuration attributes" associated with the present request
  2. Submitting the secure object, current Authentication and configuration attributes to the AccessDecisionManager for an authorization decision
  3. Optionally change the Authentication under which the invocation takes place
  4. Allow the secure object invocation to proceed (assuming access was granted)
  5. Call the AfterInvocationManager if configured, once the invocation has returned. If the invocation raised an exception, theAfterInvocationManager will not be invoked.



- AccessDecisionManager: has a decade method which takes an authentication object representing the principal requesting access.

JSP

- 9 jsp implicit object
    - out : instance of `javax.servlet.jsp.JspWriter` implementation and it's used to output content to be sent in client response. This is one of the most used JSP implicit object and thats why we have JSP Expression to easily invoke *out.print()* method.
    - request: We can use request object to get the request parameters, cookies, request attributes, session, header information and other details about client request.
    - response: We can response object to set content type, character encoding, header information in response, adding cookies to response and redirecting the request to other resource.
    - config:get the JSP init params configured in deployment descriptor.
    - application: it's used to get the context information and attributes in JSP.We can use it to get the RequestDispatcher object in JSP to forward the request to another resource
    - session: Whenever we request a JSP page, container automatically creates a session for the JSP in the service method.
    - pageContext: We can use pageContext to get and set attributes with different scopes and to forward request to other resources. pageContext object also hold reference to other implicit object.

    - page: page object provide reference to the generated servlet class. This object is very rarely used.
    - exception: used to provide exception details in JSP error pages. We can't use this object in normal JSP pages and it's available only in JSP error pages.
- Four jsp tags:
    - JSP expression: <%= %>
    - JSP declaration: <%! %>
    - JSP scriptlet: <% %>
    - JSP directive: <%@ %>
        - Page direct: <%@page import="..."%> Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
        - Include direct: <%@include file="..."%> Includes a file during the translation phase.
        - Taglib direct: <%@taglib uri="..." prefix="..."%> Declares a tag library, containing custom actions, used in the page
- **Explain the uses of <jsp:usebean> tag.**

<jsp:useBean>
 id="beanInstName"
 scope= "page | application"
 class="ABC.class" type="ABC.class"
 </jsp:useBean>
This tag creates an instance of java bean. It firstly tries to find if bean instance already exist and assign stores a reference in the variable. Type is also specified; otherwise it instantiates from the specified class storing a reference in the new variable.

Attributes and Usage of jsp:useBean action tag

1. **id:** is used to identify the bean in the specified scope.
2. **scope:** represents the scope of the bean. It may be page, request, session or application. The default scope is page.
    - **page:** specifies that you can use this bean within the JSP page. The default scope is page.
    - **request:** specifies that you can use this bean from any JSP page that processes the same request. It has wider scope than page.
    - **session:** specifies that you can use this bean from any JSP page in the same session whether processes the same request or not. It has wider scope than request.
    - **application:** specifies that you can use this bean from any JSP page in the same application. It has wider scope than session.
3. **class:** instantiates the specified bean class (i.e. creates an object of the bean class) but it must have no-arg or no constructor and must not be abstract.

4. **type:** provides the bean a data type if the bean already exists in the scope. It is mainly used with class or beanName attribute. If you use it without class or beanName, no bean is instantiated.
5. **beanName:** instantiates the bean using the java.beans.Beans.instantiate() method.

- jsp <%@ include file="use.html" %> vs <jsp:include page="use.html">
    - the include directive inserts the source of `reuse.html` at translation time but the action tag inserts the response of reuse.html at runtime.

Hibernate

- important classes
    - Configuration: An instance of `Configuration` allows the application to specify properties and mapping documents to be used when creating a SessionFactory.
        - configure(): configure using default file "hibernate.cfg.xml"
        - to connect to different database, create different configuration object to build sessionFactory
    - SessionFactory: Creates Sessions. Usually an application has a single SessionFactory. Threads servicing client requests obtain Sessions from the factory.
    - Session: The main function of the Session is to offer create, read and delete perations for instances of mapped entity classes. Instances may exist in one of three states:
        - *transient:* never persistent, not associated with any Session.an object is transient if it has just been instantiated using the new operator, and it is not associated with a Hibernate Session. It has no persistent representation in the database and no identifier value has been assigned. Transient instances will be destroyed by the garbage collector if the application does not hold a reference anymore. Use the Hibernate Session to make an object persistent (and let Hibernate take care of the SQL statements that need to be executed for this transition).
        - *persistent:* associated with a unique Session. a persistent instance has a representation in the database and an identifier value. It might just have been saved or loaded, however, it is by definition in the scope of a Session. Hibernate will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work

completes. Developers do not execute manual UPDATE statements, or DELETE statements when an object should be made transient.

- *detached:* previously persistent, not associated with any Session. a detached instance is an object that has been persistent, but its Session has been closed. The reference to the object is still valid, of course, and the detached instance might even be modified in this state. A detached instance can be reattached to a new Session at a later point in time, making it (and all the modifications) persistent again. This feature enables a programming model for long running units of work that require user think-time. We call them *application transactions*, i.e., a unit of work from the point of view of the user.
- Transient instances may be made persistent by calling `save()`, `persist()` or `saveOrUpdate()`. Persistent instances may be made transient by calling `delete()`. Any instance returned by a `get()` or `load()` method is persistent. Detached instances may be made persistent by calling `update()`, `saveOrUpdate()`, `lock()` or `replicate()`. The state of a transient or detached instance may also be made persistent as a new persistent instance by calling `merge()`
- `load()`: Return the persistent instance of the given entity class with the given identifier, assuming that the instance exists.You should not use this method to determine if an instance exists (use `get()` instead).

- Transaction: A transaction is associated with a Session and is usually instantiated by a call to Session.beginTransaction().
- EhcacheProvider: Provide second-level-cache in hibernate. Use hibernate.cache.provider_class=org.hibernate.cache.EhCacheProvider in the Hibernate configuration to enable this provider for Hibernate's second level cache. Configuration will be read from ehcache.xml for a cache declaration. or specified in hibernate.cache.provider_configuration_file_resource_path

- How to disable second-level-cache?
  <property name="hibernate.cache.use_second_level_cache">false</property>
- differences between get() and load()?
  - load method applies proxy pattern. It will return a proxy object first. When the details of the object are accessed, it will run the corresponding query and return the real object. So it has lazy-loading feature. If an object does not exist, it will throw exception.
  - get method will run the corresponding query directly and return the real object. If the object does not exist, it will return null.
  - Both load and get methods need to know the primary key of the object.

- Four types of conditions in Criteria?
    - Restrictions

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

    - Orders

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

    - Example

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

    - Projection

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

- How to delete/update records

There are three ways:

- Use HQL/SQL to run a query. It can be applied if many records need to be updated or deleted.
- Use load or get method to load the object, then use setter to change fields and finally commit. It is easy to use and very smart because an update query will not be created if no fields are changed.
- Similar to the second way, but use criteria to load the object. No primary is required in this way.

- Three basic mapping strategy fro class hierachy.
    - **Table per class hierarchy**
      Suppose we have an interface Payment with the implementors CreditCardPayment, CashPayment, andChequePayment. The table per hierarchy mapping would display in the following way:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

      Exactly one table is required. There is a limitation of this mapping strategy: columns declared by the subclasses, such as CCTYPE, cannot have NOT NULL constraints.
    - **Table per sub-class**
      A table per subclass mapping looks like this:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
```

```
    <key column="PAYMENT_ID"/>
    ...
  </joined-subclass>
</class>
```

Four tables are required. The three subclass tables have primary key associations to the superclass table so the relational model is actually a one-to-one association.

- Table per concrete-class

  There are two ways we can map the table per concrete class strategy. First, you can use <union-subclass>.

```
<class name="Payment">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="sequence"/>
  </id>
  <property name="amount" column="AMOUNT"/>
  ...
  <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <property name="creditCardType" column="CCTYPE"/>
    ...
  </union-subclass>
  <union-subclass name="CashPayment" table="CASH_PAYMENT">
    ...
  </union-subclass>
  <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    ...
  </union-subclass>
</class>
```

Three tables are involved for the subclasses. Each table defines columns for all properties of the class, including inherited properties.
The limitation of this approach is that if a property is mapped on the superclass, the column name must be the same on all subclass tables. The identity generator strategy is not allowed in union subclass inheritance. The primary key seed has to be shared across all unioned subclasses of a hierarchy.
If your superclass is abstract, map it with abstract="true". If it is not abstract, an additional table (it defaults toPAYMENT in the example above), is needed to hold instances of the superclass.

- Lazy loading and LazyInitializationException
  - In Hibernate, if it loads one bean object, it won't load its dependent objects unless we access the dependent objects. This is called lazy loading. It can be disabled by setting lazy="false" but this way is not suggested.
  - Lazy loading is a very good feature in Hibernate. It can save the memory and increase the performance.

    - If the session is closed before the dependent objects are loaded, when we try to load the dependent object, the system will throw LazyInitializationException.
    - To avoid LazyInitializationException, we have the following solutions.
        - Use Hibernate.initialize method to initialize the dependent data before the session is closed. This way always works.
        - If we use HQL to connect to database, we can add “left join fetch” keyword in the query to fetch the dependent data.
        - If we use load, get or criteria to connect to database, we can add fetch=”join” in the mapping configuration file for dependent data.
        - If we use Spring to integrate with Hibernate, we can configure OpenSessionInViewFilter or OpenSessionInViewInterceptor class to open a big session that will never be closed.
- HQL
    - A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select.

```
from Cat as cat
        inner join fetch cat.mate
        left join fetch cat.kittens
```

    - You may supply extra join conditions using the HQL with keyword.
        -
```
from Cat as cat
  left join cat.kittens as kitten
    with kitten.bodyWeight > 10.0
```

    - The where clause allows you to refine the list of instances returned. If no alias exists, you can refer to properties by name:
        - `from Cat where name='Fritz'`

- Spring bean scope
    - singleton:This scopes the bean definition to a single instance per Spring IoC container (default).
    - protoype:This scopes a single bean definition to have any number of object instances.
    - request:This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
    - session:This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
    - global-session:This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
- Spring auto-wire: Controls whether bean properties are "autowired". This is an automagical process in which bean references don't need to be coded explicitly in the XML bean definition file, but rather the Spring container works out dependencies. There are 4 modes:
    - "no" The traditional Spring default. No automagical wiring. Bean references must be defined in the XML file via the <ref/> element (or "ref" attribute). We recommend this in most cases as it makes documentation more explicit. Note