Seven SEAS

Solutions for Enterprise Applications and Services

Shiva R Dhanuskodi

Mesonsoft LLC

An awesome book for invincible programmers.

For online information and downloading of this and other books published by Mesonsoft LLC, please go to www.sevenseasbook.us or www.mesonsoft.com or www.anishiv.com.

For more information, please contact:
Mesonsoft LLC
Email: shiva.dhanuskodi@mesonsoft.com

Preface

First of all, this book is not a tutorial for any frameworks, methodologies or technologies that are currently in use. But, I can guarantee you that this book will help you to sharpen your Java skills and prepare you for future challenges with your career. You can use it as a personal guide to keep you on track. Enough said!

Let's just dive into **Seven SEAS** (**S**olutions for **E**nterprise **A**pplications & **S**ervices)

1. One Platform-Independent Language (Core Java)

2. Two Unique Languages (Unified Modeling Language and Structured Query Language)

3. Three-Stick NIM – Network, Integration and Management (Computer Networking, Clustering & Continuous Integration Process [Hudson- Jenkins], Project Management [Maven, ANT/Ivy] & Unit Testing [JUnit, Easy Mock & Mockito])

4. Gang of Four (23 Java Design Patterns) and J2EE design patterns

5. Five Methodologies (RUP, Six Sigma [DMAIC], SDLC, Agile Software Development and Waterfall Model)

6. Six Frameworks (Struts, Spring, Hibernate, iBatis, Apache Axis & Jersey)

7. Seven useful Java EE APIs (Messaging Service [JMS], Enterprise Bean [EJB], Persistence [JPA/JDBC], User Interface [JSF/JSP], Security [JAAS], SOAP and REST [JAXB/JAXP] and Web Service [JAX-RS/JAX-WS/JAX-RPC])

Bonus Features JavaScript, jQuery, AngularJS, AJAX, JSON, DOJO & UNIX Shell Scripting

My Favorites Websites are: www.developersbook.com, www.mkyong.com, www.w3schools.org, www.wikipedia.org, beginner-sql-tutorial.com, www.plsql-tutorial.com, java.dzone.com and of course www.sevenseasbook.us

Chapter 1 – OOPS

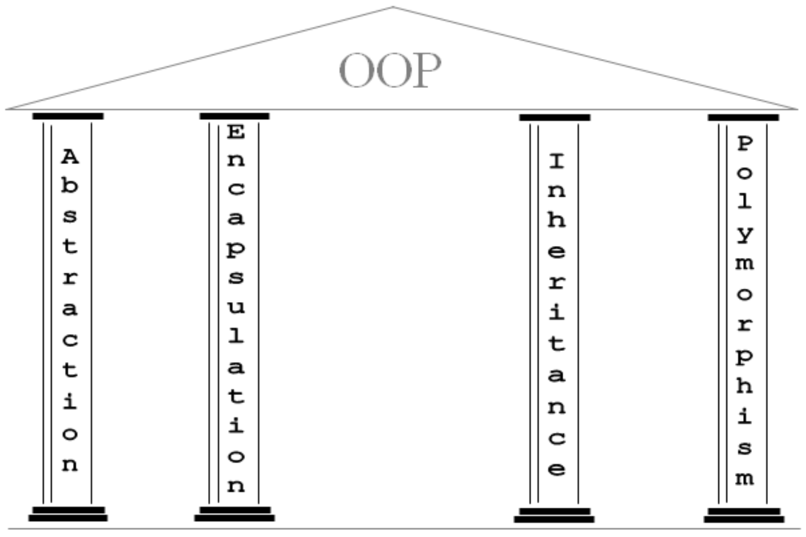
Object Oriented Programming Systems (OOPS)

A PIE – **A**bstraction, **P**olymorphism, **I**nheritance and **E**ncapsulation.

Object Orientation involving encapsulation, inheritance, polymorphism, and abstraction, is an important approach in programming and program design. It is widely accepted and used in industry and is growing in popularity.

So, the four principle concepts upon which object oriented design and programming rest are:

- Abstraction
- Polymorphism
- Inheritance
- Encapsulation



Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations.

Encapsulation

Encapsulation is a technique used for hiding the properties and behaviors of an object and allowing outside access only as appropriate. It prevents other objects from directly altering or accessing the properties or methods of the encapsulated object.

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class.

- A class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass.
- Inheritance is done by using the keyword extends.
- The two most common reasons to use inheritance are:

    1. To promote code reuse
    2. To use polymorphism

Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods.

Polymorphism

Polymorphism is briefly described as "one interface, many implementations." Polymorphism is a characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.

Different forms of Polymorphism

There are two types of polymorphism one is Compile time polymorphism and the other is run time polymorphism. Compile time polymorphism is method overloading. Runtime time polymorphism is method overriding done using inheritance and interface.

Note: From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

Inheritance, Overloading and Overriding are used to achieve Polymorphism in java. Polymorphism manifests itself in Java in the form of multiple methods having the same name.
In some cases, multiple methods have the same name, but different formal argument lists (overloaded methods). In other cases, multiple methods have the same name, same return type, and same formal argument list (overridden methods).

Runtime polymorphism or Dynamic method dispatch

In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.

Method Overloading

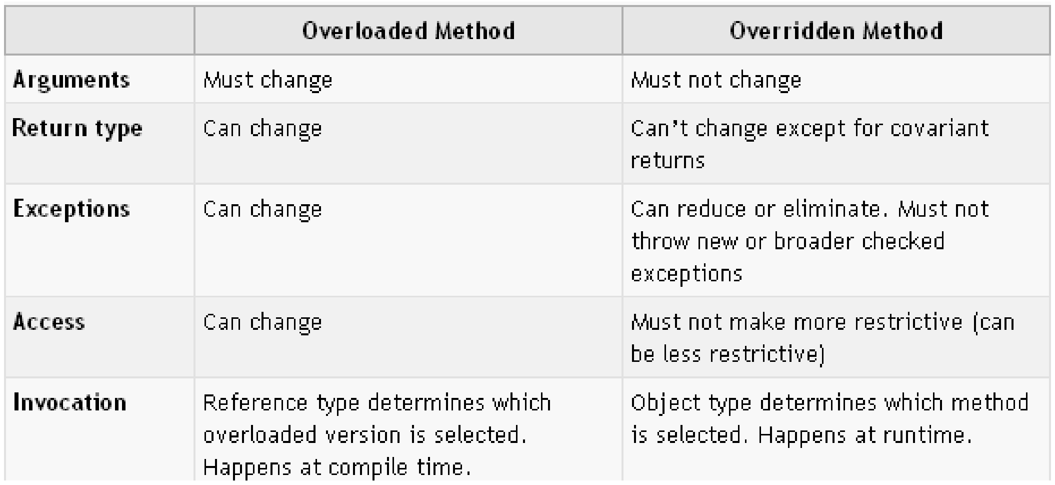
Method Overloading means to have two or more methods with same name in the same class with different arguments. The benefit of method overloading is that it allows you to implement methods that support the same semantic operation but differ by argument number or type.

- Overloaded methods MUST change the argument list
- Overloaded methods CAN change the return type
- Overloaded methods CAN change the access modifier
- Overloaded methods CAN declare new or broader checked exceptions
- A method can be overloaded in the same class or in a subclass

Method Overriding

Method overriding occurs when sub class declares a method that has the same type arguments as a method declared by one of its superclass. The key benefit of overriding is the ability to define behavior that's specific to a particular subclass type.

- The overriding method cannot have a more restrictive access modifier than the method being overridden (Ex: You can't override a method marked public and make it protected).
- You cannot override a method marked final
- You cannot override a method marked static



	Overloaded Method	Overridden Method
Arguments	Must change	Must not change
Return type	Can change	Can't change except for covariant returns
Exceptions	Can change	Can reduce or eliminate. Must not throw new or broader checked exceptions
Access	Can change	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version is selected. Happens at compile time.	Object type determines which method is selected. Happens at runtime.

Difference between Abstraction and Encapsulation

Abstraction focuses on the outside view of an object (i.e. the interface) Encapsulation (information hiding) prevents clients from seeing it's inside view, where the behavior of the abstraction is implemented.

Abstraction solves the problem in the design side while Encapsulation is the Implementation.
Encapsulation is the deliverables of Abstraction. Encapsulation barely talks about grouping up your abstraction to suit the developer needs.

Interface versus Abstract Class

1. Abstract class is a class which contains one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain no abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants and doesn't contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present.

2. Abstract class definition begins with the keyword "abstract" keyword followed by Class definition. An Interface definition begins with the keyword "interface".

3. Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses

4. All variables in an Interface are by default - public static final while an abstract class can have instance variables.

5. An interface is also used in situations when a class needs to extend another class apart from the abstract class. In such situations it's not possible to have multiple inheritances of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple Inheritances.

6. An Interface can only have public members whereas an abstract class can contain private as well as protected members.

7. A class implementing an interface must implement all of the methods defined in the interface, while a class extending an abstract class need not implement any of the methods defined in the abstract class.

8. The problem with an interface is, if you want to add a new feature (method) in its contract, then you MUST implement those methods in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass

9. Interfaces are slow as it requires extra indirection to to find corresponding method in in the actual class. Abstract classes are fast

10. Interfaces are often used to describe the peripheral abilities of a class, and not its central identity, E.g. an Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.

Chapter 2 - Core Java

JVM - Write Once, Run Everywhere

JVM, or the Java Virtual Machine, is an interpreter which accepts 'Byte code' and executes it.

Java has been termed as a 'Platform Independent Language' as it primarily works on the notion of 'compile once, run everywhere'. Here's a sequential step establishing the Platform independence feature in Java:

- The Java Compiler outputs Non-Executable Codes called 'Byte code'.
- Byte code is a highly optimized set of computer instruction which could be executed by the Java Virtual Machine (JVM).
- The translation into Byte code makes a program easier to be executed across a wide range of platforms, since all we need is a JVM designed for that particular platform.
- JVMs for various platforms might vary in configuration, those they would all understand the same set of Byte code, thereby making the Java Program 'Platform Independent'.

JDK versus JRE

The "JDK" is the Java Development Kit. i.e., the JDK is bundle of software that you can use to develop Java based software. Typically, each JDK contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.

The "JRE" is the Java Runtime Environment. I.e., the JRE is an implementation of the Java Virtual Machine which actually executes Java programs.

Java History

Java SE 8 - Jigsaw 2014

Java SE 7 - Dolphin 2011

Java SE 6 - Mustang 2006

- JDBC 4.0 support
- Support for pluggable annotations

J2SE 5.0 (1.5) - Tiger 2004

1. **Scanner** - Used to convert text into primitives or Strings

2. **Printf** - The format() and printf() methods were added to java.io.PrintStream

3. **Auto-boxing** - Eliminates the manual conversion between primitive types(int)and wrapper types(Integer)

4. **Type safe Enums** - Type Code "enum" (new keyword) with methods and fields

5. **For each loop (Enhanced For Loop)** - Eliminates the error-proneness of iterators

6. **Generics** - Adds compile-time type safety to Collections Framework and eliminates the need for casting

7. **Static Imports** - Import static fields and methods

8. **VarArgs** - Eliminates the need for manual boxing up argument list into an array

9. **Metadata (Annotations)** - This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it.

10. **Reflection** - Added support for generics, annotations and enums

11. **Collections Framework** - Three new interfaces (Queue -2 - Concurrent, Blocking Queue - 5 - Concurrent, ConcurrentMap - 1 - ConcurrentHashMap) Copy-on-write List and Set implementations - CopyOnWriteArrayList

12. **Concurrent Utilities** – It provides a powerful, extensible framework of high performance, scalable and thread-safe concurrent apps (java.util.concurrent.*).

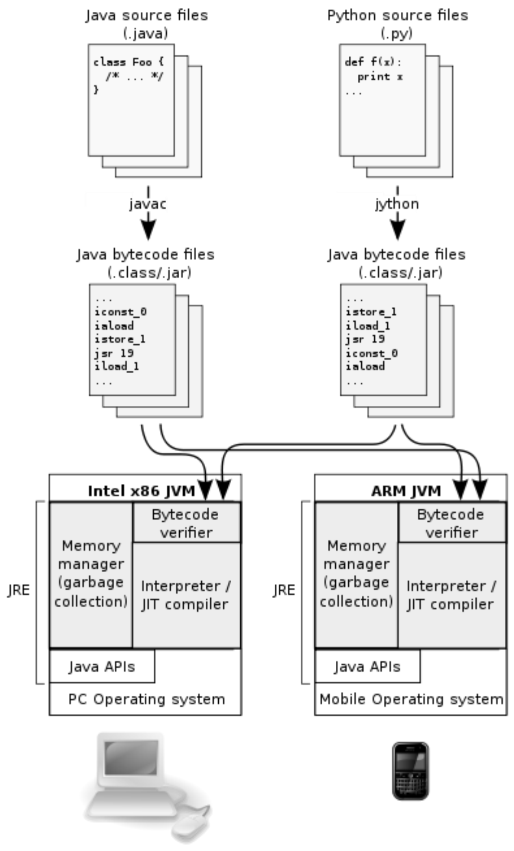
    Mutex - a non-entrant mutual exclusion lock which is another term for a lock. This utility class is used to control locking mechanism.

J2SE 1.4 - Merlin 2002
J2SE 1.3 - Kestrel 2000
J2SE 1.2 - Playground 1998
JDK 1.1 - 1997
JDK 1.0 - Code Name: Oak 1996 Initial Release – Founder: James Gosling

Chapter 3 – JVM and Class Loader

Java Virtual Machine (JVM)

A Java virtual machine is software that is implemented on non-virtual hardware and on standard operating systems. A JVM provides an environment in which Java byte code can be executed, enabling such features as automated exception handling, which provides "root-cause" debugging information for every software error (exception), independent of the source code. A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together with JVM form the Java Runtime Environment (JRE).



Java Class Loader

Each Java class must be loaded by a class loader. Furthermore, Java programs may make use of external libraries (that is, libraries written and provided by someone other than the author of the program) or they may be composed, at least in part, of a number of libraries.

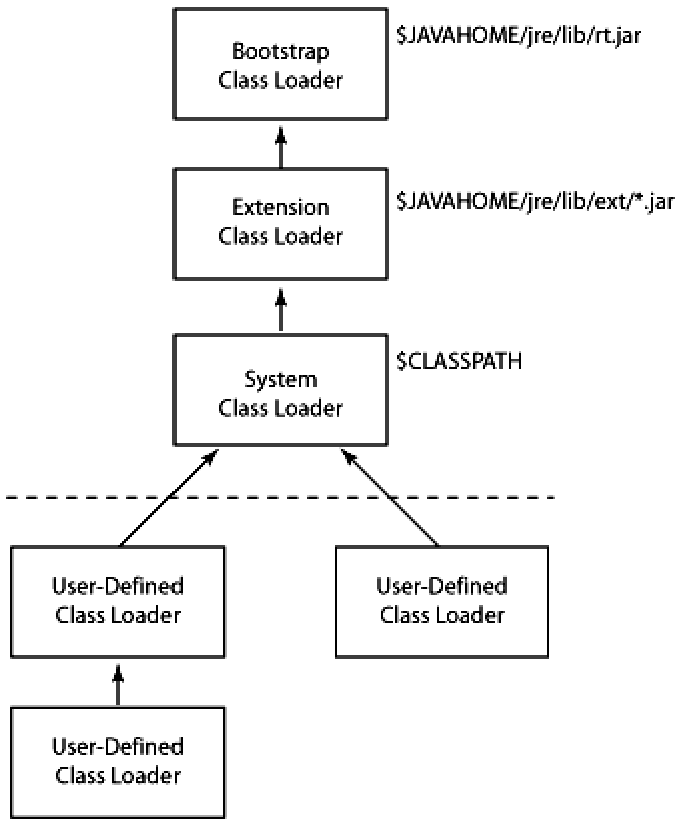
When the JVM is started, three class loaders are used:

1. Bootstrap class loader
2. Extensions class loader
3. System class loader

The bootstrap class loader loads the core Java libraries (<JAVA_HOME>/lib directory). This class loader, which is part of the core JVM, is written in native code.

The extensions class loader loads the code in the extensions directories (<JAVA_HOME>/lib/ext or any other directory specified by the java.ext.dirs system property). It is implemented by the sun.misc.Launcher$ExtClassLoader class.

The system class loader loads code found on java.class.path, which maps to the system CLASSPATH variable. This is implemented by the sun.misc.Launcher$AppClassLoader class.


Bootstrap Class Loader
$JAVAHOME/jre/lib/rt.jar
Extension Class Loader
$JAVAHOME/jre/lib/ext/*.jar
System Class Loader
$CLASSPATH
User-Defined Class Loader
User-Defined Class Loader
User-Defined Class Loader

Class Loader Delegation

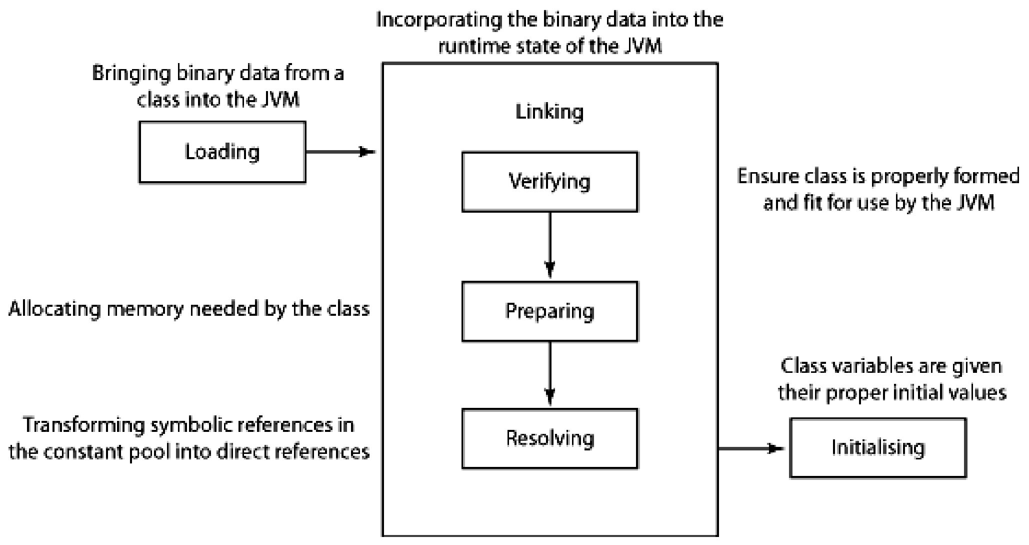
The loading of Java classes is performed by class loaders (CL), they are responsible for loading classes into the JVM. Simple applications can use the Java platform's built-in class loading facility to load their classes, more complex applications tend to define their own custom class loaders.

The class loaders in Java are organized in a tree. By request a class loader determines if the class has already been loaded in the past, looking up in its own cache. If the class is present in the cache the CL returns the class, if not, it delegates the request to the parent. If the parent is not set (is Null) or cannot load the class and throws a ClassNotFoundException the classloader tries to load the class itself and searches its own path for the class file. If the class can be loaded it is returned, otherwise a ClassNotFoundException is thrown. The cache lookup goes on recursively from child to parent, until the tree root is reached or a class is found in cache. If the root is reached the class loaders try to load the class and unfold the recursion from parent to child. Summarizing that we have following order:

- Cache
- Parent
- Self

This mechanism ensures that classes tending to be loaded by class loaders nearest to the root. Remember, that parent class loader is always has the opportunity to load a class first. It is important to ensure that core Java classes are loaded by the bootstrap loader, which guarantees that the correct versions of classes such as java.lang.Object are loaded. Furthermore it ensures that one class loader sees only classes loaded by itself or its parent (or further ancestors) and it cannot see classes loaded by its children or siblings.

Phases of class loading

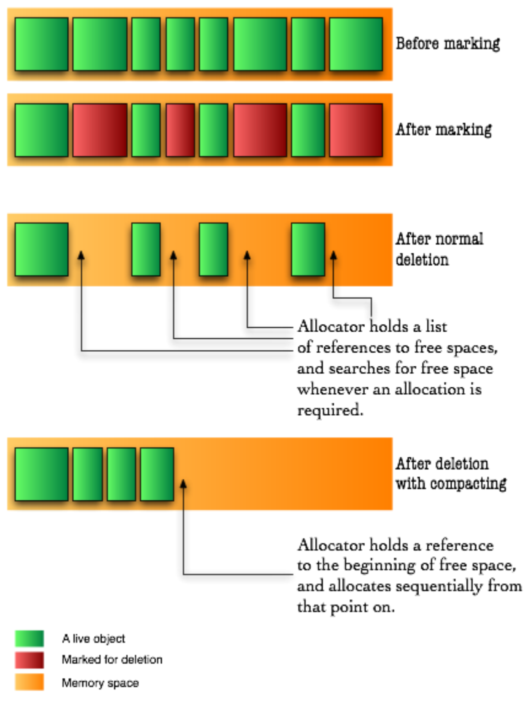

Incorporating the binary data into the runtime state of the JVM
Bringing binary data from a class into the JVM
Loading
Linking
Verifying
Ensure class is properly formed and fit for use by the JVM
Allocating memory needed by the class
Preparing
Class variables are given their proper initial values
Transforming symbolic references in the constant pool into direct references
Resolving
Initialising

Chapter 4 – Garbage Collector

The basis of garbage collection

The garbage collector first performs a task called marking. The garbage collector traverses the application graph, starting with the root objects; those are objects that are represented by all active stack frames and all the static variables loaded into the system. Each object the garbage collector meets is marked as being used, and will not be deleted in the sweeping stage.

The sweeping stage is where the deletion of objects takes place. There are many ways to delete an object: The traditional C way was to mark the space as free, and let the allocator methods use complex data structures to search the memory for the required free space. This was later improved by providing a defragmenting system which compacted memory by moving objects closer to each other, removing any fragments of free space and therefore allowing allocation to be much faster:


Before marking
After marking
After normal deletion
Allocator holds a list of references to free spaces, and searches for free space whenever an allocation is required.
After deletion with compacting
Allocator holds a reference to the beginning of free space, and allocates sequentially from that point on.
A live object
Marked for deletion
Memory space

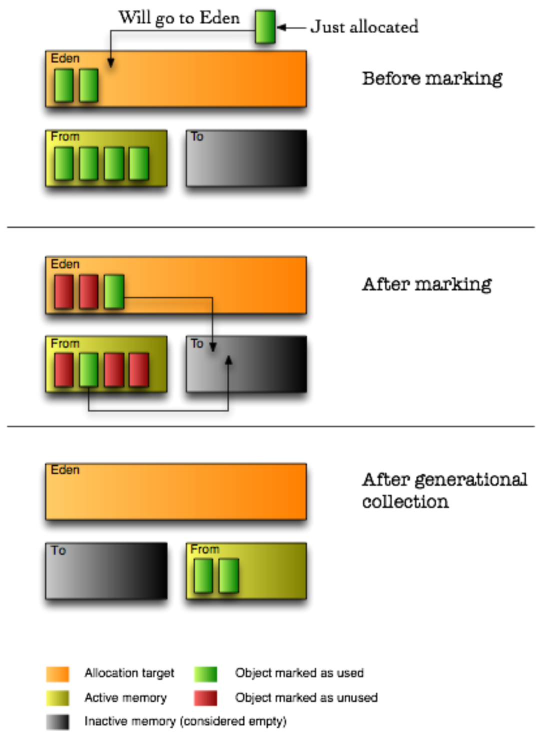
For the last trick to be possible a new idea was introduced in garbage collected languages: even though objects are represented by references, much like in C, they don't really reference their real memory location. Instead, they refer to a location in a dictionary which keeps track of where the object is at any moment.

Fortunately for us – but unfortunately for these garbage collection algorithms – our servers and personal computers got faster (and multiple) processors and bigger memory capacities. Compacting memory areas this large often was very taxing on the application, especially considering that when doing that, the whole application had to freeze due to the changes in the virtual memory map. Fortunately for us though, some smart people improved those algorithms in three ways: concurrency, parallelization and generational collection.

Generational Garbage Collection

In any application, objects could be categorized according to their life-line. Some objects are short-lived, such as most local variables, and some are long-lived such as the backbone of the application. The thought about generational garbage collection was made possible with the understanding that in an application's lifetime, most instantiated objects are short-lived, and that there are few connections between long-lived objects to short-lived objects.

In order to take advantage of this information, the memory space is divided to two sections: young generation and old generation. In Java, the long-lived objects are further divided again to permanent objects and old generation objects. Permanent objects are usually objects the Java VM itself created for caching like code, reflection information etc. Old generation objects are objects that survived a few collections in the young generation area.



Since we know that objects in the young generation memory space become garbage early, we collect that area frequently while leaving the old generation's memory space to be collected in larger intervals. The young generation memory space is much smaller, thus having shorter collection times.

An additional advantage to the knowledge that objects die quickly in this area, we can also skip the compacting step and do something else called copying. This means that instead of seeking free areas (by seeking the areas marked as unused after the marking step), we copy the live objects from one young generation area to another young generation area. The originating area is called the ‘From’ area, and the target area is called the ‘To’ area, and after the copying is completed the roles switch: the ‘From’ becomes the ‘To’, and the ‘To’ becomes the ‘From’.

In addition, the Java VM splits the young generation to three areas, by adding an area called Eden which is where all objects are allocated into. To my understanding this is done to make allocation faster by always having the allocator reference to the beginning of Eden after a collection.

By using the copying method, garbage collection achieves defragmentation without seeking for dead memory blocks. However, this method proves itself to be more efficient in areas where most objects are garbage, so it is not a good approach to take on the old generation memory area. Indeed, that area is still collected using the compacting algorithm – but now, thanks to the separation of young and old generations, it is done in much larger intervals.

Chapter 5 – Java Collection Framework

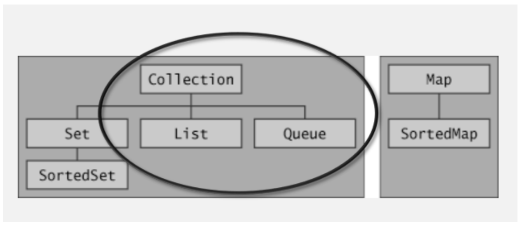
Java Collection versus Java Collections

Java Collection

It's a root interface in the collections hirerachy.

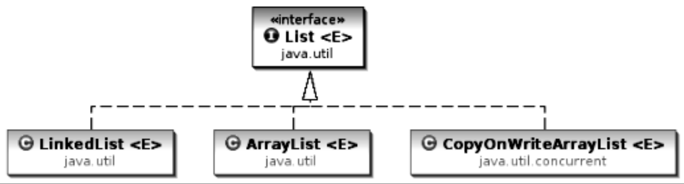
Java Collections

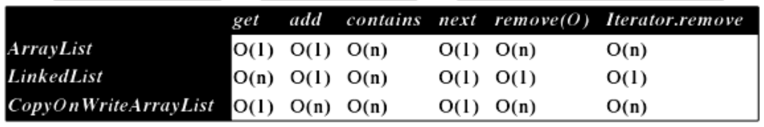
It implements the polymorphic algorithms that operate on the collections. It means the same method can be used on many different implementations of the appropriate Collection interface. (Collections.synchronizeMap(collection), Collections.sort(collection), Collections.unmodifiableList(collection), Collections.unmodifiableMap(collection), etc.)



List Interface

- The List interface provides support for **ordered collections** of objects.
- Lists may contain **duplicate** elements.





	get	*add*	*contains*	*next*	*remove(O)*	*Iterator.remove*
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

Main Implementations of List Interface

1. **ArrayList:** Resizable-array implementation of the List interface. It’s the best all-around implementation of the List interface.

2. **Vector:** Synchronized resizable-array implementation of the List interface with additional "legacy methods."

3. **LinkedList:** Doubly-linked list implementation of the List interface. It may provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list. It’s useful for queues and double-ended queues (deques).

Advantages of ArrayList over Array

Some of the advantages ArrayList has over Array are:

1. It can grow dynamically
2. It provides more powerful insertion and search mechanisms than arrays

Differences between ArrayList and Array

ArrayList	Array
It stores only objects	It can store primitive values and objects
It can grow dynamically and re-sizable	It has fixed size
It can be only single dimensional	It can have multi dimensional
It resides in the Collection framework (java.util)	It resides in Java core package (java.lang)

Differences between ArrayList and Vector

ArrayList	Vector
ArrayList is NOT synchronized by default	Vector List is synchronized by default
ArrayList can use only Iterator to access the elements	Vector list can use Iterator and Enumeration Interface to access the elements
The ArrayList increases its array size by 50 percent if it runs out of room	A Vector defaults to doubling the size of its array if it runs out of room
ArrayList has no default size	While vector has a default size of 10

Conversion from ArrayList to Array and Vice-versa

ArrayList to Array	Array to ArrayList
List<Element> arrayList = new ArrayList<Element>(); Object[] array = arrayList.toArray(new Object[arrayList.size]);	List<Element> arrayList = new ArrayList<Element>(Arrays.asList(array))

ArrayList versus LinkedList

1. ArrayList internally uses and array to store the elements, when that array gets filled by inserting elements a new array of roughly 1.5 times the size of the original array is created and all the data of old array is copied to new array. During deletion, all elements present in the array after the deleted elements have to be moved one step back to fill the space created by deletion.

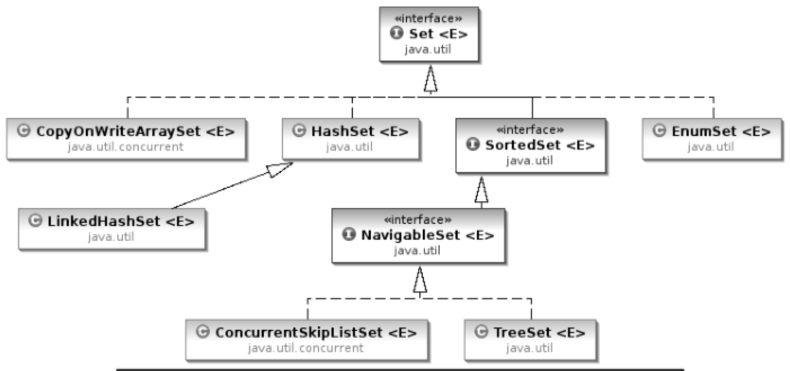
2. In linked list data is stored in nodes that have reference to the previous node and the next node so adding element is simple as creating the node an updating the next pointer on the last node and the previous pointer on the new node. Deletion in linked list is fast because it involves only updating the next pointer in the node before the deleted node and updating the previous pointer in the node after the deleted node.

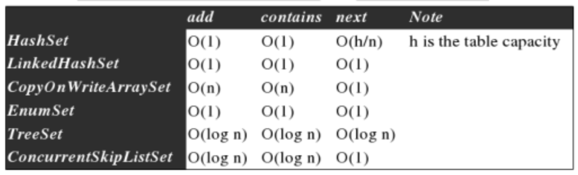
3. If you need to support **random access**, without inserting or removing elements from any place other than the end, then ArrayList offers the optimal collection.

4. If, however, you need to **frequently add and remove elements** from the middle of the list and only access the list elements **sequentially**, then LinkedList offers the better implementation.

Set Interface

- The Set interface provides methods for accessing the elements of a finite mathematical set. It provides an **unordered collection** of objects.
- Sets do not **allow duplicate elements**
- Contains no methods other than those inherited from Collection
- It adds the restriction that duplicate elements are prohibited
- Two Set objects are equal if they contain the same elements





	add	*contains*	*next*	*Note*
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

Main Implementations of Set Interface

1. HashSet
2. TreeSet
3. LinkedHashSet
4. EnumSet

HashSet

- A HashSet is an unsorted, unordered Set.
- It uses the hashcode of the object being inserted (so the more efficient your hashcode() implementation the better access performance you'll get).
- Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

TreeSet

TreeSet is a SortedSet implementation that keeps the elements in sorted order. The elements are sorted according to the natural order of elements or by the comparator provided at creation time.

EnumSet

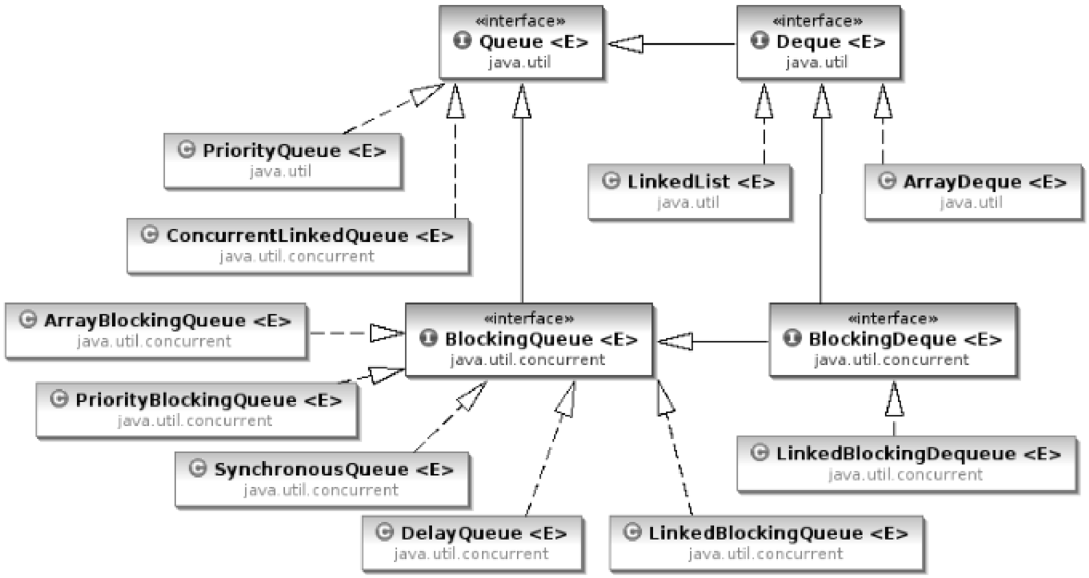
An EnumSet is a specialized set for use with enum types, all of the elements in the EnumSet type that is specified, explicitly or implicitly, when the set is created.

Differences between HashSet and TreeSet

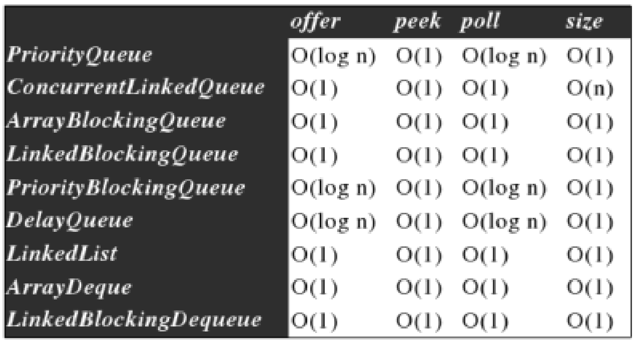
HashSet	**TreeSet**
HashSet is under Set interface. So, it does not guarantee for either sorted order or sequence order.	TreeSet is under SortedSet interface. So, it provides elements in a sorted order (natural - asceding order).
We can add any type of elements to HashSet.	We can add only similar types of elements to TreeSet.

Queue Interface

Stack is a data structure that is based on Last in First out (**LIFO**) rule, while Queues are based on First in First out (**FIFO**) rule.



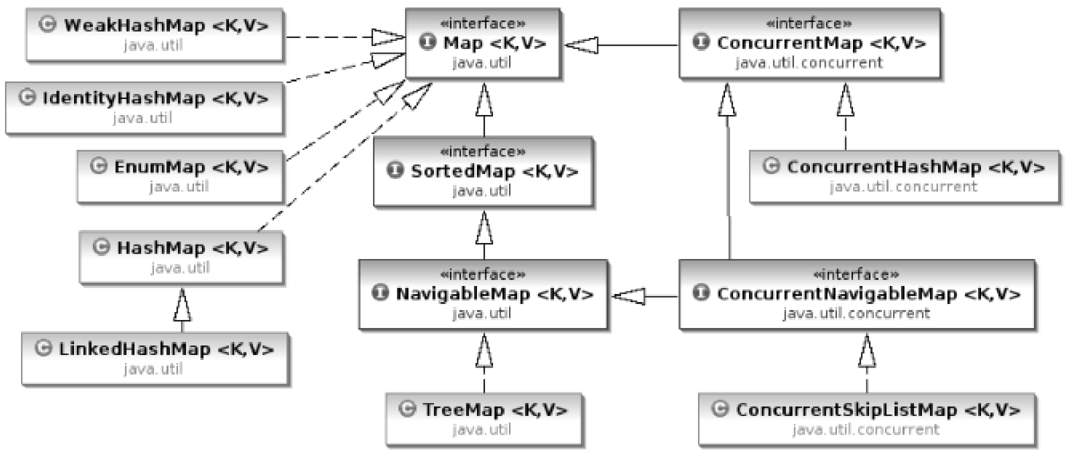
«interface» Queue <E> java.util
«interface» Deque <E> java.util
PriorityQueue <E> java.util
ConcurrentLinkedQueue <E> java.util.concurrent
LinkedList <E> java.util
ArrayDeque <E> java.util
ArrayBlockingQueue <E> java.util.concurrent
«interface» BlockingQueue <E> java.util.concurrent
«interface» BlockingDeque <E> java.util.concurrent
PriorityBlockingQueue <E> java.util.concurrent
SynchronousQueue <E> java.util.concurrent
LinkedBlockingDequeue <E> java.util.concurrent
DelayQueue <E> java.util.concurrent
LinkedBlockingQueue <E> java.util.concurrent

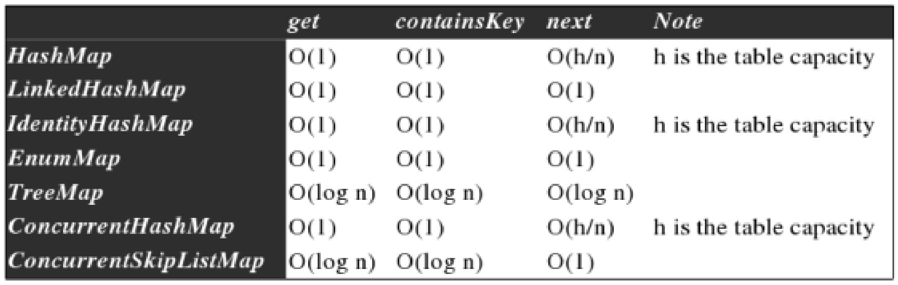


	offer	*peek*	*poll*	*size*
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDequeue	O(1)	O(1)	O(1)	O(1)

Map Interface

- A map is an object that stores associations between **keys and values** (key/value pairs).
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Some maps can accept a null key and null values, others cannot.
- The key class of Map should override **equals** and **hashcode** methods of Object superclass.





	get	*containsKey*	*next*	*Note*
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

Main Implementations of Map

1. HashMap
2. HashTable
3. TreeMap
4. EnumMap

TreeMap

TreeMap actually implements the SortedMap interface which extends the Map interface. In a TreeMap the data will be sorted in ascending order of keys according to the natural order for the key's class, or by the comparator provided at creation time. TreeMap is based on the Red-Black tree data structure.

HashMap versus TreeMap

- For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative.
- If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative.
- Depending upon the size of your collection, it may be faster to add elements to a HashMap, and then convert the map to a TreeMap for sorted key traversal.

Differences between HashMap and HashTable

HashMap	**HashTable**
HashMap lets you have null values as well as one null key. Only one NULL is allowed as a key in HashMap. HashMap does not allow multiple keys to be NULL. Nevertheless, it can have multiple NULL values.	HashTable does not allow null values as key and value. The **properties** class is a subclass of HashTable that can be read from or written to a stream.
The iterator in the HashMap is fail-safe.	The enumerator for the Hashtable is not fail-safe.
HashMap is NOT synchronized.	Hashtable is synchronized.

Collection Views provided by Map

1. **Key Set -** allow a map's contents to be viewed as a set of keys. KeySet is a set returned by the **keySet ()** method of the Map interface.
2. **Values Collection -** allow a map's contents to be viewed as a set of values. Values Collection View is a collection returned by the **values ()** method of the Map interface.
3. **Entry Set -** allow a map's contents to be viewed as a set of key-value mappings. Entry Set view is a set that is returned by the **entrySet ()** method in the Map interface.

Traverse Collection

There are two ways to traverse collections:

1. Using Iterator or Enumeration
2. Using Enhanced For Loop

Iterator interface provides functions for:

Creating the data structure

- add(e)
- addAll(c)

Querying the data structure

- size()
- isEmpty()
- contains(e)
- containsAll(c)
- toArray()
- equals(e)

Modifying the data structure

- remove(e)
- removeAll(c)
- retainAll(c)
- clear()

Iterator

- The Iterator interface is used to step through the elements of a Collection.
- Iterators let you process each element of a Collection.
- Iterators are a generic way to go through all the elements of a Collection no matter how it is organized.
- Iterator is an Interface implemented a different way for every Collection.
- To use an iterator to traverse through the contents of a collection, follow these steps:
- Obtain an iterator to the start of the collection by calling the collection's **iterator**() method.
- Set up a loop that makes a call to **hasNext**(). Have the loop iterate as long as hasNext() returns true.
- Within the loop, obtain each element by calling **next**().
- Iterator also has a method **remove**() when remove is called, the current element in the iteration is deleted.

Fail Fast/Safe Iterator

Because, if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Differences between Enumeration and Iterator

Enumeration	Iterator
Enumeration doesn't have a remove() method	Iterator has a remove() method
Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects. So, Enumeration is used whenever we want to make Collection objects as Read-Only.	Iterator can be used to edit the collection by adding or removing elements. It can be abstract, final, native, static, or synchronized.

ListIterator

ListIterator is just like Iterator, except it allows us to access the collection in either the **forward or backward direction** and lets us modify an element.

Sorting Mechanisms

Collections can use either Comparable or Comparator interfaces to define the sorting order.

Comparable Interface

The Comparable interface is used to sort collections and arrays of objects using the **Collections.sort()** and **java.utils.Arrays.sort()** methods respectively. The objects of the class implementing the Comparable interface can be ordered.

The Comparable interface in the generic form is written as follows:

```
interface Comparable<T>
```

where T is the name of the type parameter.

All classes implementing the Comparable interface must implement the compareTo() method that has the return type as an integer. The signature of the compareTo() method is as follows:

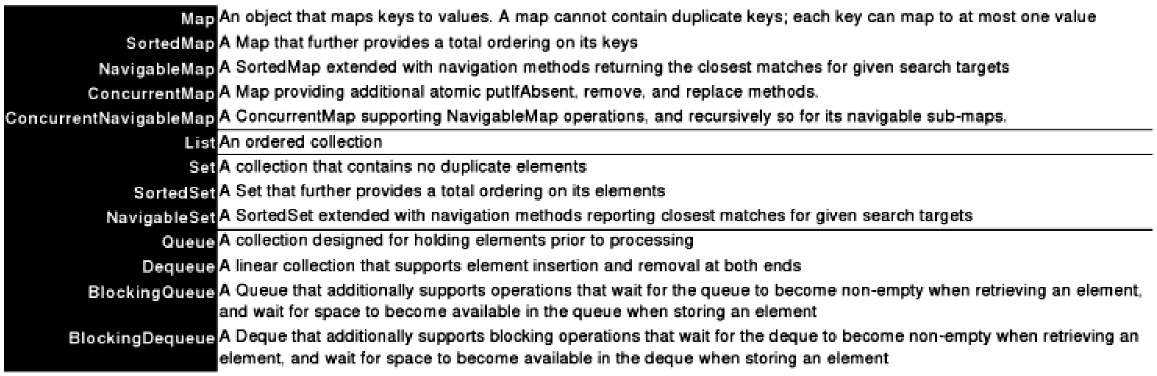
```
int i = object1.compareTo(object2)
```

- If object1 < object2: The value of i returned will be negative.
- If object1 > object2: The value of i returned will be positive.
- If object1 = object2: The value of i returned will be zero.

Differences between Comparable and Comparator Interfaces

Comparable	Comparator
It uses the compareTo() method. int objectOne.compareTo(objectTwo).	It uses the compare() method. int compare(ObjOne, ObjTwo)
It is necessary to modify the class whose instance is going to be sorted.	A separate class can be created in order to sort the instances.
Only one sort sequence can be created.	Many sort sequences can be created.
It is frequently used by the API classes.	It used by third-party classes to sort instances.

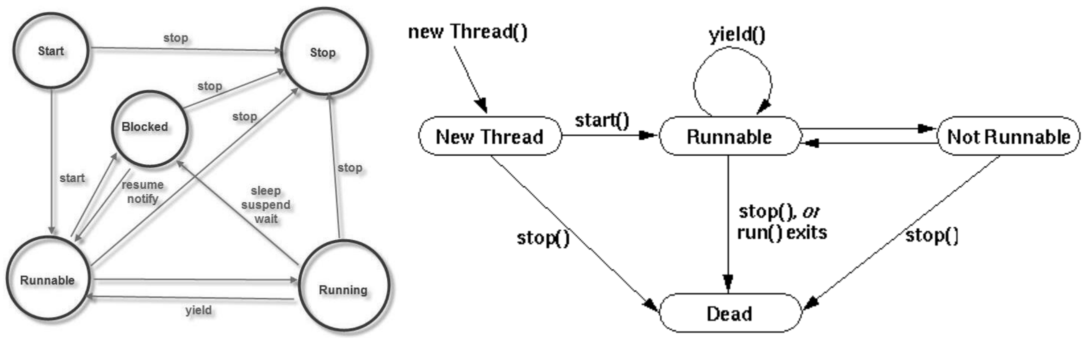
Summary



Map	An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value
SortedMap	A Map that further provides a total ordering on its keys
NavigableMap	A SortedMap extended with navigation methods returning the closest matches for given search targets
ConcurrentMap	A Map providing additional atomic putIfAbsent, remove, and replace methods.
ConcurrentNavigableMap	A ConcurrentMap supporting NavigableMap operations, and recursively so for its navigable sub-maps.
List	An ordered collection
Set	A collection that contains no duplicate elements
SortedSet	A Set that further provides a total ordering on its elements
NavigableSet	A SortedSet extended with navigation methods reporting closest matches for given search targets
Queue	A collection designed for holding elements prior to processing
Dequeue	A linear collection that supports element insertion and removal at both ends
BlockingQueue	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element
BlockingDequeue	A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element

Chapter 6 – Java Threading Model

Java Threads



Java Thread Lifecycle

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

Provide a Runnable object
The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {
   public void run() {
      System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
      (new Thread(new HelloRunnable())).start();
   }
}
```

Subclass Thread

The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread {
   public void run() {
      System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
      (new HelloThread()).start();
   }
}
```

Notice that both examples invoke Thread.start in order to start the new thread. The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.

Concurrency and Multi-Threading in Java

It covers the concepts of parallel programming, immutability, threads, the executor framework (thread pools), futures, callables and the fork-join framework.

Concurrency

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improve the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

Process versus Thread

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A thread is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Concurrency Issues

Threads have their own call stack, but can also access shared data. Therefore you have two basic problems, visibility and access problems.

A visibility problem occurs if thread A reads shared data which is later changed by thread B and thread A is unaware of this change.

An access problem can occur if several thread access and change the same shared data at the same time.

Visibility and access problem can lead to

Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.

Safety failure: The program creates incorrect data.

Chapter 7 - Unified Modeling Language (UML)

Static Diagrams (Structural)

1. Use Case
2. Class

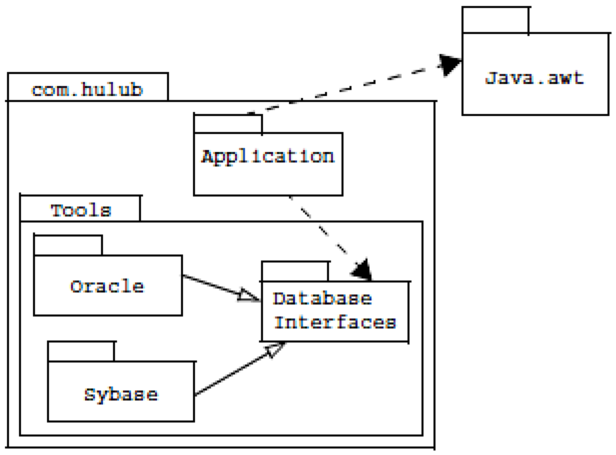
Dynamic Diagrams (Behavioral)

1. Object
2. Sequence (Interaction)
3. Collaboration (Interaction)
4. State Chart
5. Activity

Implementation Diagrams

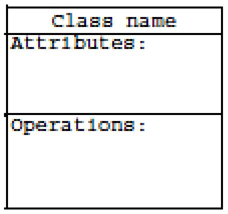
1. Component
2. Deployment

Packages



- C++ **namespace**.
- Group together functionally-similar classes.
- Derived classes need not be in the same package.
- Packages can nest. Outer packages are sometimes called **domains**. (In the diagram, "Tools" is arguably an outer package, not a domain).
- Package name is part of the class name (e.g. given the class *fred* in the flintstone package, the **fully-qualified** class name is *flintstone.fred*).
- Generally needed when entire static-model won't fit on one sheet.

Classes (Box contains three compartments)



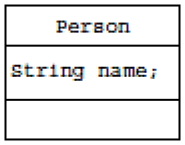
1. The name compartment (required) contains the class name and other documentation-related information:

E.g.:

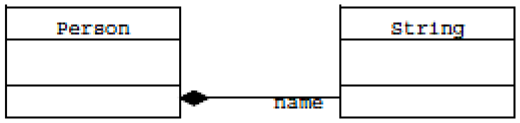
<u>**Some_class**</u> «abstract»
{ author: George Jetson
modified: 10/6/2999
checked_out: y
}

- Guillemets identify stereotypes. E.g.: «static», «abstract» «JavaBean». Can use graphic instead of word.
- Access privileges (see below) can precede name.
- Inner (nested) classes identify outer class as prefix of class name: (**Outer.Inner** or **Outer::Inner**).

2. The *attributes compartment* (optional):
   - *During Analysis*: identify the attributes (i.e. defining characteristics) of the object.
   - *During Design*: identify a relationship to a stock class.
     This:



is a more compact (and less informative) version of this:



Everything, here, is private. Always. Period.

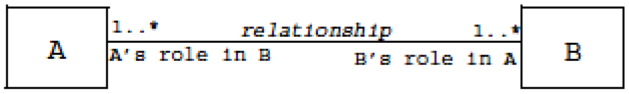
3. The *operations compartment* (optional) contains method definitions. Use implementation-language syntax, except for **access privileges**:

+	`public`
#	`protected`
-	`private`
~	`package (my extension to UML)¹`

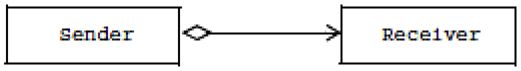
- Abstract operations (C++ virtual, Java non-final) indicated by *italics* (or <u>underline</u>).
- **Boldface** operation names are easier to read.

If attributes and operations are both omitted, a more complete definition is assumed to be on another sheet.

Associations (relationships between classes)



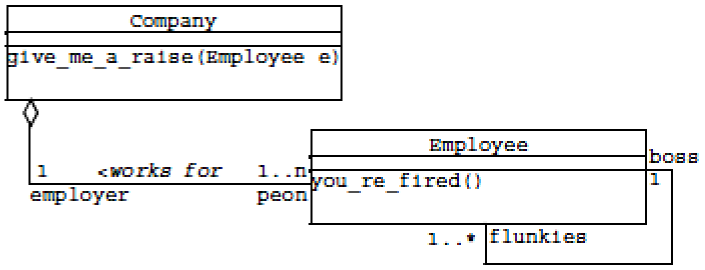
- Associated classes are connected by lines.
- The relationship is identified, if necessary, with a < or > to indicate direction (or use solid arrowheads).
- The role that a class plays in the relationship is identified on that class's side of the line.
- Stereotypes (like «friend») are appropriate.
- Unidirectional message flow can be indicated by an arrow (but is implicit in situations where there is only one role):



- Cardinality:

1	Usually omitted if 1:1
n	Unknown at compile time, but bound.
0..1	(1..2 1..n)
1..*	1 or more
*	0 or more

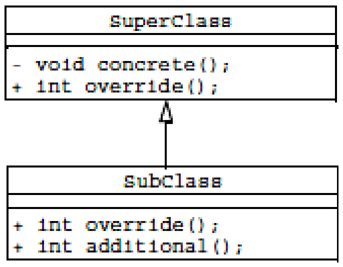
- Example:



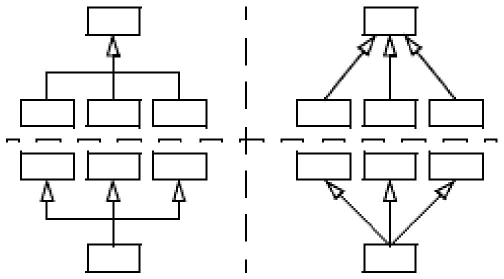
```
class Company
{
  private Employee[] peon = new Employee[n];
  public void give_me_a_raise( Employee e ) { ... }
}

class Employee
{
   private Company  employer;
   private Employee boss;
   private Vector   flunkies = new Vector();
   public  void     you_re_fired() { ... }
}
```

(A Java Vector is a variable-length array. In this case it will hold Employee objects)

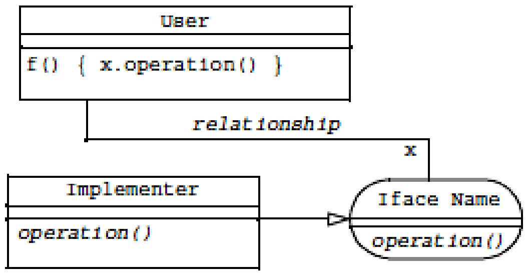
Implementation Inheritance



Outline arrows identify derivation relationships: extends, implements, is-a, has-properties-of, etc. Variations include:

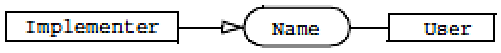


Interface Inheritance

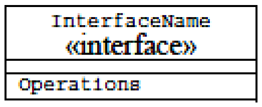


In C++, an interface is a class containing nothing but pure virtual methods. Java supports them directly (c.f. "abstract class," which can contain method and field definitions in addition to the abstract declarations.)

My extension to UML: rounded corners identify interfaces. If the full interface specification is in some other diagram, I use:



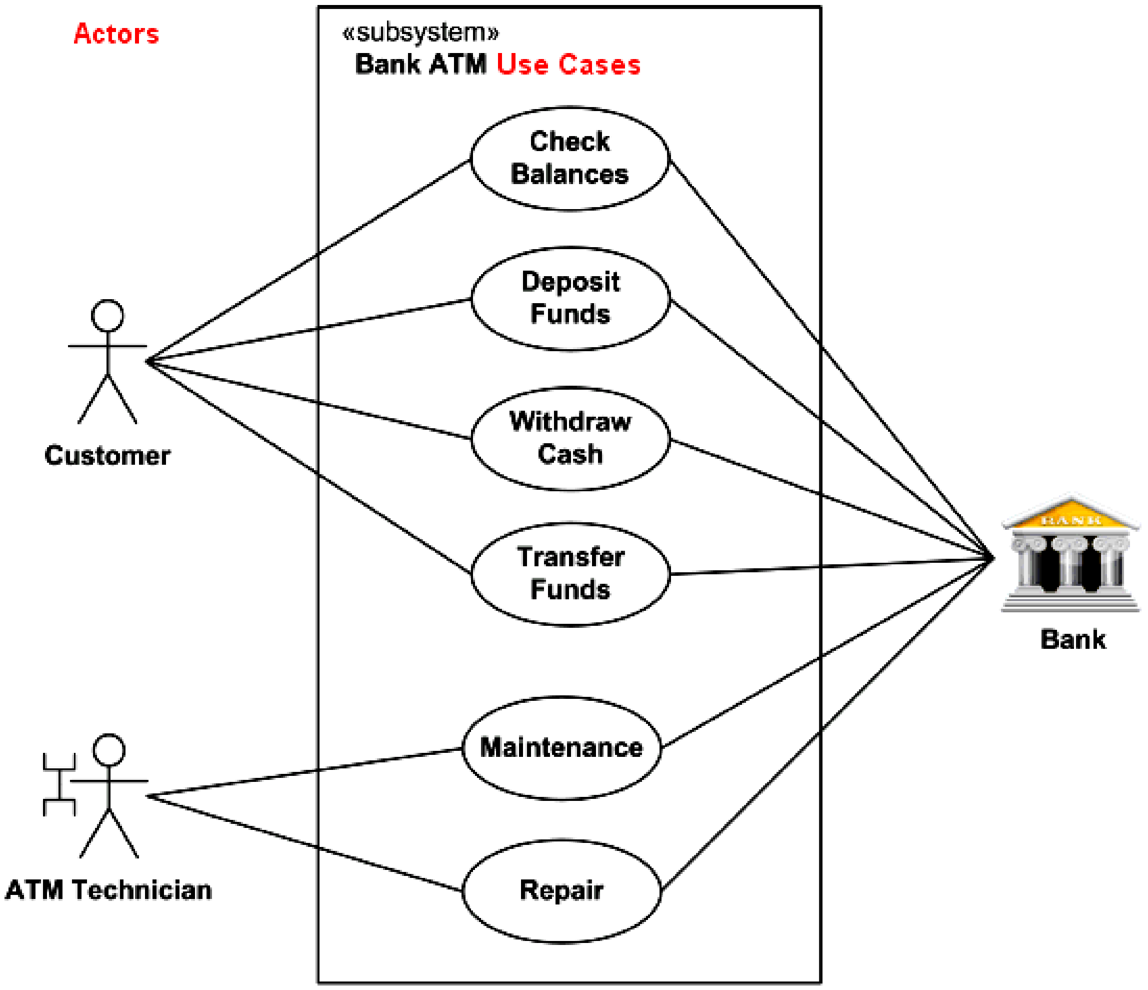
Strict UML uses the «**interface**» stereotype in the name compartment of a standard class box:



Interfaces contain no attributes, so the attribute compartment is always empty.

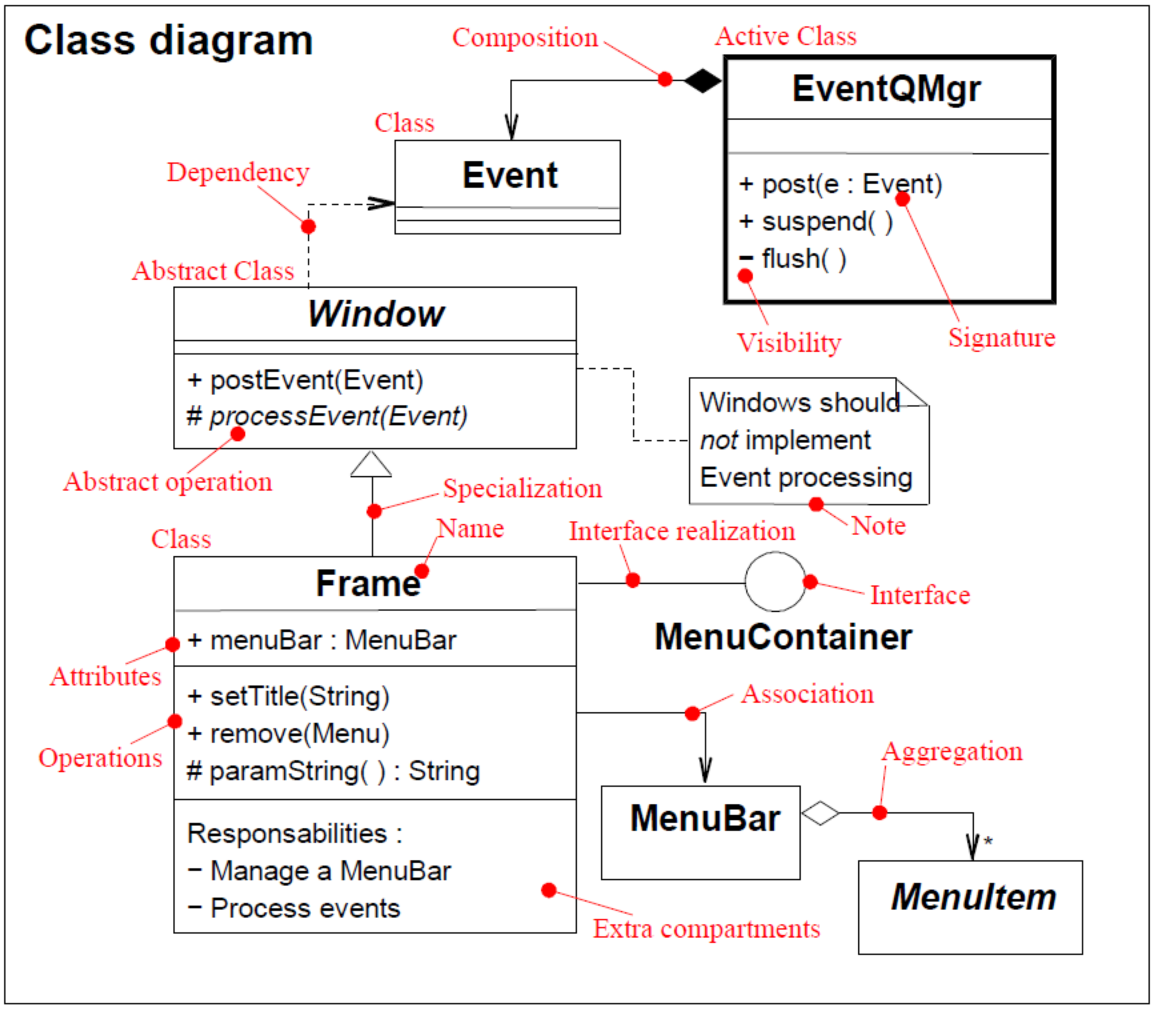
Static Diagrams – 1: Use Case (Structural)

Use case modeling is an approach to capture user requirements. A use-case is a sequence of interactions between the user and the system under consideration to achieve a goal. Use case diagrams model the functionality of a system using actors and use cases. Use cases are services or functions provided by the system to its users.



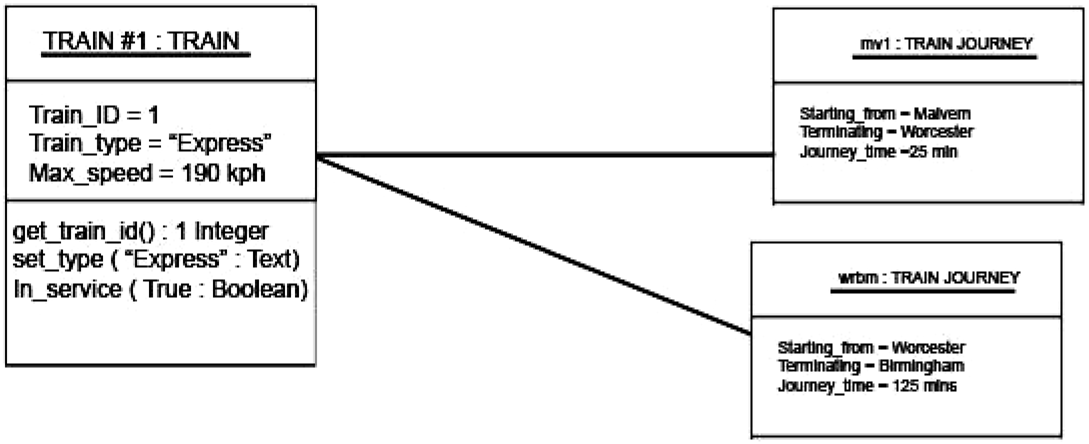
Static Diagrams – 2: Class (Structural)

A class diagram is a graphical-notation for depicting the system's classes along with their relationship to one another. Classes relate to each other through different forms of association.

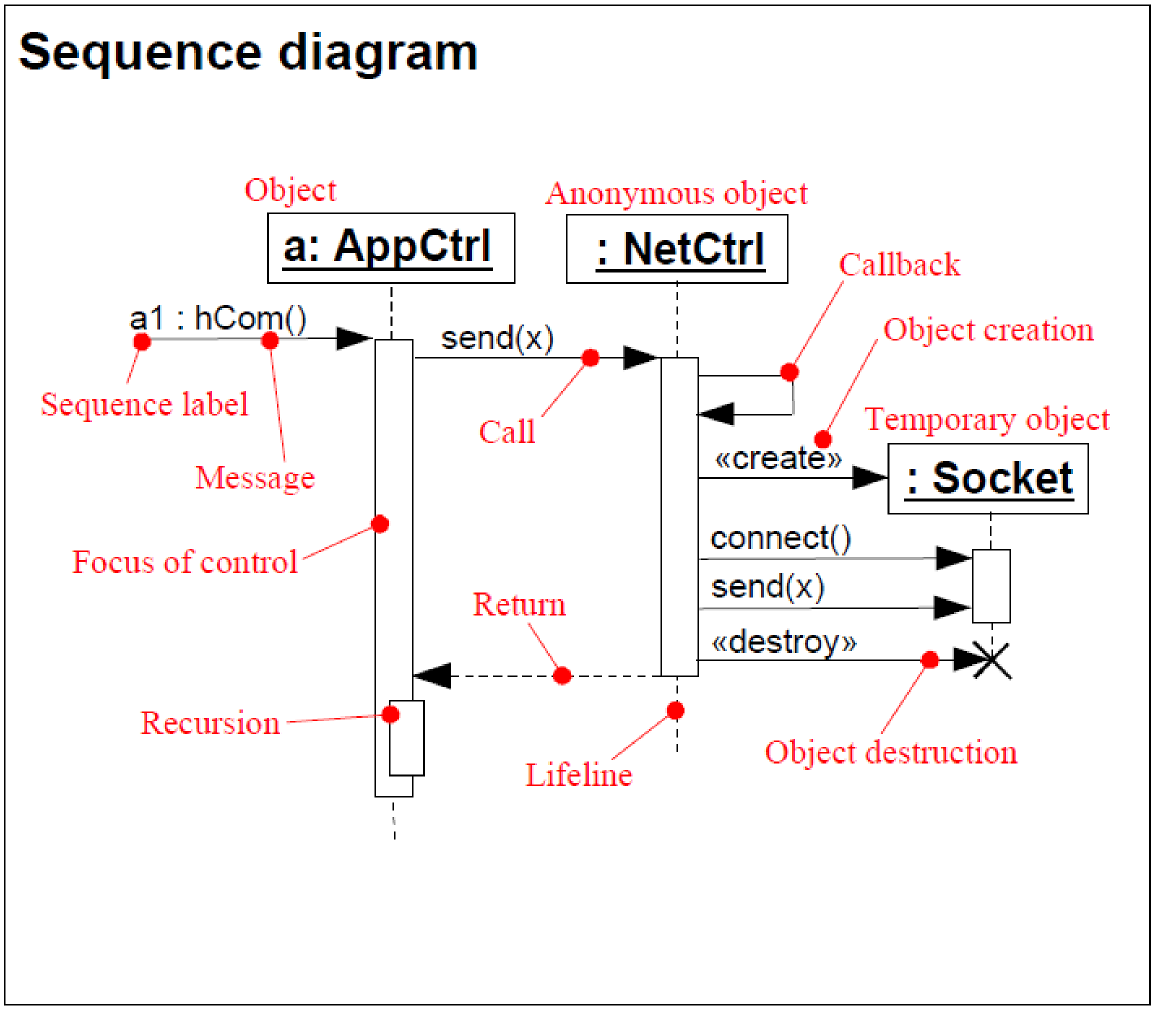


Dynamic Diagrams – 1: Object (Behavioral)

Object diagrams are also closely linked to class diagrams. Just as an object is an instance of a class, an object diagram could be viewed as an instance of a class diagram. Object diagrams describe the static structure of a system at a particular time and they are used to test the accuracy of class diagrams.

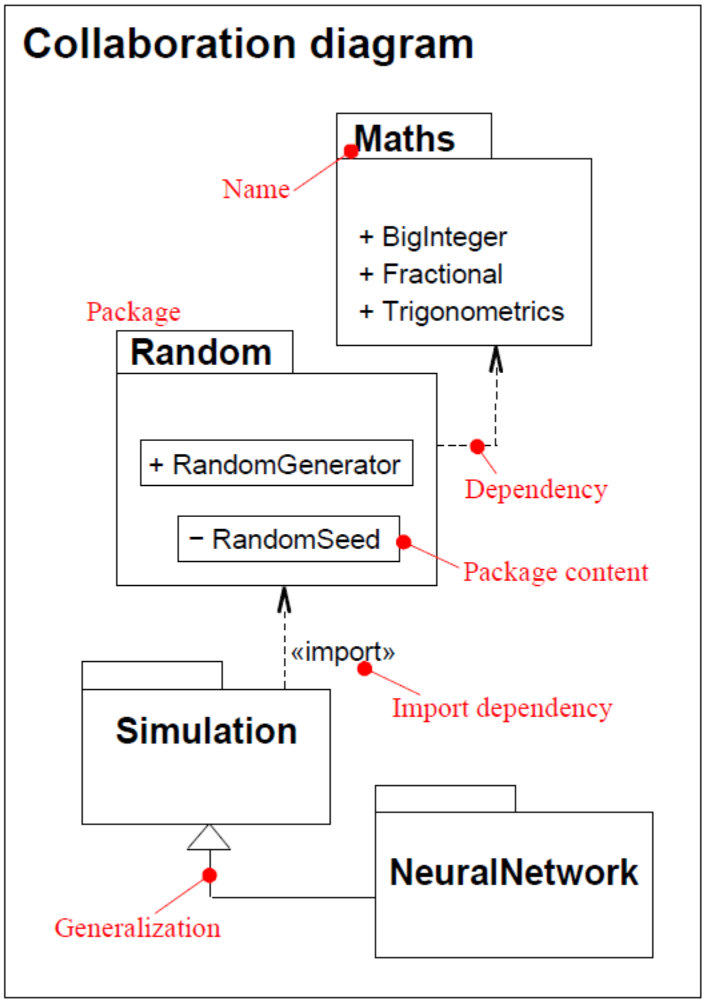


Sequence diagrams describe interactions among classes in terms of an exchange of messages over time. Sequence diagrams model the dynamic aspects of a software system. The emphasis is on the “sequence” of messages rather than relationship between objects. A sequence diagram maps the flow of logic or flow of control within a usage scenario into a visual diagram enabling the software architect to both document and validate the logic during the analysis and design stages.

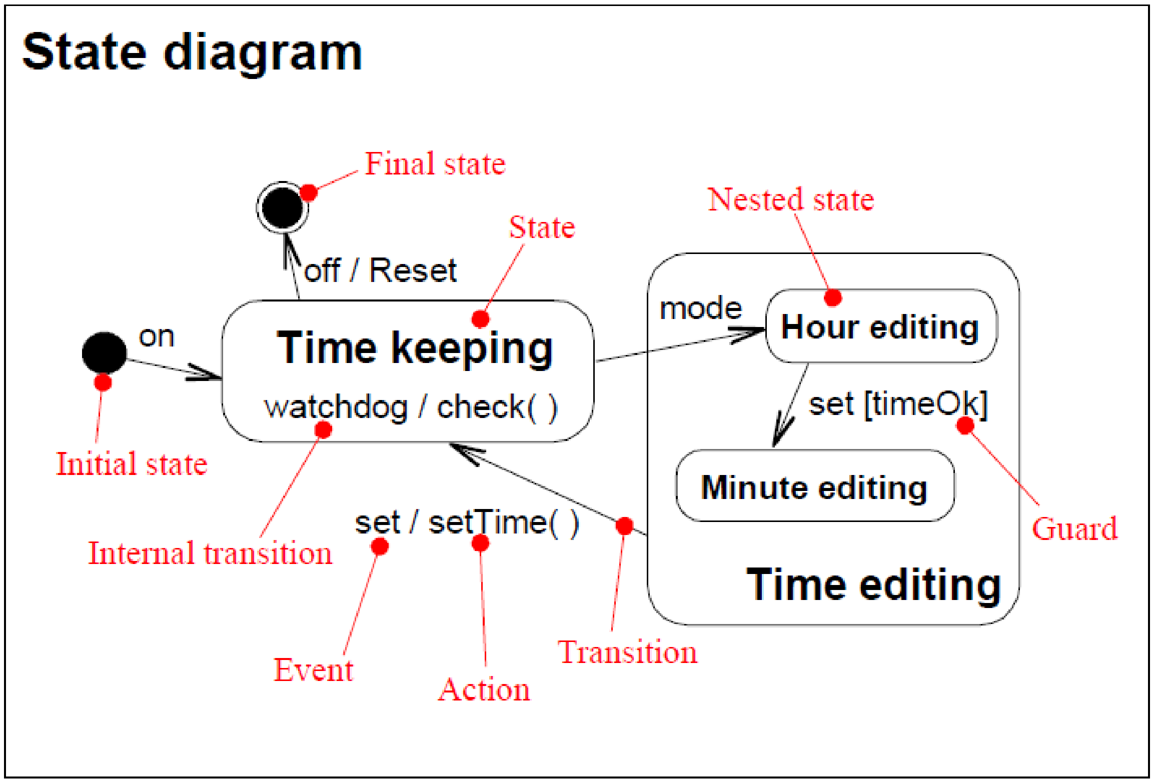


Dynamic Diagrams – 3: Collaboration (Behavioral) - Interaction

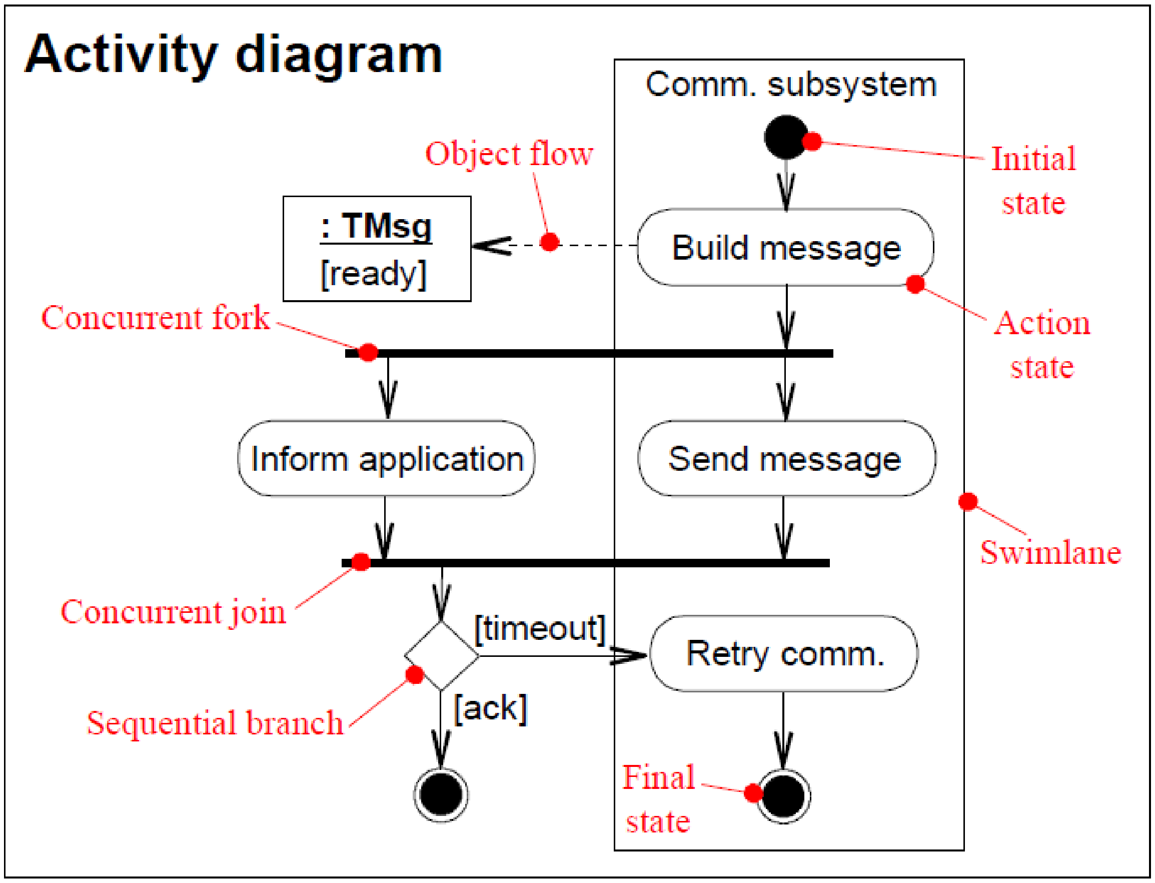
A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.



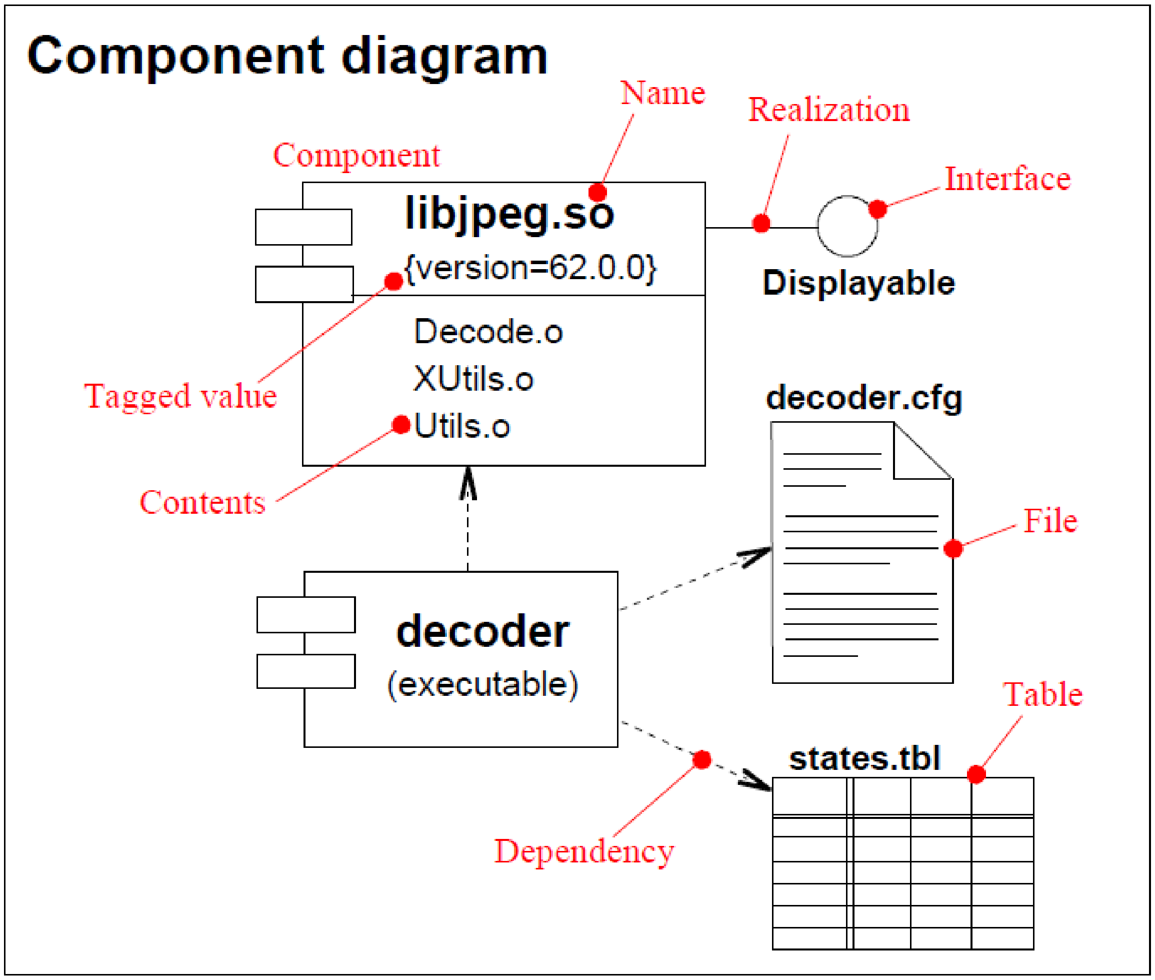
A state chart diagram is a graph in which nodes correspond to states and directed arcs correspond to transitions labeled with event names. **A state diagram combines states and events in the form of a network to model all possible object states during its life cycle, helping to visualize how an object responds to different stimuli.** A state chart diagram shows the behavior of classes in response to external stimuli. This diagram models the dynamic flow of control from state to state within a system.



An activity diagram is a type of flow chart with additional support for parallel behavior. Activity diagrams include the following new concepts. Branches and Merges model the conditional behavior of activity diagrams. A branch has a single incoming transition and multiple, conditional, outgoing transitions. The control flow is directed to one of the outgoing transitions depending on the condition satisfied. A merge is a node in the activity diagram at which the conditional behavior terminates. Each branch in an activity diagram has a corresponding merge. An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation. Because an activity diagram is a special kind of state chart diagram, it uses some of the same modeling conventions.

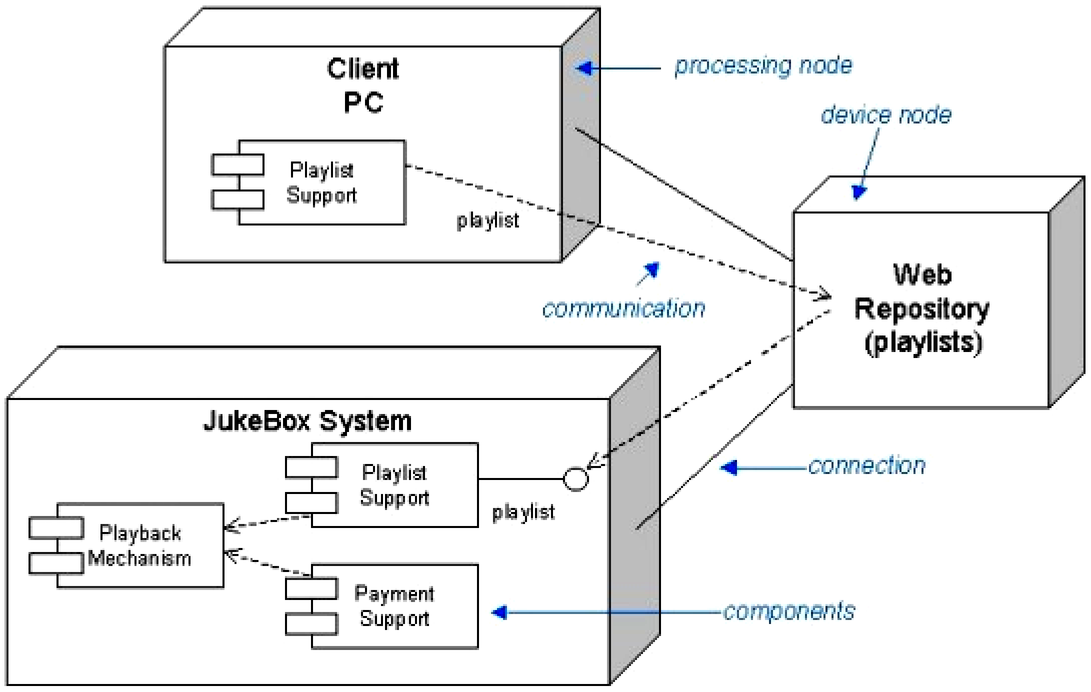


A component diagram describes the organization of the physical components in a system.



Implementation Diagrams – 2: Deployment Diagram

Deployment diagrams depict the physical resources in a system including nodes, components, and connections.



Chapter 8 - Structured Query Language (SQL)

Introduction

SQL (pronounced as Sequel), referred to as Structured Query Language is a programming language designed for managing data in relational database management systems (RDBMS).

The SQL language is subdivided into several language elements, including:

- **Clauses**, which are constituent components of statements and queries. (In some cases, these are optional.)[11]
- **Expressions**, which can produce either scalar values or tables consisting of columns and rows of data.
- **Predicates**, which specify conditions that can be evaluated to SQL three-valued logic (3VL) or Boolean (true/false/unknown) truth values and which are used to limit the effects of statements and queries, or to change program flow.
- **Queries**, which retrieve the data based on specific criteria. This is the most important element of SQL.
- **Statements**, which may have a persistent effect on schemata and data, or which may control transactions, program flow, connections, sessions, or diagnostics. SQL statements also include the semicolon (";") statement terminator. Though not required on every platform, it is defined as a standard part of the SQL grammar.
- **Insignificant whitespace** is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

Classification

The SQL statements are classified as follows:

1) Data Definition Language (DDL)

These statements are used to define the database structure or schema. It manages table and index structure. Some examples are:

1. CREATE - to create objects in the database
2. ALTER - alters the structure of the database
3. DROP - delete objects from the database

4. TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
5. COMMENT - add comments to the data dictionary
6. RENAME - rename an object

2) Data Manipulation Language (DML)

These statements are used for managing data within schema objects. It is the subset of SQL used to add, update and delete data. Some examples are:

1. SELECT - retrieve data from the a database
2. INSERT - insert data into a table
3. UPDATE - updates existing data within a table
4. DELETE - deletes all records from a table, the space for the records remain
5. MERGE - UPSERT operation (insert or update)
6. CALL - call a PL/SQL or Java subprogram
7. EXPLAIN PLAN - explain access path to data
8. LOCK TABLE - control concurrency

3) Data Control Language (DCL)

The Data Control Language (DCL) authorizes users and groups of users to access and manipulate data. Its two main statements are:

1. GRANT - authorizes one or more users to perform an operation or a set of operations on an object.
2. REVOKE - eliminates a grant, which may be the default grant.

4) Transaction Control Language (TCL)

These statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

1. COMMIT - Persist the changes
2. SAVEPOINT - Identify a point in a transaction to which you can later roll back
3. ROLLBACK - Restore database to original since the last COMMIT
4. SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

Normalization – 3 Normal Forms

The purpose is to

- Remove Redundancy
- Maintain Data Integrity
- Improve Efficiency

First Normal Form (1NF) - No repeating attributes or group of attributes (No repeating columns)

Second Normal Form (2NF) - All non-key attributes must depend on the whole key (No repeating rows)

Third Normal Form (3NF) - No dependencies on non-key attributes (No transitive functional dependencies)

Cartesian product

Also referred to as a cross-join product, returns all the rows in all the tables listed in the query. Each row in the first table is paired with all the rows in the second table. This happens when there is no relationship defined between the two tables.

Joins

SQL Joins are used to relate information in different tables. A Join condition is a part of the SQL query that retrieves rows from two or more tables. A SQL Join condition is used in the SQL WHERE clause of SELECT, UPDATE and DELETE statements. SQL Joins can be classified into Equi joins and Non Equi joins.

SQL Equi Join
It is a simple sql join condition which uses the equal sign as the comparison operator.
Equi Join is further classified into two categories:

1. **SQL Inner Join**
   All the rows returned by the sql query satisfy the sql join condition specified.

2. **SQL Outer Join**
   This sql join condition returns all rows from both tables which satisfy the join condition along with rows which do not satisfy the join condition from one of the tables. The sql outer join operator in Oracle is (+) and is used on one side of the join condition only. They are further classified into left and right outer joins.

   Left Outer Join
   If the (+) operator is used in the left side of the join condition it is equivalent to left outer join.

   Right Outer Join
   If used on the right side of the join condition it is equivalent to right outer join.

SQL Non-Equi Join
It is a sql join condition which makes use of some comparison operator other than the equal sign like >, <, >=, <=.

Self-Join
A Self Join is a type of sql join which is used to join a table to itself, particularly when the table has a FOREIGN KEY that references its own PRIMARY KEY. It is necessary to ensure that the join statement defines an alias for both copies of the table to avoid column ambiguity.

Visit http://beginner-sql-tutorial.com/ and http://www.plsql-tutorial.com/

SELECT Statement
```
SELECT [DISNCT] {*, column [alias],...}
   FROM table
   [WHERE condition(s)]
   [ORDER BY {column, exp, alias} [ASC|DESC]]
```
Cartesian Product
```
SELECT table1.*, table2.*,[...]
   FROM table1,table2[,...]
```
Equijoin(Simple joins or inner join)
```
SELECT table1.*,table2.*
   FROM  table1,table2
   WHERE table1.column = table2.column
```
Non-Equijoins
```
SELECT table1.*, table2.*
   FROM table1, table2
   WHERE table1.column
   BETWEEN table2.column1 AND table2.column2
```
Outer joins
```
SELECT table1.*,table2.*
   FROM  table1,table2
   WHERE table1.column(+) = table2.column
SELECT table1.*,table2.*
   FROM  table1,table2
   WHERE table1.column = table2.column(+)
```
Self joins
```
SELECT alias1.*,alias2.*
   FROM  table1 alias1,table1 alias2
   WHERE alias1.column = alias2.column
```
Aggregation Selecting
```
SELECT [column,] group_function(column)
   FROM table
   [WHERE condition]
   [GROUP BY group_by_expression]
   [HAVING group_condition]
   [ORDER BY column] ;
```
Group function
```
AVG([DISTINCT|ALL]n)
COUNT(*|[DISTINCT|ALL]expr)
MAX([DISTINCT|ALL]expr)
MIN([DISTINCT|ALL]expr)
STDDEV([DISTINCT|ALL]n)
SUM([DISTINCT|ALL]n)
VARIANCE([DISTINCT|ALL]n)
```
Subquery
```
SELECT select_list
   FROM table
   WHERE expr operator(SELECT select_list FROM table);
single-row comparison operators
        =    >    >=   <    <=    <>
multiple-row comparison operators
        IN    ANY    ALL
```
Multiple-column Subqueries
```
SELECT column, column, ...
   FROM table
   WHERE (column, column, ...) IN
          (SELECT column, column, ...
           FROM table
           WHERE condition) ;
```
Manipulating Data

INSERT Statement(one row)
```
INSERT INTO table [ (column [,column...])]
   VALUES          (value [,value...]) ;
```
INSERT Statement with Subquery
```
INSERT INTO table [ column(, column) ]
        subquery ;
```
UPDATE Statement
```
UPDATE table
   SET column = value [, column = value,...]
   [WHERE condition] ;
```
Updating with Multiple-column Subquery
```
UPDATE table
   SET (column, column,...) =
        (SELECT column, column,...
         FROM table
         WHERE condition)
   WHERE condition ;
```
Deleting Rows with DELETE Statement
```
DELETE [FROM] table
   [WHERE conditon] ;
```
Deleting Rows Based on Another Table
```
DELETE FROM table
   WHERE column = (SELECT column
                   FROM table
                   WHERE condtion) ;
```
Transaction Control Statements
```
COMMIT ;
SAVEPOINT name ;
ROLLBACK [TO SAVEPOINT name] ;
```
CREATE TABLE Statement
```
CREATE TABLE [schema.]table
        (column datatype [DEFAULT expr] [,...]) ;
```
CREATE TABLE Statement with Subquery
```
CREATE TABLE [schema.]table
        [(column, column...)]
   AS subquery
```
Datatype
```
VARCHAR2(size) CHAR(size)      NUMBER(p,s)    DATE
LONG           CLOB            RAW            LONG RAW
BLOB           BFILE
```
ALTER TABLE Statement (Add columns)
```
ALTER TABLE table
   ADD  (column datatype [DEFAULT expr]
        [, column datatype]...) ;
```
Changing a column's type, size and default of a Table
```
ALTER TABLE table
   MODIFY  (column datatype [DEFAULT expr]
           [, column datatype]...) ;
```
Dropping a Table
```
DROP TABLE table ;
```
Changing the Name of an Object
```
RENAME old_name TO new_name ;
```
Trancating a Table
```
TRUNCATE TABLE table ;
```
Adding Comments to a Table
```
COMMENT ON TABLE table | COLUMN table.column
```

```
          IS 'text' ;
```
Dropping a comment from a table
```
COMMENT ON TABLE table | COLUMN table.column IS '' ;
```
Data Dictionary

ALL_OBJECTS	USER_OBJECTS
ALL_TABLES	USER_TABLES
ALL_CATALOG	USER_CATALOG or CAT
ALL_COL_COMMENTS	USER_COL_COMMENTS
ALL_TAB_COMMENTS	USER_TAB_COMMENTS

Defineing Constraints
```
CREATE TABLE [schema.]table
        (column datatype [DEFAULT expr] [NOT NULL]
        [column_constraint],...
        [table_constraint][,...]) ;
```
Column constraint level
```
column [CONSTRAINT constraint_name] constraint_type,
```
Constraint_type
```
PRIMARY KEY    REFERENCES table(column)      UNIQUE
CHECK (codition)
```
Table constraint level(except NOT NULL)
```
column,...,[CONSTRAINT constraint_name]
    constraint_type (column,...),
```
NOT NULL Constraint (Only Column Level)
```
CONSTRAINT table[_column...]_nn NOT NULL ...
```
UNIQUE Key Constraint
```
CONSTRAINT table[_column..]_uk UNIQUE (column[,...])
```
PRIMARY Key Constraint
```
CONSTRAINT table[_column..]_pk PRIMARY (column[,...])
```
FOREIGN Key Constraint
```
CONSTRAINT table[_column..]_fk
    FOREIGN KEY (column[,...])
    REFERENCES table (column[,...])[ON DELETE CASCADE]
```
CHECK constraint
```
CONSTRAINT table[_column..]_ck CHECK (condition)
```
Adding a Constraint(except NOT NULL)
```
ALTER TABLE table
   ADD [CONSTRAINT constraint_name ] type (column) ;
```
Adding a NOT NULL constraint
```
ALTER TABLE table
   MODIFY (column datatype [DEFAULT expr]
   [CONSTRAINT constraint_name_nn] NOT NULL) ;
```
Dropping a Constraint
```
ALTER TABLE table
   DROP CONSTRAINT constraint_name ;
ALTER TABLE table
   DROP PRIMARY KEY | UNIQUE (column) |
   CONSTRAINT constraint_name [CASCADE] ;
```
Disabling Constraints
```
ALTER TABLE table
   DISABLE CONSTRAINT constraint_name [CASCADE] ;
```
Enabing Constraints
```
ALTER TABLE table
   ENABLE CONSTRAINT constraint_name ;
```
Data Dictionary

ALL_CONSTRAINTS	USER_CONSTRAINTS
ALL_CONS_COLUMNS	USER_CONS_COLUMNS

Creating a View
```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
       [(alias[, alias]...)]
   AS subquery
   [WITH CHECK OPTION [CONSTRAINT constraint_name]]
   [WITH READ ONLY] ;
```
Removing a View
```
DROP VIEW view ;
```
CREATE SEQUENCE Statement
```
CREATE SEQUENCE sequence
        [INCREMENT BY n]
        [START WITH n]
        [{MAXVALUE n| NOMAXVALUE}]
        [{MINVALUE n| NOMINVALUE}]
        [{CYCLE | NOCYCLE}]
        [{CACHE [n|20]| NOCACHE}] ;
```
Pseudocolumns
```
sequence.NEXTVAL       sequence.CURRVAL
```
Modifying a Sequence (No START WITH option)
```
ALTER SEQUENCE sequence
        [INCREMENT BY n]
        [{MAXVALUE n| NOMAXVALUE}]
        [{MINVALUE n| NOMINVALUE}]
        [{CYCLE | NOCYCLE}]
        [{CACHE [n|20]| NOCACHE}] ;
```
Removing a Sequence
```
DROP SEQUENCE sequence ;
```
Creating an Index
```
CREATE INDEX index
   ON TABLE (column[,column]...) ;
```
Removing an Index
```
DROP INDEX index ;
```
Synoyms
```
CREATE [PUBLIC] SYNONYM synonym FOR object ;
```
Removing Synonyms
```
DROP SYNONYM synonym ;
```
Data Dictionary

ALL_VIEWS	USER_VIEWS
ALL_SEQUENCES	USER_SEQUENCES
ALL_INDEXES	USER_INDEXES
ALL_IND_COLUMNS	USER_IND_COLUMNS

System Privileges(DBA)	**User System Privileges**
CREATE USER	CREATE SESION
DROP USER	CREATE TABLE
DROP ANY TABLE	CREATE SEQUENCE
BACKUP ANY TABLE	CREATE VIEW
	CREATE PROCEDURE

Creating Users
```
CREATE USER user
   IDENTIFIED BY password ;
```
Creating Roles
```
CREATE ROLE role ;
```
Granting System Privileges
```
GRANT privelges[,...] TO user[,...] ;
GRANT privelges TO role ;
GRANT role TO user[,...] ;
```
Changing Password
```
ALTER USER user IDENTIFIED BY password ;
```
Dropping Users
```
DROP USER user [CASCADE] ;
```

Dropping Roles
```
DROP ROLE role ;
```
Object Privileges

Object	Table	View	Sequence	Procedure
ALTER	X		X	
DELETE	X	X		
EXECUTE				X
INDEX	X			
INSERT	X	X		
REFERENCES	X			
SELECT	X	X	X	
UPDATE	X	X		

Object Privileges
```
GRAND object_priv [(column)]
   ON object
   TO  {user|role|PUBLIC}
   [WITH GRANT OPTION] ;
```
Revoking Object Privileges
```
REVOKE {privilege [,privilege...] | ALL}
   ON  object
   FROM {user[,user...]|role|PUBLIC}
   [CASCADE CONSTRAINTS] ;
```
Data Dictionary
```
ROLE_SYS_PRIVS
ROLE_TAB_PRIVS        USER_ROLE_PRIVS
USER_TAB_PRIVS_MADE   USER_TAB_PRIVS_RECD
USER_COL_PRIVS_MADE   USER_COL_PRIVS_RECD
```
PL/SQL Block Structure
```
DECLARE --Optional
 --Variables, Cursors, User-defined exceptions
BEGIN --Mandatory
 --SQL statements
 --PL/SQL statements
EXCEPTION --Optional
 --Actions to perform when errors occur
END ; --Mandatory
```
PL/SQL Block Type

Anonymous	**Procedure**	**Function**
[DECLARE]	PROCEDURE name	FUNCTION name
	IS	RETURN datatype IS
	[DECLARE]	[DECLARE]
BEGIN	BEGIN	BEGIN
--statements	--statements	--statements
[EXCEPTION]	[EXCEPTION]	[EXCEPTION]
END ;	END ;	END ;

Declaring PL/SQL Variables
```
identifier [CONSTANT] datatype [NOT NULL]
  [:=|DEFAULT expr] ;
```
Assigning Values to Variables
```
identifier := expr ;
```
Base Scalar Datatypes
```
VARCHAR2(n)      NUMBER(p,s)     DATE      CHAR(n)
LONG             LONG RAW        BOOLEAN
BINARY_INTEGER PLS_INTEGER
```
The %TYPE Attribute
```
table_name.column_name%TYPE ;
variable_name%TYPE ;
```
Composite Datatypes
```
TABLE          RECORD         NESTED TABLE    VARRAY
```
LOB Datatypes
```
CLOB           BLOB           BFILE           NCLOB
```
Creating Bind Variables
```
VARIABLE variable_name dataype
```
Displaying Bind Variables
```
PRINT [variable_name]
```
Commenting Code
```
--prefix single-line comments with two dashes
/* Place muti-line comment between the symbols */
```
SELECT Statements in PL/SQL
```
SELECT {column_list|*}
INTO {variable_name[,variable_name]...
      |record_name}
FROM table
WHERE condition
```
Implicit Cursor Attributes for DML statements
```
SQL%ROWCOUNT
SQL%FOUND
SQL%NOTFOUND
SQL%ISOPEN
```
Constrol Structures

IF Statement
```
IF condition THEN
   statements ;
[ELSIF condition THEN
   statements ;]
[ELSE
   statements;]
END IF ;
```
Basic Loop
```
LOOP
  statements;
  ...
  EXIT [WHEN condition];
END LOOP
```
FOR Loop
```
FOR conter in [REVERSE]
  lower..upper LOOP
  statement1;
  statement2;
  ...
END LOOP;
```
WHILE Loop
```
WHILE condition LOOP
   statement1;
   statement2;
   ...
END LOOP ;
```
Creating a PL/SQL Record
```
TYPE record_name_type IS RECORD
     (field_declaration[,field_declaration]...) ;
record_name record_name_type ;
```
Where field_declaration is
```
field_name {field_type|variable%TYPE|
                 table.column%TYPE|table%ROWTYPE}
                 [[NOT NULL] {:=|DEFAULT} expr]
```
Referencing Fields in the Record
```
record_name.field_name
```
Declaring Records with the %ROWTYPE Attribute
```
DECLARE
       record_name     reference%ROWTYPE
```
Creating a PL/SQL Table
```
TYPE type_name IS TABLE OF
     {column_scalr_type|variable%TYPE|table.column%TYPE
     |variable%ROWTYPE} [NOT NULL]
     [INDEX BY BINARY_INTEGER];
identifier type_name ;
```
Referencing a PL/SQL table
```
pl_sql_table_name(primary_key_value)
```

Using PL/SQL Table Method

```
table_name.method_name[(parameters)]
```

PL/SQL Table Methods

```
EXITS(n)       COUNT   FIRST   LAST    PRIOR(n)
NEXT(n)        EXTEND(n,i)     TRIM    DELETE
```

PL/SQL Table of Records

```
TYPE table_name_type IS TABLE OF table_name%ROWTYPE
     INDEX BY BINARY_INTEGER ;
table_name table_name_type ;
```

Referencing a Table of Records

```
table_name(index).field
```

Declaring the Cursor in Declaration Section

```
CURSOR cursor_name IS select_statement ;
record_name cursor_name%ROWTYPE ;
```

Opening and Closing the Cursor

```
OPEN cursor_name ;
CLOSE cursor_name ;
```

Fetching Data from the Cursor

```
FETCH cursor_name
INTO [variable1(,variable2,...)
                |record_name] ;
```

Explicit Cusor Attributes

```
cursor_name%ISOPEN
cursor_name%NOTFOUND
cursor_name%FOUND
cursor_name%ROWCOUNT
```

Cursor FOR Loops

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  ...
END LOOP;
```

Cursor FOR Loops Using Subqueries

```
FOR record_name IN (subqueries) LOOP
  statement1
  ...
END LOOP ;
```

Cursors with Parameters

```
CURSOR cursor_name [(cursor_parameter_name datatype
[,...])]
IS select_statement
[FOR UPDATE [OF column_reference][NOWAIT]];
```

Parameter Name

```
cursor_parameter_name [IN] datatype [{:=|DEFAULT}expr]
```

Openning with Parameters

```
OPEN cursor_name(cursor_parameter_name[,...]);
```

Cursor FOR Loops with parameters

```
FOR record_name IN cursor_name(cursor_parameter_name
[,...]) LOOP
  statement1;
  statement2;
  ...
END LOOP;
```

WHERE CURRENT OF clause

```
UPDATE|DELETE ... WHERE CURRENT OF cursor_name ;
```

Predefined Exceptions

```
NO_DATA_FOUND
TOO_MANY_ROWS
INVALID_CURSOR
ZERO_DIVIDE
DUP_VAL_ON_INDEX
```

Trapping Exceptions

```
EXCEPTION
   WHEN exception1 [OR exception2 ...] THEN
      statement1 ;
      statement2 ;
      ...
  [WHEN exception3 [OR exception4 ...] THEN
      statement1 ;
      statement2 ;
      ...]
  [WHEN OTHERS THEN
      statement1 ;
      statement2 ;
      ...]
```

Declaring Non-Predefined Oracle Sever Exception

```
DECLARE
   exception EXCEPTION ;
   PRAGMA EXCEPTION_INIT(exception, error_number) ;
```

Referencing the declared Non-predefined execption

```
BEGIN
   ...
EXCEPTION
   WHEN exception THEN
      statement1 ;
      ...
END ;
```

Trapping User-Defined Exceptions

```
DECLARE
   exception EXCEPTION ;
BEGIN
   ...
   IF SQL%NOTFOUND THEN
      RAISE exception ;
   END IF ;
   ...
EXCEPTION
   WHEN exception THEN
      statement1 ;
      ...
END ;
```

Functions for Trapping Exceptions

SQLCODE	return error code
SQLERRM	return error message

RAISE_APPLICATION_ERROR procedure(Executable/Exception Section)

```
RAISE_APPLICATION_ERROR ( error_number,
                          message [, {TRUE|FALSE}]) ;
```

error_number	between -20000 to -20999
message	string up to 2,048 bytes long
TRUE	placed on the stack of previous errors.
FALSE	replaces all previous errors

Single-Row Functions

Character Functions

```
LOWER(column|expression)
UPPER(column|expression)
```

```
INITCAP(column|expression)
INSTR(column|expression,m)
CONCAT(column1|expression1,column2|expression2}
SUBSTR(column|expression,m, [n])
LENGTH(column|expression)
LPAD(column|expression,n,'string')
```

Number Functions

```
MOD(m,n)
ROUND(column|expression,n)
TRUNC(column|expression,n)
```

Date Functions

```
MONTHS_BETWEEN(date1,date2)
ADD_MONTHS(date,n)
NEXT_DAY(date,'char')
LAST_DAY(date)
ROUND(date[,'fmt'])
TRUNC(date[,'fmt'])
```

Conversion Functions

```
TO_CHAR(number|date[,'fmt'])
TO_NUMBER(char[,'fmt'])
TO_DATE(char[,'fmt'])
NVL(expr1,expr2)
DECODE(col/expr,search1,result1
                   [,search2,result2,...,]
                   [,default])
```

Operators

Comparison	= > >= < <= <>
	BETWEEN..AND, IN, LIKE, IS NULL
Logical	AND OR NOT

Order of Operations

Operator	Operation
**,NOT	Exponentiation, logical negation
+,-	Identity, negation
*,/	Muliplication, division
+,-,||	Addition, subtraction, concatenation
=,!=,<,>,<=	Comparison
>=,IS NULL,LIKE	
BETEEN,IN	
AND	Conjunction
OR	Inclusion

Chapter 9 – PL/SQL

Procedural Language Extension of SQL

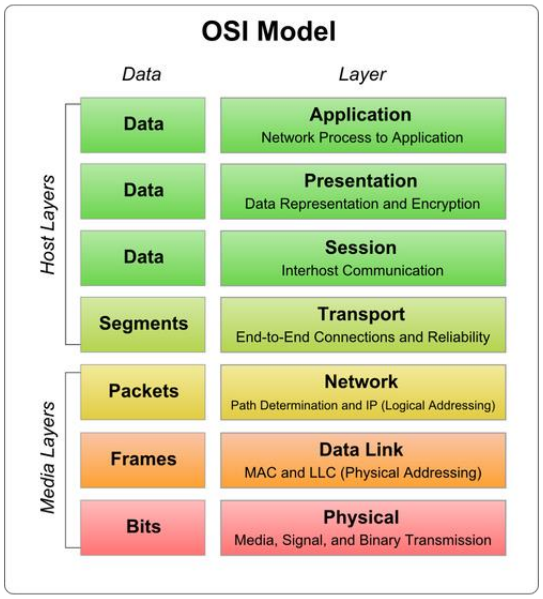
Chapter 10 – Computer Network

The OSI Model

ISO OSI (Open Systems Interconnection) Reference Model since it describes or relates to connecting systems that are open for communication with other systems. The OSI model depicts how data communications should take place. It splits the functions or processes into seven groups that are described as layers.

Each of the layers of the OSI model has a numerical level or layer and plain text descriptor. The Seven Layers of the OSI Model are 1 – Physical, 2 – Data Link, 3 – Network, 4 – Transport, 5 – Session, 6 – Presentation, 7 – Application.

A common mnemonic used to remember the OSI model layers starting with the seventh layer (Application) are: "**A**ll **P**eople **S**eem **t**o **N**eed **D**ata **P**rocessing." The lower two layers of the model are normally implemented through software and hardware solutions, while the upper five layers are typically implemented through the use of software only.


OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Physical Addressing)
Bits
Physical
Media, Signal, and Binary Transmission

Layer Seven – Application

The Application Layer's function is to provide services to the end-user such as email, file transfers, terminal access, and network management. This is the primary layer of interaction for the end-user in the OSI Model.

Layer Six – Presentation

The Presentation Layer's primary responsibility is to define the syntax that network hosts use to communicate. Compression and encryption fall in the functions of this layer. It is sometimes referred to as the "syntax" layer and is responsible for transforming information or data into format(s) the application layer can use.

Layer Five – Session

The Session Layer establishes process to process communications between two or more networked hosts. Under OSI, this layer is responsible for gracefully closing sessions (a property of TCP) and for session check pointing and recovery (not used in IP). It is used in applications that make use of remote procedure calls.

Layer Four – Transport

The Transport Layer is responsible for the delivery of messages between two or more networked hosts. It handles fragmentation and reassembly of messages and controls the reliability of a given link.

Layer Three – Network

The Network Layer is primarily responsible for establishing the paths used for data transfer on the network. Network routers operated at this layer which can commonly be divided into three sub-layers: Sub network access, Sub network-dependent convergence, and Sub network-independent convergence.

Layer Two – Data Link

The Data Link Layer is primarily responsible for communications between adjacent network nodes. Network switches and hubs operate at this layer which may also correct errors generated in the Physical Layer.

Layer One – Physical

The Physical Layer handles the bit level transmission between two or more network nodes. Components in this layer include connectors, cable types, pin-outs, and voltages which are defined by the applicable standards organization.

Real World Protocols Map to the OSI Model

The following are commonly used or implemented protocols mapped to the appropriate layer of the OSI Model (as best as they can be mapped).

Layer	Name	Common Protocols
7	Application	SSH, FTP, telnet
6	Presentation	HTTP, SNMP, SMTP
5	Session	RPC, Named Pipes, NETBIOS
4	Transport	TCP, UDP
3	Network	IP
2	Data Link	Ethernet
1	Physical	Cat-5, Co-axial or Fiber Optic Cable

TCP/IP Model Functions

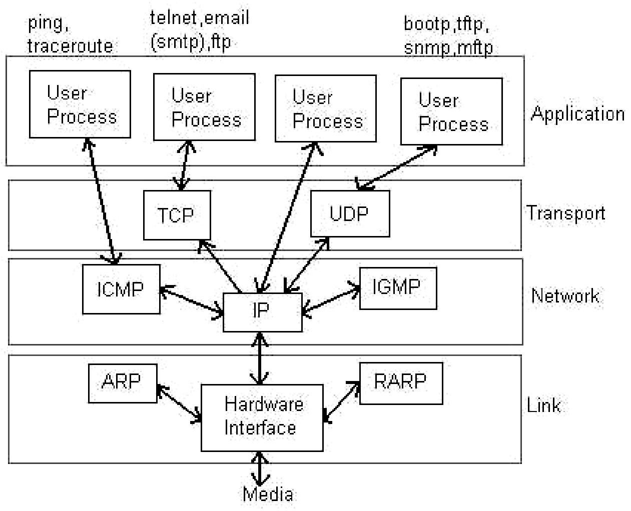
The TCP/IP Model has four functions. Starting from the lowest level, these include the Physical Layer, the Link Layer, the Internet, and the transport layers.

Physical Layer – The Physical Layer consists of purely hardware and includes the network interface card, connection cable, satellite, etc.

Link Layer – Also referred to as the "Network Access Layer." It is the networking scope of the local network connection that a host is attached. The lowest layer of IP, it is used to move data packets between the Internet Layer interfaces of two hosts on the same link. Controlling the process can be accomplished in either the software driver for the network card or via firmware in the chipset. The specifications for translating network addressing methods are included in the TCP/IP model, but lower level aspects are assumed to exist and not explicitly defined. A hierarchical encapsulation sequence is not dictated either.

Internet Layer – Handles the problem of sending data packets to or across one or more networks to a destination address in the routing process.

Transport Layer – The Transport Layer is responsible for end-end message transfer capabilities that are independent of the network. The specific tasks in this layer include error, flow, and congestion control, port numbers, and segmentation. Message transmission at this layer can either be connection-based as defined in TCP, or connectionless as implemented in the User Datagram Protocol (UDP).



The Internet Protocol performs two functions:

1 – Host identification and addressing. This function uses a hierarchical addressing system referred to as the IP address.

2 – Packet routing. This is the task of moving data packets from the source to destination host by sending the information to the next router or network node that is closer to the final destination. Information can be transported that relates to a number of upper layer protocols which are identified by a unique protocol number. Some examples are IGMP (Internet Group Management Protocol) and ICMP (Internet Control Message Protocol) that perform internetworking functions which help show the differences in the TCP/IP and OSI models.

TCP/IP Model Map to Real World Networking

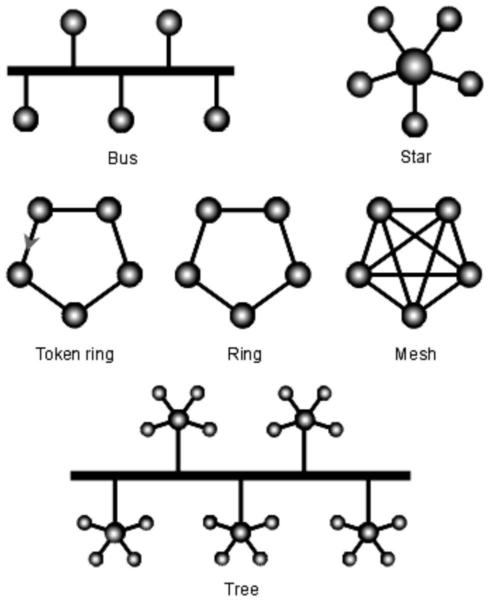
The TCP/IP model has become the defacto standard for real world implementation of networking. Some of the real world protocol mappings to the TCP/IP Model layers are:

TCP/IP Model	
Application Layer	FTP, HTTP, POP3, IMAP, telnet, SMTP, DNS, TFTP
Transport Layer	TCP, UDP, RTP
Internet Layer	IP, ICMP, ARP, RARP
Network Interface Layer	Ethernet, Token Ring, FDDI, X.25, Frame Relay, RS-232, v.36

Network Topology

In communication networks, a topology is a usually schematic description of the arrangement of a network, including its nodes and connecting lines. There are two ways of defining network geometry: the physical topology and the logical (or signal) topology.

The physical topology of a network is the actual geometric layout of workstations. There are several common physical topologies, as described below and as shown in the illustration.



In the bus network topology, every workstation is connected to a main cable called the bus. Therefore, in effect, each workstation is directly connected to every other workstation in the network.

In the star network topology, there is a central computer or server to which all the workstations are directly connected. Every workstation is indirectly connected to every other through the central computer.

In the ring network topology, the workstations are connected in a closed loop configuration. Adjacent pairs of workstations are directly connected. Other pairs of workstations are indirectly connected, the data passing through one or more intermediate nodes.

If a Token Ring protocol is used in a star or ring topology, the signal travels in only one direction, carried by a so-called token from node to node.

The mesh network topology employs either of two schemes, called full mesh and partial mesh. In the full mesh topology, each workstation is connected directly to each of the others. In the partial mesh topology, some workstations are connected to all the others, and some are connected only to those other nodes with which they exchange the most data.

The tree network topology uses two or more star networks connected together. The central computers of the star networks are connected to a main bus. Thus, a tree network is a bus network of star networks.

Logical (or signal) topology refers to the nature of the paths the signals follow from node to node. In many instances, the logical topology is the same as the physical topology. But this is not always the case. For example, some networks are physically laid out in a star configuration, but they operate logically as bus or ring networks.

Chapter 11 – Clustering and Load Balancing

Cluster

A cluster is defined as a group of application servers that transparently run a J2EE application as if it were a single entity. There are two methods of clustering: vertical scaling and horizontal scaling. Vertical scaling is achieved by increasing the number of servers running on a single machine, whereas horizontal scaling is done by increasing the number of machines in the cluster. Horizontal scaling is more reliable than vertical scaling, since there are multiple machines involved in the cluster environment, as compared to only one machine. With vertical scaling, the machine's processing power, CPU usage, and JVM heap memory configurations are the main factors in deciding how many server instances should be run on one machine (also known as the server-to-CPU ratio).

Clustering for Scalability & Failover

Scalability

Clustering typically makes one instance of an application server into a master controller through which all requests are processed and distributed to a number of instances using industry standard algorithms like round robin, weighted round robin, and least connections. Clustering, like load balancing, enables horizontal scalability that is the ability to add more instances of an application server nearly transparently to increase the capacity or response time performance of an application. Clustering features usually include the ability to ensure an instance is available through the use of ICMP ping checks and, in some cases, TCP or HTTP connection checks.

Server Affinity

Clustering uses server affinity to ensure that applications requiring the user interact with the same server during a session get to the right server. This is most often used in applications executing a process, for example order entry, in which the session is used between requests (pages) to store information that will be used to conclude a transaction, for example a shopping cart.

High Availability (Failover)

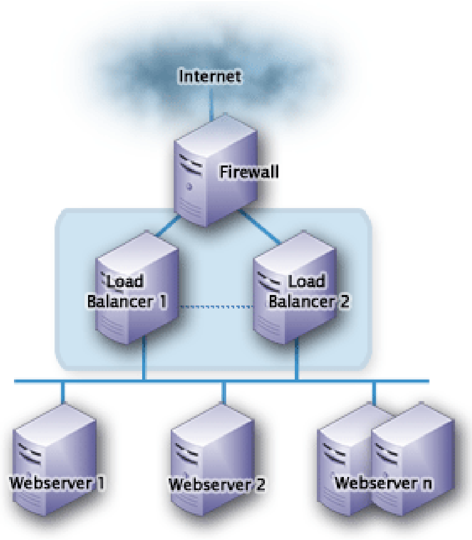
Clustering solutions claim to provide HA/Failover capabilities, when this failover is related to application process level failover, not high availability of the clustering controller itself. This is an important distinction as in the event the clustering controller instance fails, the entire system falls apart. While cluster-based load-balancing provides high availability for members of the cluster, the controller instance becomes a single point of failure in the data path.

Transparency

Many clustering solutions require a node-agent be deployed on each instance of an application server being clustered by the controller. This agent is often already deployed, so it's often not a burden in terms of deployment and management, but it is another process running on each server that is consuming resources such as memory and CPU and which adds another point of failure into the data path.

Load Balancing

Load balancing (also known as high availability switch over) is a mechanism where the server load is distributed to different nodes within the server cluster, based on a load balancing policy. Rather than execute an application on a single server, the system executes application code on a dynamically selected server. When a client requests a service, one (or more) of the cooperating servers is chosen to execute the request. Load balancers act as single points of entry into the cluster and as traffic directors to individual web or application servers.



Two popular methods of load balancing in a cluster are DNS round robin and hardware load balancing. DNS round robin provides a single logical name, returning any IP address of the nodes in the cluster. This option is inexpensive, simple, and easy to set up, but it doesn't provide any server affinity or high availability. In contrast, hardware load balancing solves the limitations of DNS round robin through virtual IP addressing. Here, the load balancer shows a single IP address for the cluster, which maps the addresses of each machine in the cluster. The load balancer receives each request and rewrites headers to point to other machines in the cluster. If we remove any machine in the cluster, the changes take effect immediately. The advantages of hardware load balancing are server affinity and high availability; the disadvantages are that it's very expensive and complex to set up.

Load Balancing Algorithms

There are many different algorithms to define the load distribution policy, ranging from a simple round robin algorithm to more sophisticated algorithms used to perform the load balancing. Some of the commonly used algorithms are:

- Round-robin
- Random
- Weight-based
- Minimum load
- Last access time
- Programmatic parameter-based (where the load balancer can choose a server based upon method input arguments)

Chapter 12 – Continuous Integration Process

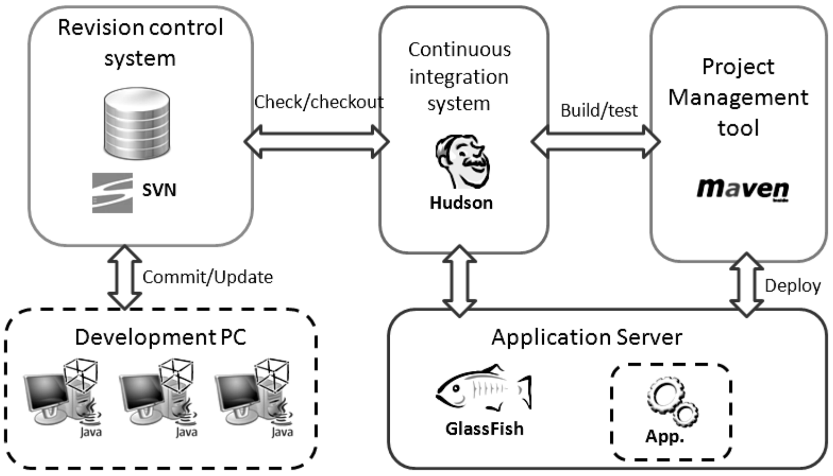
Continuous Integration

CI implements continuous processes of applying quality control. In other words, small pieces of effort applied frequently. CI aims to improve the quality of software, and reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development.

Hudson-Jenkins

Extensible continuous integration server features are:

1. Check-out the source code from the repository
2. Build and test the project. It covers unit testing, code quality reporting, code coverage reporting and build status notification by using tools like PMD, CheckStyle, VeraCode, Cruise Control, Damage Control, Cobertura, SONAR, and so on.)
3. Publish the result.
4. Communicate the results to team members.
5. Extendable with plug-ins (Clover)
6. Monitor the executions of externally run cron or procmail jobs.

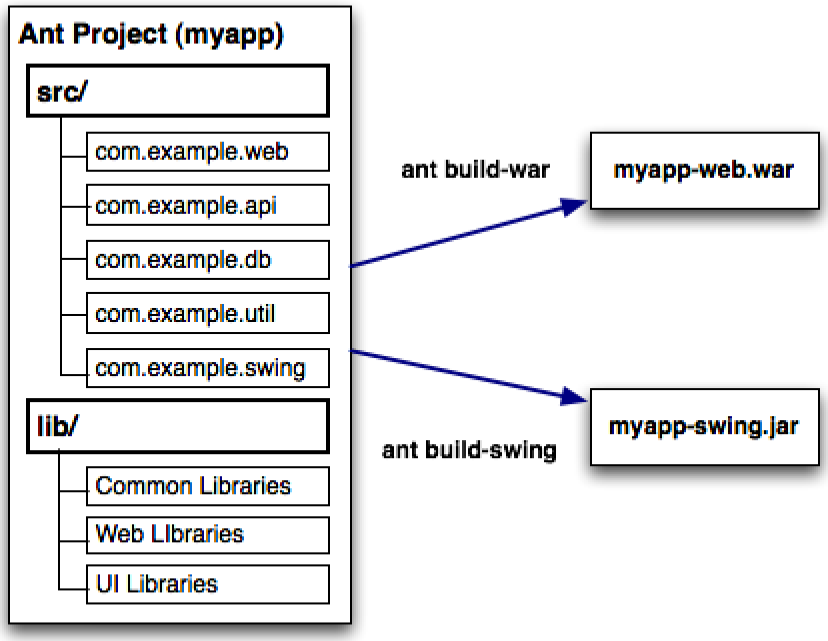


Chapter 13 - Project Management Tools

Ant

Apache Ant ("**A**nother **N**eat **T**ool") is the build tool used by most Java development projects. Ant made it trivial to integrate JUnit tests with the build process; Ant has made it easy for willing developers to adopt test-driven development, and even Extreme Programming. Ant build files are written in XML.

ANT Project Structure



Sample build.xml file

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
  <target name="clean" description="remove intermediate files">
    <delete dir="classes"/>
  </target>
  <target name="clobber" depends="clean" description="remove all artifact files">
    <delete file="hello.jar"/>
  </target>
  <target name="compile" description="compile the Java source code to class files">
    <mkdir dir="classes"/>
    <javac srcdir="." destdir="classes"/>
  </target>
  <target name="jar" depends="compile" description="create a Jar file for the application">
    <jar destfile="hello.jar">
      <fileset dir="classes" includes="**/*.class"/>
      <manifest>
        <attribute name="Main-Class" value="HelloProgram"/>
      </manifest>
    </jar>
  </target>
</project>
```

Within each target are the actions that Ant must take to build that target; these are performed using built-in tasks. For example, to build the compile target Ant must first create a directory called classes (Ant will only do so if it does not already exist) and then invoke the Java compiler. Therefore, the tasks used are mkdir and javac. These perform a similar task to the command-line utilities of the same name.

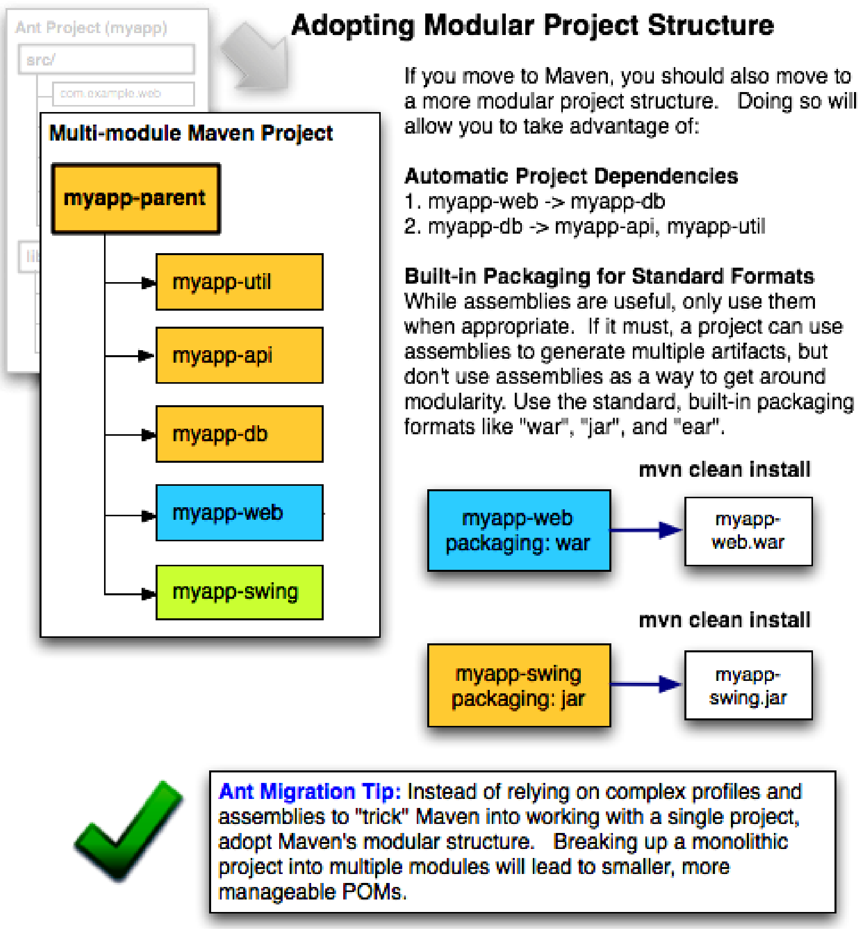
Another task used in this example is named jar:
 <jar destfile="hello.jar">

This ant task has the same name as the common java command-line utility, JAR, but is really a call to the ant program's built-in jar/zip file support. This detail is not relevant to most end users, who just get the JAR they wanted, with the files they asked for.
Many Ant tasks delegate their work to external programs, either native or Java. They use Ant's own <exec> and <java> tasks to set up the command lines, and handle all the details of mapping from information in the build file to the program's arguments -and interpreting the return value. Users can see which tasks do this (e.g. <cvs>, <signjar>, <chmod>, <rpm>), by trying to execute the task on a system without the underlying program on the path, or without a full Java Development Kit (JDK) installed.

Maven

Apache Maven is a project management, software comprehension and build automation tool primarily for Java. It can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. Maven serves a similar purpose to the Apache Ant tool, but it is based on different concepts and works in a profoundly different manner. Maven uses a construct known as a **Project Object Model (POM)** to describe the software project being built, its dependencies on other external modules and components, and the build order. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging.

Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository. This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated.
Maven is built using a plugin-based architecture that allows it to make use of any application controllable through standard input. Theoretically, this would allow anyone to write plugins to interface with build tools (compilers, unit test tools, etc.) for any other language. In reality, support and use for languages other than Java has been minimal.



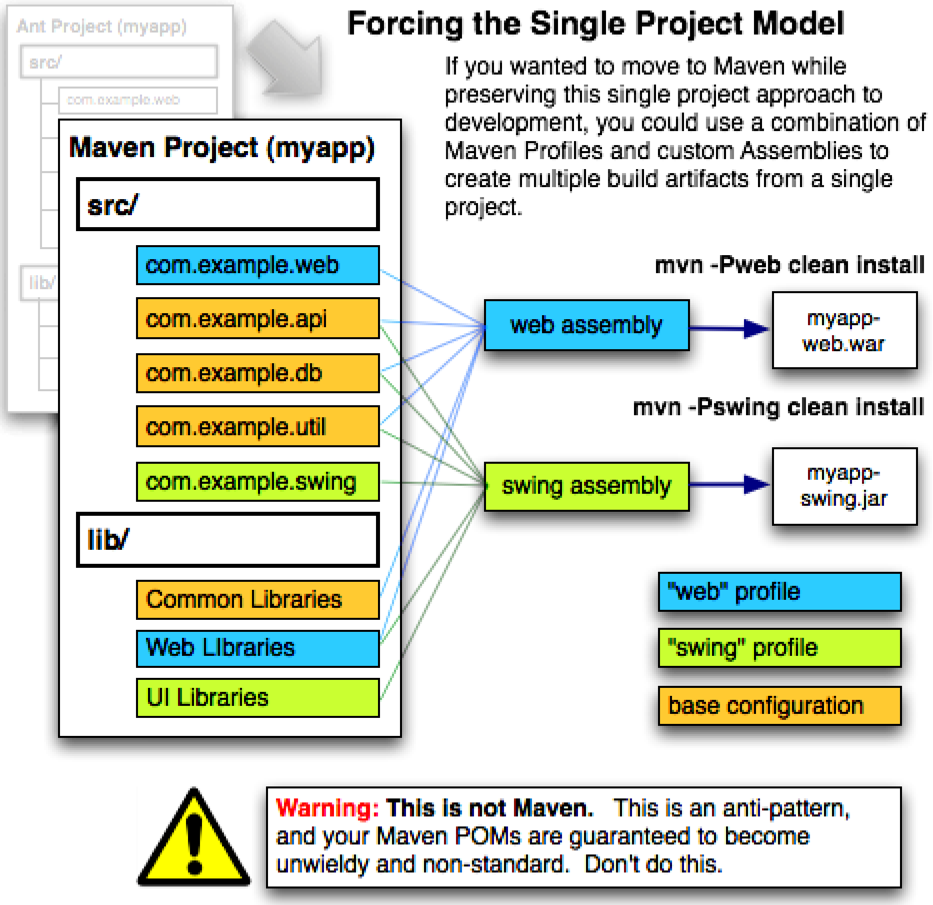
Adopting Modular Project Structure

If you move to Maven, you should also move to a more modular project structure. Doing so will allow you to take advantage of:

Automatic Project Dependencies
1. myapp-web -> myapp-db
2. myapp-db -> myapp-api, myapp-util

Built-in Packaging for Standard Formats
While assemblies are useful, only use them when appropriate. If it must, a project can use assemblies to generate multiple artifacts, but don't use assemblies as a way to get around modularity. Use the standard, built-in packaging formats like "war", "jar", and "ear".

Ant Migration Tip: Instead of relying on complex profiles and assemblies to "trick" Maven into working with a single project, adopt Maven's modular structure. Breaking up a monolithic project into multiple modules will lead to smaller, more manageable POMs.



Forcing the Single Project Model

If you wanted to move to Maven while preserving this single project approach to development, you could use a combination of Maven Profiles and custom Assemblies to create multiple build artifacts from a single project.

Warning: This is not Maven. This is an anti-pattern, and your Maven POMs are guaranteed to become unwieldy and non-standard. Don't do this.

Sample pom.xml file

```xml
<project>
 <!-- model version is always 4.0.0 for Maven 2.x POMs -->
 <modelVersion>4.0.0</modelVersion>
  <!-- project coordinates, i.e. a group of values which
     uniquely identify this project -->
  <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <version>1.0</version>
  <!-- library dependencies -->
  <dependencies>
   <dependency>
     <!-- coordinates of the required library -->
     <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
     <!-- this dependency is only used for running and compiling tests -->
     <scope>test</scope>
    </dependency>
 </dependencies>
</project>
```

Project Object Model

A Project Object Model (POM) provides the entire configuration for a single project. General configuration covers the project's name, its owner and its dependencies on other projects. One can also configure individual phases of the build process, which are implemented as plugins. For example, one can configure the compiler-plugin to use Java version 1.5 for compilation, or specify packaging the project even if some unit test fails.

Larger projects should be divided into several modules, or sub-projects, each with its own POM. One can then write a root POM through which one can compile all the modules with a single command. POMs can also inherit configuration from other POMs. All POMs inherit from the Super POM] by default. The Super POM provides default configuration, such as default source directories, default plugins, and so on.

Plugins

Most of Maven's functionality is in plugins. A plugin provides a set of goals that can be executed using the following syntax:

mvn [plugin-name]:[goal-name]

For example, a Java project can be compiled with the compiler-plugin's compile-goal by running mvn compiler:compile.

There is Maven plugins for building, testing, source control management, running a web server, generating Eclipse project files, and much more. Plugins are introduced and configured in a <plugins>-section of a pom.xml file. Some basic plugins are included in every project by default, and they have sensible default settings.

However, it would be cumbersome if one would have to run several goals manually in order to build, test and package a project:

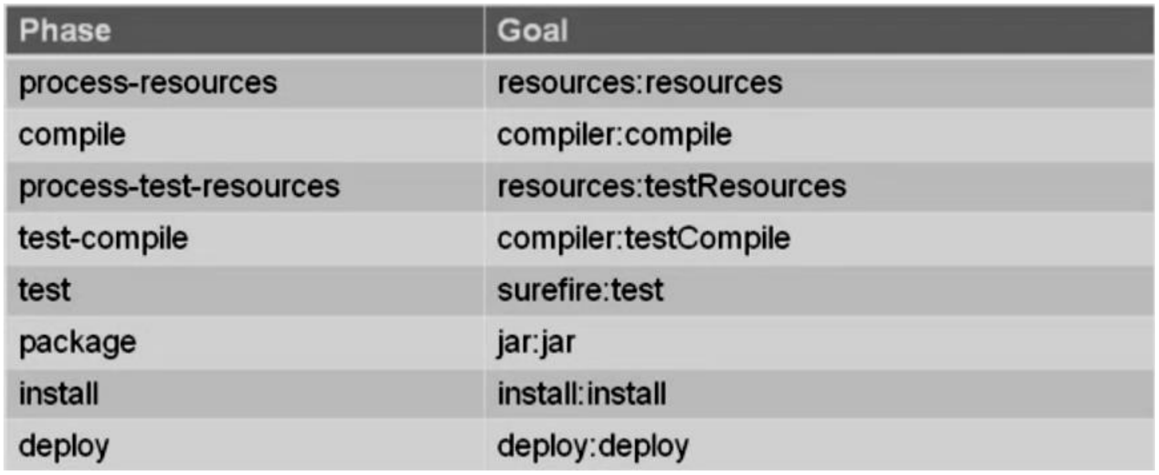
mvn compiler:compile
mvn surefire:test
mvn jar:jar

Maven's lifecycle-concept handles this issue.

Build lifecycles

Build lifecycle is a list of named phases that can be used to give order to goal execution. One of Maven's standard lifecycles is the default lifecycle, which includes the following phases, in this order:

1. process-resources
2. compile
3. process-test-resources
4. test-compile
5. test
6. package
7. install
8. deploy



Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

Goals provided by plugins can be associated with different phases of the lifecycle. For example, by default, the goal "compiler:compile" is associated with the compile-phase, while the goal "surefire:test" is associated with the test-phase. When the command ‘mvn test’ is executed, Maven will run all the goals associated with each of the phases up to the test-phase. So it will run the "resources:resources"-goal associated with the process-resources-phase, then "compiler:compile", and so on until it finally runs the "surefire:test"-goal.

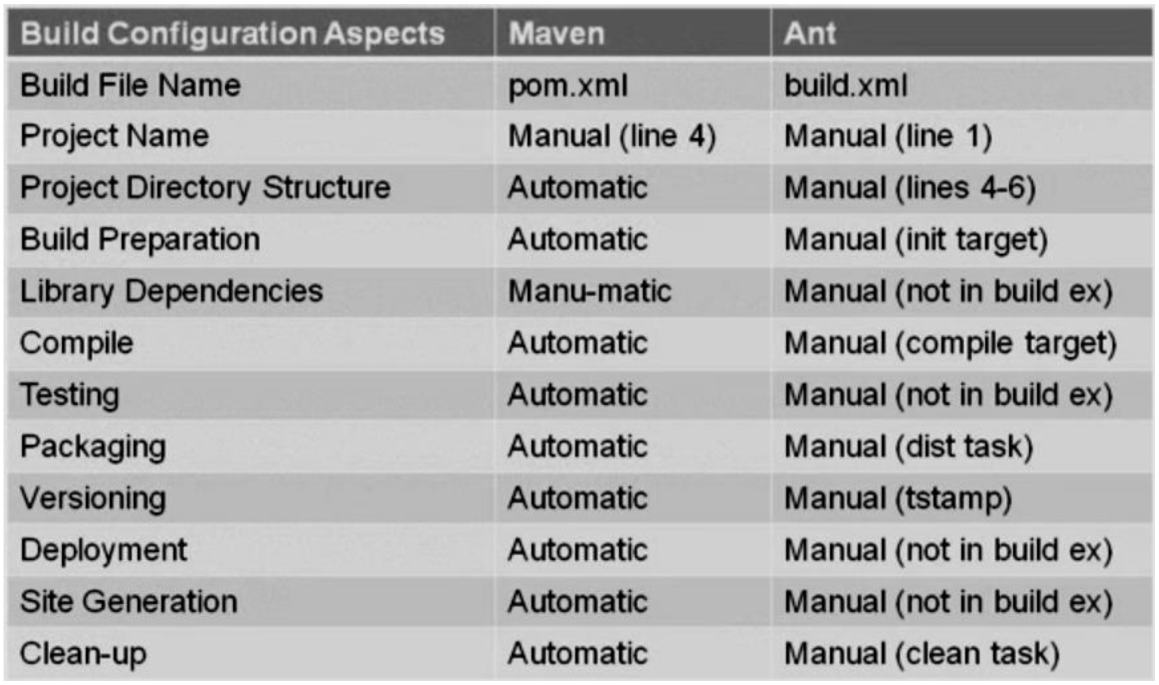
Maven also has standard lifecycles for cleaning the project and for generating a project site. If cleaning were part of the default lifecycle, the project would be cleaned every time it was built. This is clearly undesirable, so cleaning has been given its own lifecycle.

Thanks to standard lifecycles, one should be able to build, test and install every Maven-project using the mvn install-command.

Dependencies

The example-section hinted at Maven's dependency-handling mechanism. A project that needs the Hibernate-library simply has to declare Hibernate's project coordinates in its POM. Maven will automatically download the dependency and the dependencies that Hibernate itself needs (called transitive dependencies) and store them in the user's local repository. Maven 2 Central Repository is used by default to search for libraries, but one can configure the repositories used (e.g. company-private repositories) in POM. There are search engines such as Maven Central, which can be used to find out coordinates for different open-source libraries and frameworks.

Maven compared with ANT



Build Configuration Aspects	Maven	Ant
Build File Name	pom.xml	build.xml
Project Name	Manual (line 4)	Manual (line 1)
Project Directory Structure	Automatic	Manual (lines 4-6)
Build Preparation	Automatic	Manual (init target)
Library Dependencies	Manu-matic	Manual (not in build ex)
Compile	Automatic	Manual (compile target)
Testing	Automatic	Manual (not in build ex)
Packaging	Automatic	Manual (dist task)
Versioning	Automatic	Manual (tstamp)
Deployment	Automatic	Manual (not in build ex)
Site Generation	Automatic	Manual (not in build ex)
Clean-up	Automatic	Manual (clean task)

The fundamental difference between Maven and Ant is that Maven's design regards all projects as having a certain structure and a set of supported task work-flows (e.g. getting resources from source control, compiling the project, unit testing, etc.). While most software projects in effect support these operations and actually do have a well-defined structure, Maven requires that this structure and the operation implementation details be defined in the POM file. Thus, Maven relies on a convention on how to define projects and on the list of work-flows that are generally supported in all projects.

This design constraint is more like how an IDE handles projects and it provides many benefits, such as a succinct project definition and the possibility of automatic integration of a Maven project with other development tools such as IDEs, build servers, etc.

The downside is that it requires a user to first understand what a project is from the Maven point of view and how Maven works with projects, because what happens when one executes a phase in Maven is not immediately obvious just from examining the Maven project file. This required structure is also often a barrier in migrating a mature project to Maven because it is usually hard to adapt from other approaches.

In Ant, projects do not really exist from the tool's technical perspective. Ant works with XML build scripts defined in one or more files. It processes targets from these files and each target executes tasks. Each task performs a technical operation such as running a compiler or copying files around. Tasks are executed primarily in the order given by their defined dependency on other tasks. Thus, Ant is a tool that chains together tasks and executes them based on inter-dependencies and other Boolean conditions.

The benefits provided by Ant are also numerous. It has an XML language optimized for clearer definition of what each task does and on what it depends. Also, all the information about what will be executed by an Ant target can be found in the Ant script.

A developer not familiar with Ant would normally be able to determine what a simple Ant script does just by examining the script. This is not usually true for Maven.

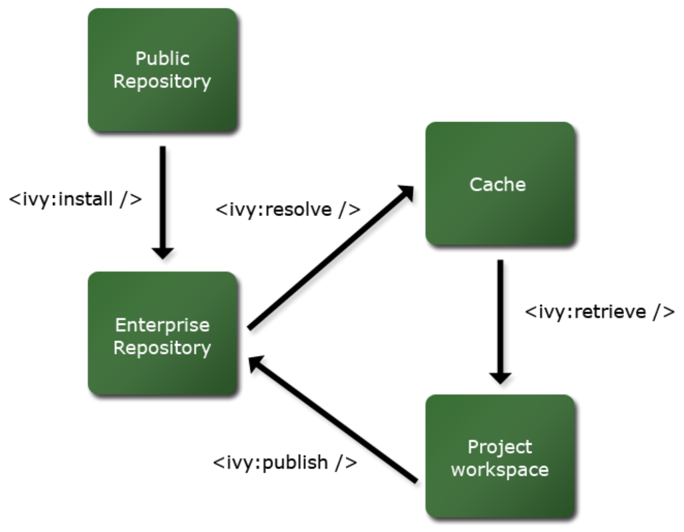
However, even an experienced developer that is new to a project using Ant cannot infer what the higher level structure of an Ant script is and what it does without examining the script in detail. Depending on the script's complexity, this can quickly become a daunting challenge. With Maven, a developer who previously worked with other Maven projects can quickly examine the structure of a never before seen Maven project and execute the standard Maven work-flows against it while already knowing what to expect as an outcome.

It is possible to use Ant scripts that are defined and behave in a uniform manner for all projects in a working group or an organization. However, when the number and complexity of projects rises, it is also very easy to stray from the initially desired uniformity. With Maven this is less of a problem because the tool always imposes a certain way of doing things.
Note that it is also possible to extend and configure Maven in a way that departs from the Maven way of doing things.

Ivy

Apache Ivy is a transitive relation dependency manager. It is a sub-project of the Apache Ant project, with which Ivy works to resolve project dependencies. An external XML file defines project dependencies and lists the resources necessary to build a project. Ivy then resolves and downloads resources from an artifact repository: either a private repository or one publicly available on the Internet.

To some degree, it competes with Apache Maven, which also manages dependencies. However, Maven is a complete build tool, whereas Ivy focuses purely on managing transitive dependencies.



Features:

- Managing project dependencies
- XML-driven declaration of project dependencies and jar repositories
- Automatic retrieval of transitive dependency definitions and resources
- Automatic integration to publicly-available artifact repositories
- Resolution of dependency closures
- Configurable project state definitions, which allow for multiple dependency-set definitions
- Publishing of artifacts into a local enterprise repository

Chapter 14 – Unit Test

JUnit

A unit test is a piece of code written by a developer that tests a specific functionality in the code. Unit tests can ensure that functionality is working and can be used to validate that this functionality still works after code changes.

JUnit is a framework developed in Java for unit test purposes.

- extends junit.framework.TestCase
- protected void setup()
- protected void tearDown()
- private void testSuite()

Assertion Methods

1. assertNull
2. assertNotNull
3. assertEquals
4. assertNotEquals
5. assertSame
6. assertNotSame
7. assertTrue
8. assertFalse
9. fail

You can also use an **inner class with overridden methods** to mock and test the specific functionality in the code.

JUnit 4.x Annotations

1. @Test
2. @Test(expected=ExceptionName.class)
3. @Test(timeout=100)
4. @Before
5. @After
6. @BeforeClass - setup()
7. @AfterClass - tearDown()
8. @Ignore

Easy Mock

Easy Mock provides Mock Objects for interfaces (and objects through the class extension) by generating them on the fly using Java's proxy mechanism. Due to Easy Mock's unique style of recording expectations, most refactoring will not affect the Mock Objects. So, **Easy Mock is a perfect fit for Test-Driven Development.**

Interface Mockups - High Level Steps

1. Creating the mock
2. Defining or Recording the behavior
3. Signal a change into "replay" mode
4. Test with your mock object
5. Verify (optional)

Mock Types

There are three types of mock objects

1) Strict Mock

- The method name is createStrictMock()
- Calling an undefined behavior results in Assertion Error
- The Order matters

2) Mock (Standard)

- The method name is createMock()
- Calling an undefined behavior results in Assertion Error
- The Order doesn't matter

3) Nice Mock

- The method name is createNiceMock()
- Calling an undefined behavior results in no Assertion Error
- The Order doesn't matter

Mock Modes

The two modes of a mock are **'Record'** and **'Replay'**.

The initial mode is 'Record. When in record mode, you tell the mock object what to expect and how to react (not relevant for void). Calling a method in record mode defines an expected behavior. Behavior is the method and the parameter values.

In 'Replay' mode, the mock objects act per the behaviors defined while in record mode. The optional 'Verify' confirms that the methods you defined were all called.

Behavior

The methods with return types need an expect(). The methods with return types need a return object defined.

- andReturn()
- andThrow()
- andStubReturn()
- andStubThrow()

The stub methods won't be considered in verify().

Every behavior definition is just ONE definition

- times(int num)
- times(int min, int max)
- anyTimes()
- atLeastOnce()

Flexible Argument Matching

It lets you define the looser requirements on the parameters. The options are:

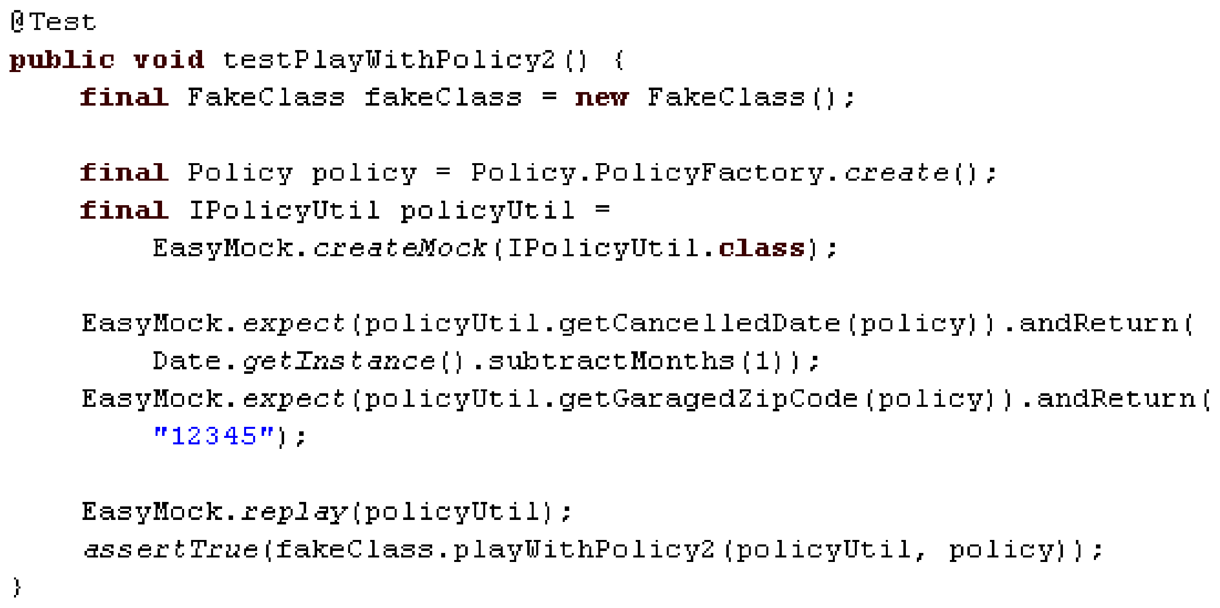
- eq(X value)
- isNull()
- notNull()
- isA(Class clazz)
- anyInteger()
- anyBoolean()
- anyObject()

Usage:
```
policyUtil.deletePolicyErrors(
    (Policy)EasyMock.notNull());
```

How to write a test case using Easy Mock

- Declare org.easymock.IMocksControl control
- In setup method, contol = createStrictControl()
- Create an instance of Testable class with overridden methods.
- control.createMock()
- expect
- andReturn
- control.checkOrder(Boolean)
- control.replay
- testableObject.invokeMethod()
- assertion
- control.verify
- control.reset
- fail

Sample Code:



```
@Test
public void testPlayWithPolicy2() {
    final FakeClass fakeClass = new FakeClass();

    final Policy policy = Policy.PolicyFactory.create();
    final IPolicyUtil policyUtil =
        EasyMock.createMock(IPolicyUtil.class);

    EasyMock.expect(policyUtil.getCancelledDate(policy)).andReturn(
        Date.getInstance().subtractMonths(1));
    EasyMock.expect(policyUtil.getGaragedZipCode(policy)).andReturn(
        "12345");

    EasyMock.replay(policyUtil);
    assertTrue(fakeClass.playWithPolicy2(policyUtil, policy));
}
```

Partial Class Mockups

- Think of it like extending a class on the fly
- Specify which methods you want to mock
- All other methods function as usual
- Can’t mock final classes
- Can’t mock private methods

Partial Class Mockups – High Level Steps

1. Creating the mock builder
2. Define the methods to be mocked
3. Creating the mock
4. Defining or Recording the behavior
5. Signal a change into “replay” mode
6. Test with your mock object
7. Verify (optional)

Sample Code:

```
// step 1: create mock builder
IMockBuilder<CommonServicesBean> csMockBuilder =
    EasyMock.createMockBuilder(CommonServicesBean.class);

// step 2: define methods to be mocked
csMockBuilder.addMockedMethod("getMailingStatesMinusNonUS");
csMockBuilder.addMockedMethod("getRegisteredStatesMinusCanada");

// step 3: create mock
CommonServicesBean csBean = csMockBuilder.createNiceMock();

// step 4: define behavior
EasyMock.expect(csBean.getMailingStatesMinusNonUS()).andStubReturn(
    new SelectItem[] {});
EasyMock.expect(csBean.getRegisteredStatesMinusCanada()).andStubReturn(
    new SelectItem[] {});

// step 5: replay
EasyMock.replay(csBean);

// step 6: test
fakeClass.playWithCommonServiceBean(csBean);

// step 7: verify
EasyMock.verify(csBean);
```

Chapter 15 - Software Design Patterns

Software Design Patterns with examples

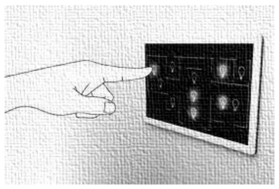
STRUCTURAL

1) Adapter
Convert the interface of a class into another interface clients expect. It lets the classes work together that couldn't otherwise because of incompatible interfaces.

Socket attaches to a Ratchet

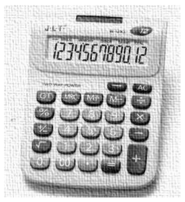
2) Bridge
Decouple an abstraction from its implementation so that the two can vary independently.

Switch controlling lights



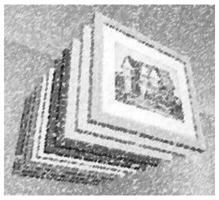
3) Composite
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Arithmetic Expression



4) Decorator
Attach additional responsibilities to objects dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Paintings with Frames



5) Facade
Provide a unified interface to a set of interfaces in a subsystem. It defines a high-level interface that makes the subsystem easier to use.

Customer Service Representative

6) Flyweight
Use sharing to support large number of fine grained objects effectively.

Public Switched Telephone Network

7) Proxy
Provide a surrogate or placeholder for another object to control access to it.

Check or Bank Draft

CREATIONAL

8) Abstract Factory
It provides an interface for creating families of related or dependent objects without specifying their concrete class.

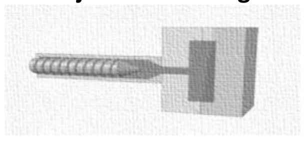
Stamping Equipment

9) Builder
Separate the construction of complex object from its representing so that the same construction process can create different representations.

Kids Meal

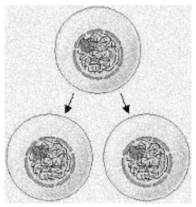
10) Factory Method
Define an interface for creating an object, but let subclasses decide which class to instantiate. Let the class defer Instantiation to subclasses.

Injection Molding



11) Prototype
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Division of a cell – Cloning



12) Singleton
Ensure a class only has one instance and provide a global point of access to it.

Office of the president of the United States

BEHAVIORAL

13) Chain of Responsibility
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

Coin Sorting Machine

14) Command
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Diner Check

MA'S DINER
BEST VITTLES IN TOWN
- ONLY VITTLES IN TOWN!

15) Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Musicians

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

Modern Television Sets

17) Mediator
Define an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly and its lets you to vary their interactions independently.

Airport Control Tower

18) Memento
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Furniture Assembling Manual

19) Observer
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Auction

20) State
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

ATM or Vending Machine

21) Strategy
Define a family of algorithms, encapsulate each one, and make them interchangeable. Let the algorithm vary independently from clients that use it.

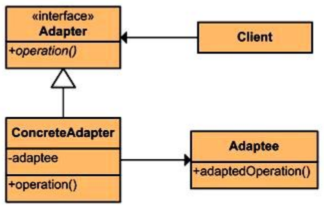
Modes of Transportation to a place

22) Template
Define the Skelton of an algorithm in an operation, deferring some steps to subclasses. Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

House Floor Plan

23) Visitor
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

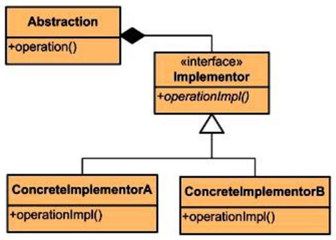
Operation of a Taxi Company



Adapter

Type: Structural

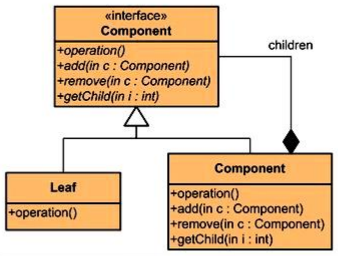
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Bridge

Type: Structural

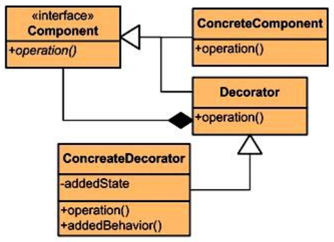
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Composite

Type: Structural

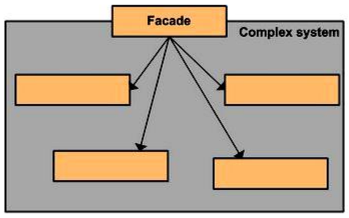
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Decorator

Type: Structural

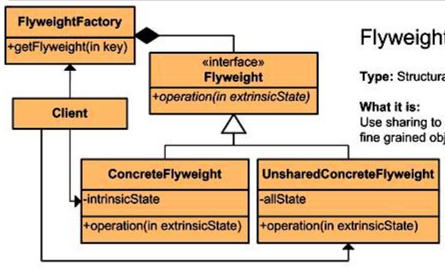
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.


FlyweightFactory
+getFlyweight(in key)
«interface»
Flyweight
+operation(in extrinsicState)
Client
ConcreteFlyweight
-intrinsicState
+operation(in extrinsicState)
UnsharedConcreteFlyweight
-allState
+operation(in extrinsicState)

Flyweight

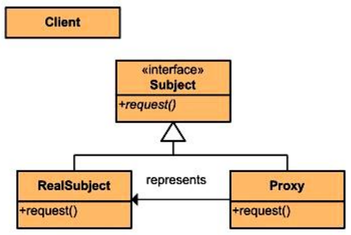
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Proxy

Type: Structural

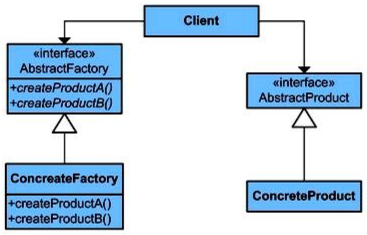
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Abstract Factory

Type: Creational

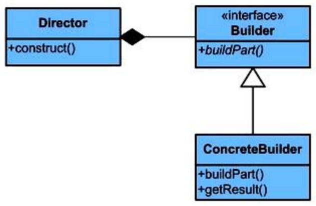
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Builder

Type: Creational

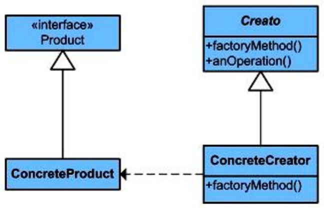
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Factory Method

Type: Creational

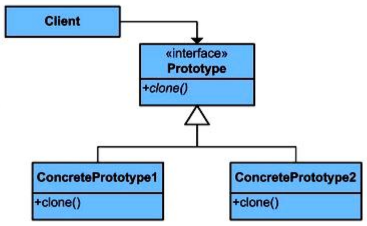
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Prototype

Type: Creational

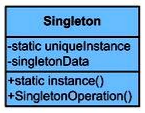
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Singleton

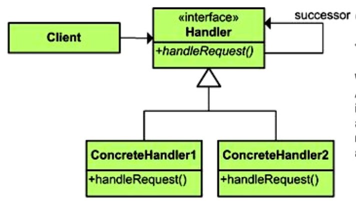
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

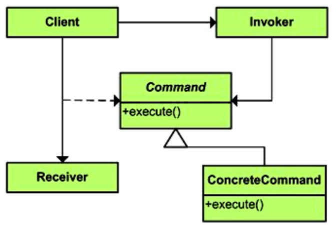
C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		



Chain of Responsibility

Type: Behavioral

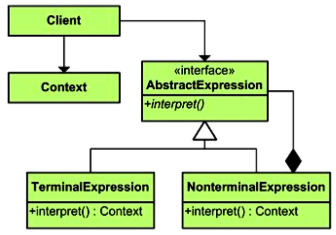
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Command

Type: Behavioral

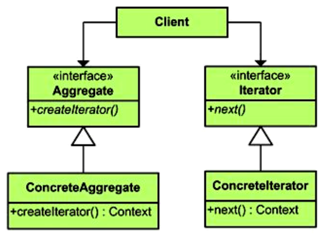
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Interpreter

Type: Behavioral

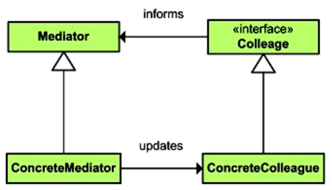
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Iterator

Type: Behavioral

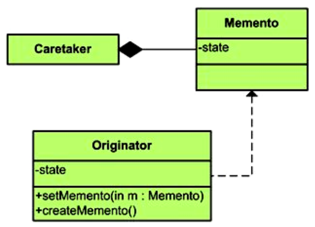
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Mediator

Type: Behavioral

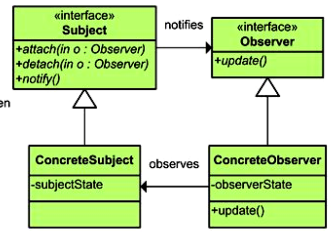
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interaction independently.



Memento

Type: Behavioral

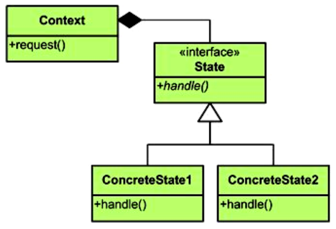
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Observer

Type: Behavioral

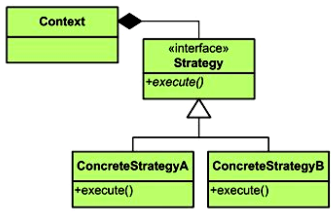
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



State

Type: Behavioral

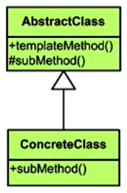
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Strategy

Type: Behavioral

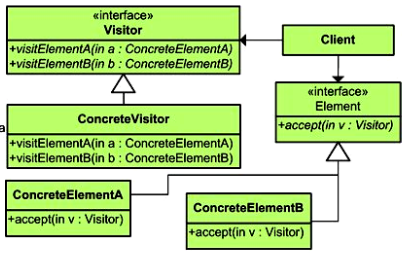
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.

Chapter 16 – J2EE Design Patterns with Frameworks

J2EE Design Patterns with Frameworks

Well, the design pattern is simply a description of a recurring solution to a problem, given a context. The context is the environment, surroundings, situation, or interrelated conditions within which the problem exists.

Design patterns have a number of advantages like

1. Once described, any level engineer can use the pattern.
2. They allow for reuse without having to reinvent in every a project.
3. They allow to better defining system structure.
4. They provide a design vocabulary.
5. They provide reusable artifacts.
6. Patterns can form frameworks that can then be used for implementations.

In this section we have tried to cover how to use and identify design patterns, in J2EE applications. The interest in design patterns has been around for a number of years in the software industry. However, interest among mainstream software developers is a fairly recent development actually it takes a highly experienced engineer to recognize a pattern, it requires collaboration, and it requires ongoing refinements.

There are a number of patterns that have been identified by the Sun Java Center for the presentation tier.

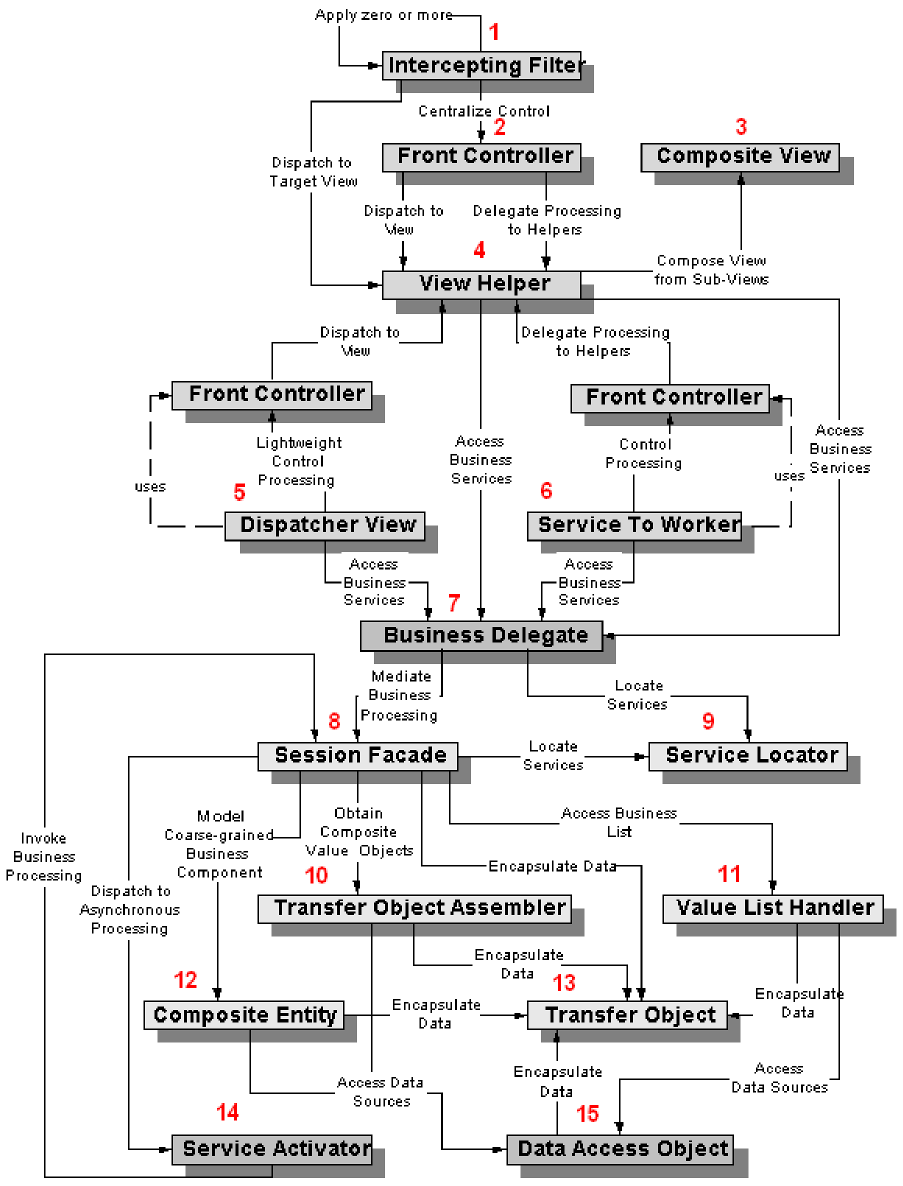
1. Intercepting Filter
It facilitates preprocessing and post-processing of a request.

2. Front Controller
It provides a centralized controller for managing the handling of requests.

3. Composite View
It creates an aggregate View from atomic subcomponents.

4. View Helper
It encapsulates logic that is not related to presentation formatting into Helper components.

5. Dispatcher View
It combines a Dispatcher component with the Front Controller and View Helper patterns, deferring many activities to View processing.



6. Service to Worker

It combines a Dispatcher component with the Front Controller and View Helper patterns. Service to Worker pattern usually works with Front Controller. The Front Controller is responsible to delegate the request to a worker module.

Dispatcher View pattern works the same way but the **opposite direction** of Service to Worker pattern.

7. Business Delegate

It reduces coupling between presentation-tier clients and business services. It hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

8. Session Facade
It encapsulates the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.

9. Service Locator
Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.

10. Transfer Object Assembler
It is used to build the required model or sub model. The Transfer Object Assembler uses Transfer Objects to retrieve data from various business objects and other objects that define the model or part of the model.

11. Value List Handler
The most critical concern in a distributed paradigm is the latency time. Value List Handler Pattern suggests an alternate approach of using EJB finder methods. The pattern is used to control the search, cache the results and provide the results to the client using a lightweight mechanism.

12. Composite Entity
It model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained entity beans. A Composite Entity bean represents a graph of objects.

13. Transfer Object
It encapsulates the business data. A single method call is used to send and retrieve the Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Transfer Object, populate it with its attribute values, and pass it by value to the client.

14. Service Activator
It enables asynchronous access to enterprise beans and other business services. It receives asynchronous client requests and messages. On receiving a message, the Service Activator locates and invokes the necessary business methods on the business service components to fulfill the request asynchronously. In EJB2.0, Message Driven beans can be used to implement Service Activator for message based enterprise applications. The Service Activator is a JMS Listener and delegation service that creates a message façade for the EJBs.

15. Data Access Object
Abstracts and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.

Delegation Pattern

In software engineering, the delegation pattern is a design pattern in object-oriented programming where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object. There is an Inversion of Responsibility in which a helper object, known as a delegate, is given the responsibility to execute a task for the delegator. The delegation pattern is one of the fundamental abstraction patterns that underlie other software patterns such as composition (also referred to as aggregation), mixins and aspects.

```
class RealPrinter { // the "delegate"
    void print() {
      System.out.print("something");
    }
}

class Printer { // the "delegator"
    RealPrinter p = new RealPrinter(); // create the delegate
    void print() {
      p.print(); // delegation
    }
}

public class Main {
    // to the outside world it looks like Printer actually prints.
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print();
    }
}
```

Delegation is the simple yet powerful concept of handing a task over to another part of the program. In object-oriented programming it is used to describe the situation where one object defers a task to another object, known as the delegate. This mechanism is sometimes referred to as aggregation, consultation or forwarding (when a wrapper object doesn't pass itself to the wrapped object).

Delegation is dependent upon dynamic binding, as it requires that a given method call can invoke different segments of code at runtime.

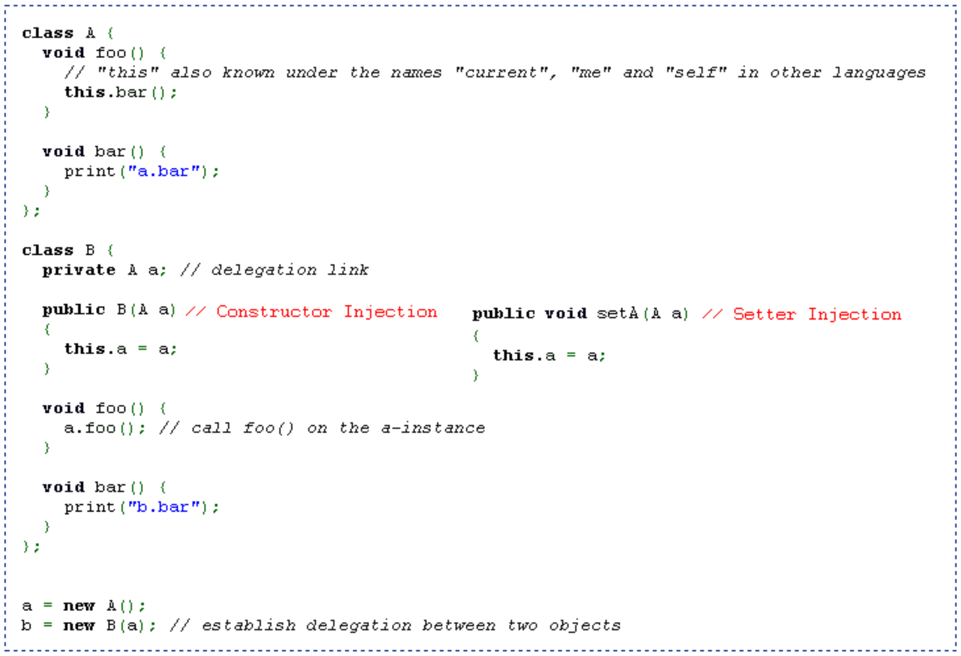
Spring's Inversion of Control (IOC), also known as Dependency Injection is achieved by applying Delegation pattern. Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

There are three types of dependency injection:

- **Constructor Injection** (e.g. Pico container, Spring etc): Dependencies are provided as constructor parameters.
- **Setter Injection** (e.g. Spring): Dependencies are assigned through JavaBeans properties (ex: setter methods).
- **Interface Injection** (e.g. Avalon): Injection is done through an interface.

Note: Spring supports only Constructor and Setter Injection

The below code snippet explains the constructor and setter dependency injection.



```
class A {
  void foo() {
    // "this" also known under the names "current", "me" and "self" in other languages
    this.bar();
  }

  void bar() {
    print("a.bar");
  }
};

class B {
  private A a; // delegation link

  public B(A a) // Constructor Injection    public void setA(A a) // Setter Injection
  {                                         {
    this.a = a;                               this.a = a;
  }                                         }

  void foo() {
    a.foo(); // call foo() on the a-instance
  }

  void bar() {
    print("b.bar");
  }
};


a = new A();
b = new B(a); // establish delegation between two objects
```

Other Patterns

Spring DAO module uses Data Access Object pattern and Spring MVC module is based on MVC2 architecture.

Design Patterns in Struts Framework

Struts is based on MVC (Model-View-Controller) Model2 or simply **MVC2** architecture. Struts controller uses the command design pattern and the action classes use the adapter design pattern. The process() method of the Request Processor uses the template method design pattern. Struts also implement the following J2EE design patterns.

- **Service to Worker**
- **Dispatcher View**
- **Composite View (Struts Tiles)**
- **Front Controller**
- **View Helper**
- **Synchronizer Token**

An Action Class in Struts performs a role of an **Adapter** between the contents of an incoming HTTP request and the corresponding business logic that should be executed to process this request. Action Servlet of Struts is part of Controller components which works as **Front Controller** to handle all the requests. The Interceptors in Struts 2 uses the **Intercepting Filter** pattern.

Design Patterns in Hibernate Framework

Composite Pattern and Composite Entity Pattern
Hibernate framework implements composite design pattern with annotations to build hierarchical tree structures of entities. It helps creating persistent trees of objects.

Proxy Pattern
The Lazy Loading concept in Hibernate framework is based on proxy collections.

Design Patterns in Apache Axis Framework

Chain of Responsibility Pattern
The chains between the request and response handlers in Apache Axis architecture uses the Chain of Responsibility pattern in which a request flows along a sequence of Handlers until it is processed. Although an Axis Chain may process a request in stages over a succession of Handlers, it has the same advantages as Chain of Responsibility: flexibility and the ease with which new function can be added.

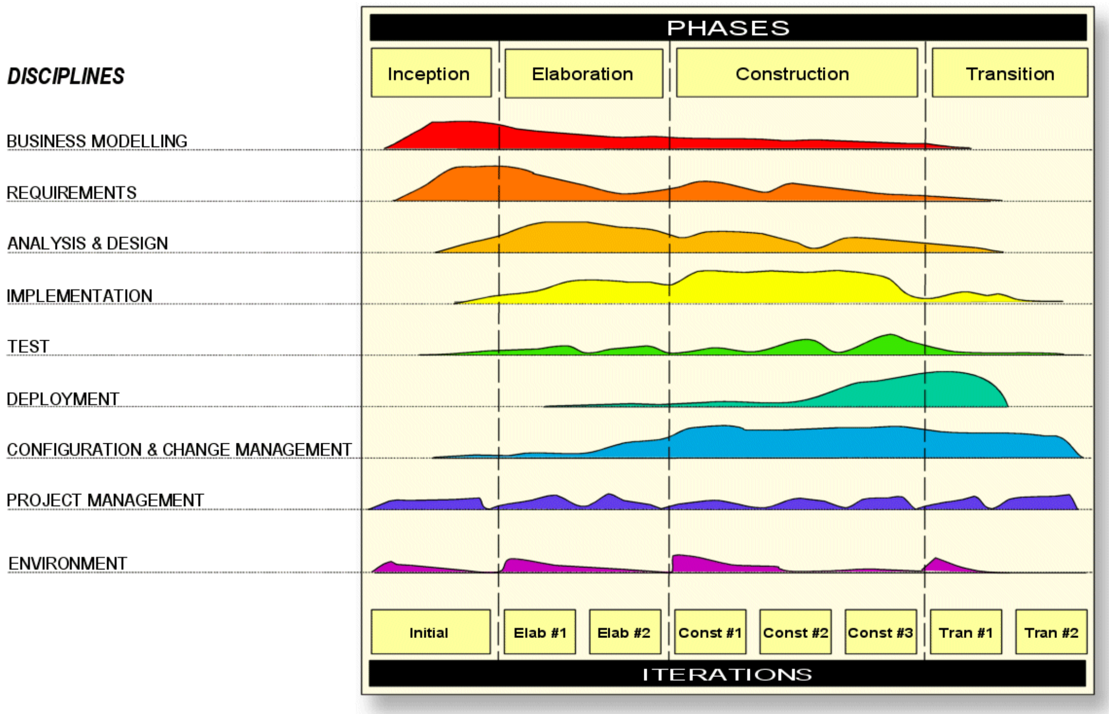
Chapter 17 – Rational Unified Process (RUP)

Rational Unified Process (RUP)

It is an iterative and incremental software development process framework created by IBM in 2003.

Four Phases:

- Inception
- Elaboration
- Construction
- Transition



Six best practices for modern software engineering:

1. Develop iteratively, with risk as the primary iteration driver
2. Manage Requirements
3. Employ component based architecture
4. Model Software Visually
5. Continuously verify quality
6. Control changes

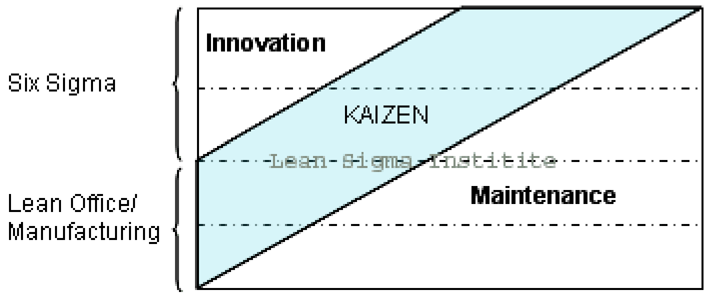
Chapter 18 – Six Sigma

Six Sigma

It's a business management strategy developed by Motorola.

Lean Six Sigma

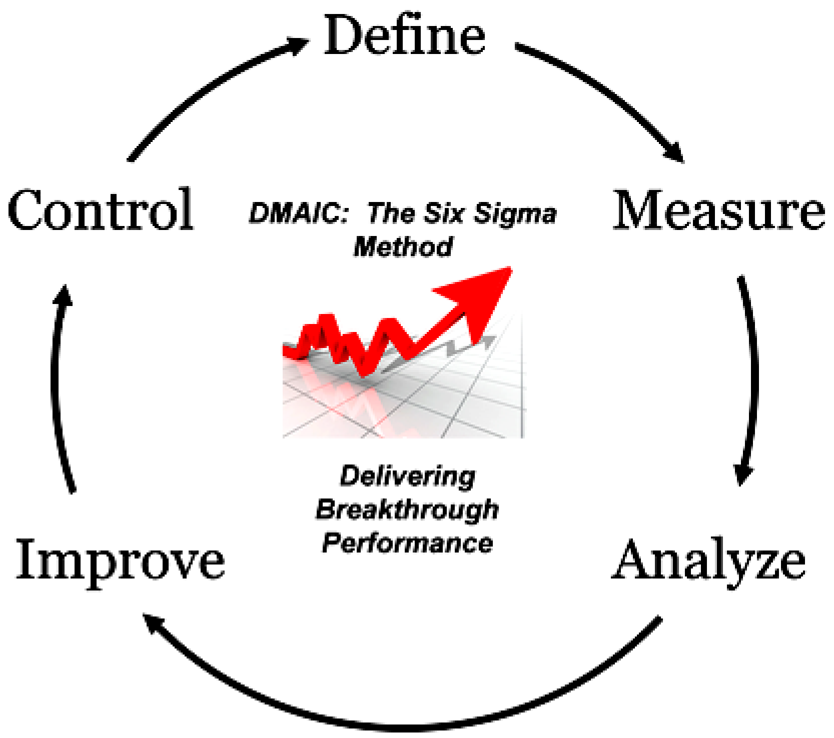
It's an integrated and balanced combination of the speed and variation reduction power of both Lean and Six Sigma to achieve business management process full optimization.



Six Sigma is deployed mainly for innovative, breakthrough and continual improvements under the black belt projects led by Black Belts and Master Black Belts while Lean is deployed mainly for daily continual improvements and performance sustaining activities under the lean kaizen events led by Line Engineers and Supervisors.

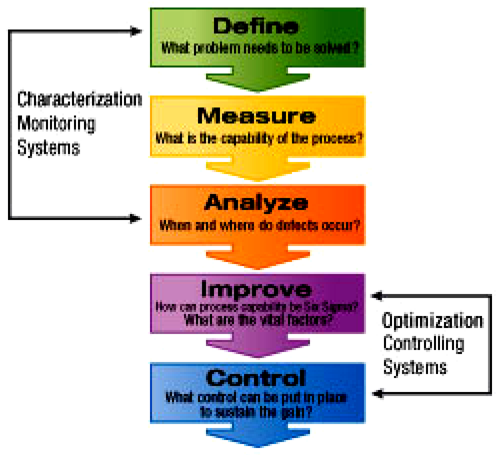
Besides, there are two project methodologies developed from Plan-Do-Check-Act and they are DMAIC and DMADV.

1. DMAIC is for improving an existing process.
2. DMADV is for creating new process designs.


Define
Measure
Analyze
Improve
Control
DMAIC: The Six Sigma Method
Delivering Breakthrough Performance

Six Sigma DMAIC

DMAIC, pronounced (De-May-Ick), refers to a data-driven quality strategy for improving processes, and is an integral part of the company's Six Sigma Quality Initiative. DMAIC is an acronym for five interconnected phases: Define, Measure, Analyze, Improve, and Control.


Define
What problem needs to be solved?
Measure
What is the capability of the process?
Analyze
When and where do defects occur?
Improve
How can process capability be Six Sigma?
What are the vital factors?
Control
What control can be put in place to sustain the gain?
Characterization Monitoring Systems
Optimization Controlling Systems

DMAIC

Define
What problem needs to be solved?

Measure
What is the capability of the process?

Analyze
When and where do defects occur?

Improve
How can process capability be Six Sigma?
What are the vital factors?

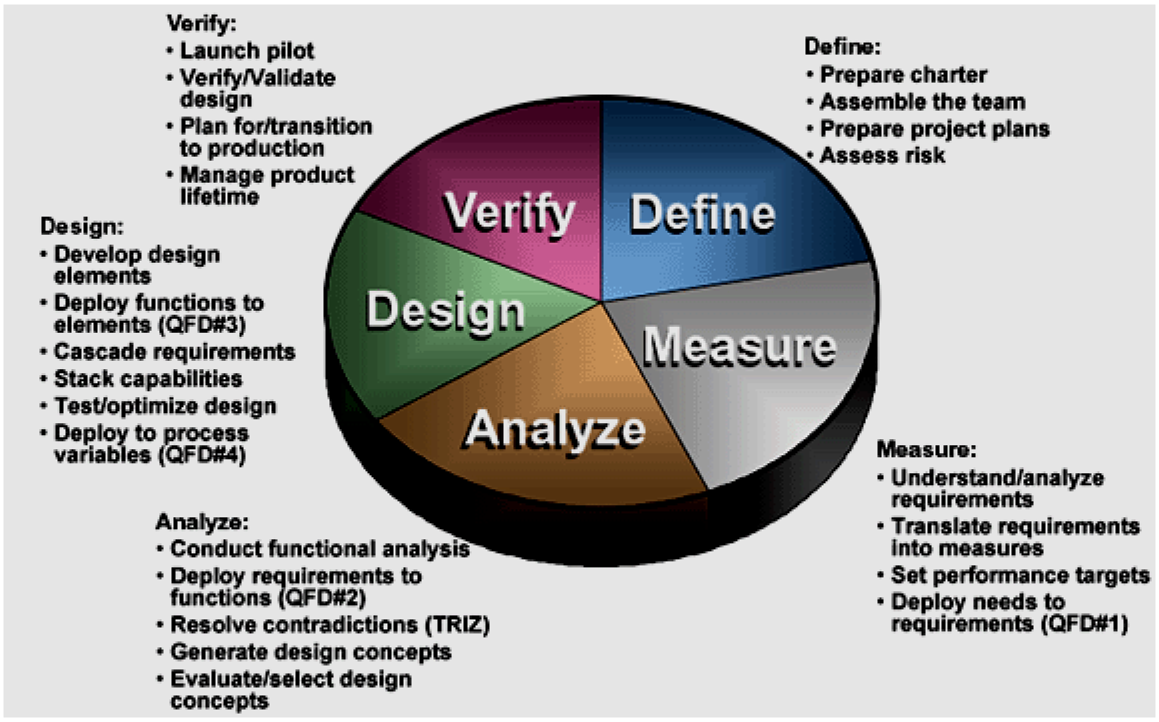
Control
What control can be put in place to sustain the gain?

This methodology has five phases:

- Define the problem
- Measure key aspects of the current process
- Analyze the data to investigate and verify cause-and-effect relationships
- Improve or optimize the current process based upon data analysis
- Control the future state process to ensure any deviations from target are corrected before they result in defects.

DMADV

DMADV (Define Measure Analyze Design Verify)

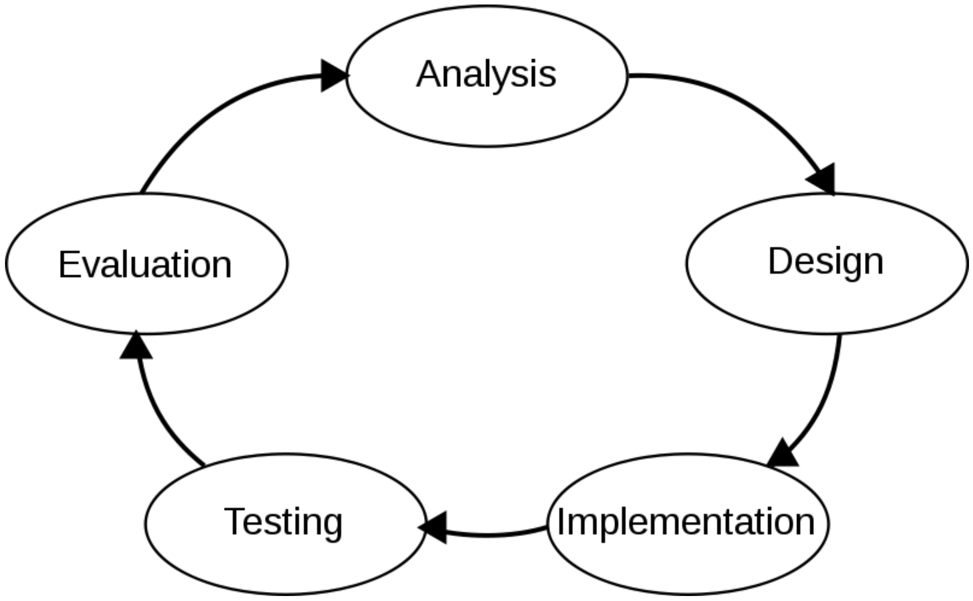

Verify:
• Launch pilot
• Verify/Validate design
• Plan for/transition to production
• Manage product lifetime
Define:
• Prepare charter
• Assemble the team
• Prepare project plans
• Assess risk
Design:
• Develop design elements
• Deploy functions to elements (QFD#3)
• Cascade requirements
• Stack capabilities
• Test/optimize design
• Deploy to process variables (QFD#4)
Verify
Define
Design
Measure
Analyze
Measure:
• Understand/analyze requirements
• Translate requirements into measures
• Set performance targets
• Deploy needs to requirements (QFD#1)
Analyze:
• Conduct functional analysis
• Deploy requirements to functions (QFD#2)
• Resolve contradictions (TRIZ)
• Generate design concepts
• Evaluate/select design concepts

Chapter 19 – Software Development Life Cycle (SDLC)

Software Development Life Cycle (SDLC)

The Systems development life cycle (SDLC), or Software development life cycle in systems engineering, information systems and software engineering, is a process of creating or altering information systems, and the models and methodologies that people use to develop these systems.

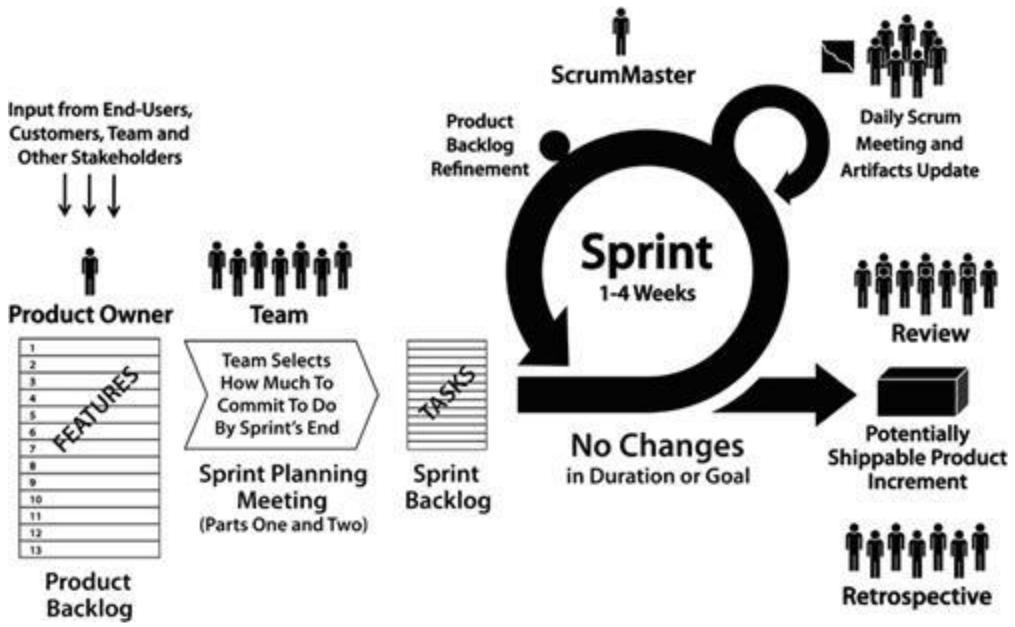
The SDLC is a process used by a systems analyst an development to develop an information system, including requirements, validation, training, and user (stakeholder) ownership. Any SDLC should result in a high quality system that meets or exceeds customer expectations, reaches completion within time and cost estimates, works effectively and efficiently in the current and planned Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance. Computer systems are complex and often (especially with the recent rise of service-oriented architecture) link multiple traditional systems potentially supplied by different software vendors. To manage this level of complexity, a number of SDLC models or methodologies have been created, such as "waterfall"; "spiral"; "Agile software development"; "rapid prototyping"; "incremental"; and "synchronize and stabilize".



Chapter 20 – Agile

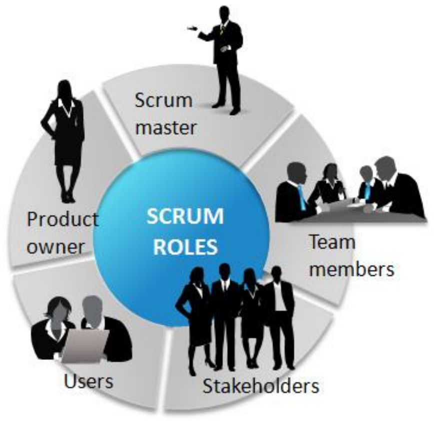
Agile Software Development

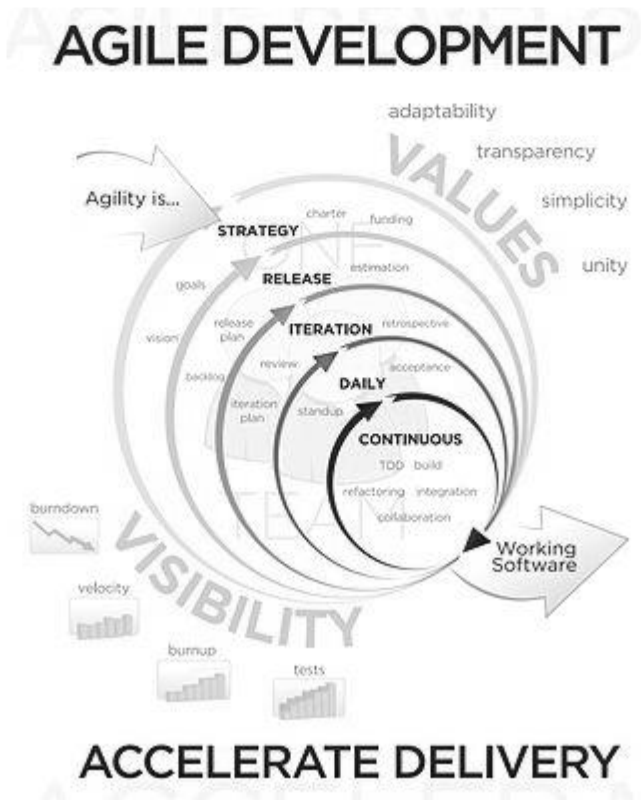
Agile is a way to quickly develop working applications by focusing on progressive requirements rather than processes. Agile development is done in iterative manner with short requirements, quick builds and frequent releases. Agile methodology when compared to traditional practices like waterfall model makes development easier, faster and adaptive.


Input from End-Users, Customers, Team and Other Stakeholders
Product Owner
FEATURES
1
2
3
4
5
6
7
8
9
10
11
12
13
Product Backlog
Team
Team Selects How Much To Commit To Do By Sprint's End
Sprint Planning Meeting
(Parts One and Two)
TASKS
Sprint Backlog
ScrumMaster
Product Backlog Refinement
Sprint
1-4 Weeks
No Changes
in Duration or Goal
Daily Scrum Meeting and Artifacts Update
Review
Potentially Shippable Product Increment
Retrospective

Scrum is a process skeleton that includes a set of practices and predefined roles. The main roles in scrum are the Scrum Master who maintains the processes and works similar to a project manager, the Product Owner who represents the stakeholders, and the Team which includes the developers.

Scrum has three major roles: Product Owner, Scrum Master, and the Team.

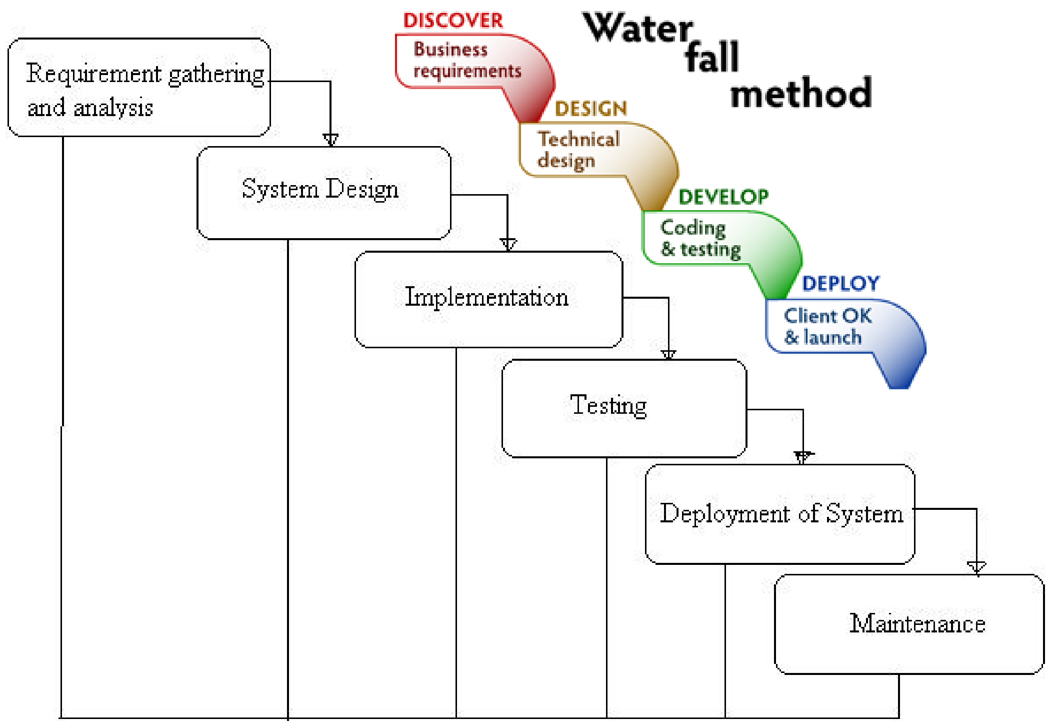




Chapter 21 – Waterfall Model

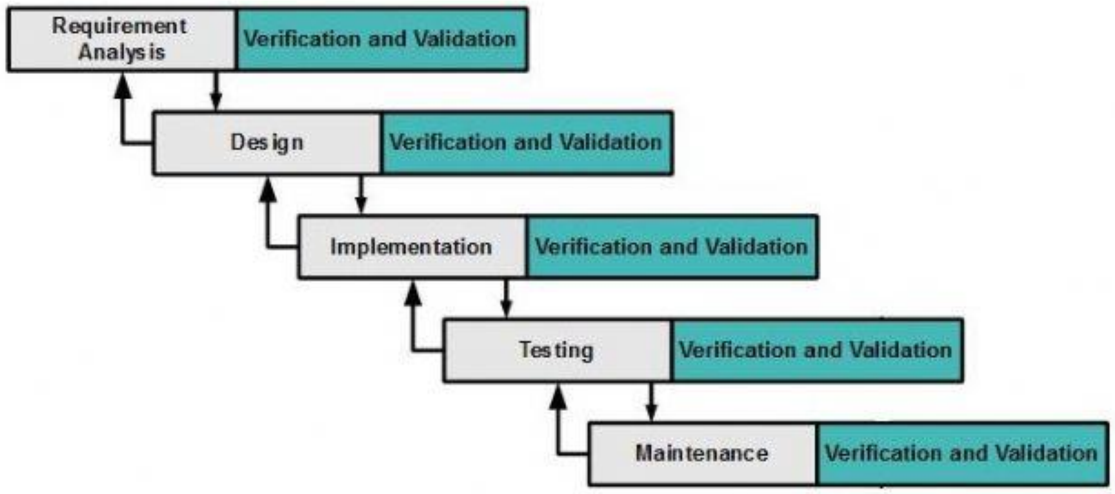
Waterfall Model

It's a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing and Maintenance.



Modified Waterfall Model

The Modified Waterfall method is a derivative of the traditional water fall model but with some minor variations relative to iterations between certain stages where the user requirements are validated and verified at the end of each phase. It involves validation and verification between the phases, so any deviations can be corrected immediately, providing the customer satisfaction, so this is preferred.


Requirement Analysis
Verification and Validation
Design
Verification and Validation
Implementation
Verification and Validation
Testing
Verification and Validation
Maintenance
Verification and Validation

Chapter 22 – Struts

Model-View-Controller (MVC)

Model-View-Controller (MVC) is a design pattern put together to help control change. MVC decouples interface from business logic and data.

- **Model:** The model contains the core of the application's functionality. The model encapsulates the state of the application. Sometimes the only functionality it contains is state. It knows nothing about the view or controller.
- **View:** The view provides the presentation of the model. It is the *look* of the application. The view can access the model getters, but it has no knowledge of the setters. In addition, it knows nothing about the controller. The view should be notified when changes to the model occur.
- **Controller:** The controller reacts to the user input. It creates and sets the model.

Struts Framework

A framework is made up of the set of classes which allow us to use a library in a best possible way for a specific requirement. Struts framework is an open-source framework for developing the web applications in Java EE, based on MVC-2 architecture. It uses and extends the Java Servlet API. Struts is robust architecture and can be used for the development of application of any size. Struts framework makes it much easier to design scalable, reliable Web applications with Java.

Struts 2

Struts2 is popular and mature web application framework based on the MVC design pattern. Struts2 is not just the next version of Struts 1, but it is a complete rewrite of the Struts architecture.

The Web Work framework started off with Struts framework as the basis and its goal was to offer an enhanced and improved framework built on Struts to make web development easier for the developers.

After some time, the Web work framework and the Struts community joined hands to create the famous Struts2 framework. The framework is designed to streamline the full development cycle, from building, to deploying, to maintaining applications over time. Apache Struts 2 was originally known as Web Work.

Important Features of Struts 2:

1. POJO forms and POJO actions
2. Tag support - Improved and Less Coding
3. AJAX support
4. Easy Integration with other Frameworks
5. Template Support for generating views
6. Plugin Support
7. Integrated Profiling and Debugging Tools
8. Easy to modify tags using Free marker templates (Basic web knowledge is enough)
9. Promote less configuration using default values for various settings

Differences between Struts 1 and Struts 2

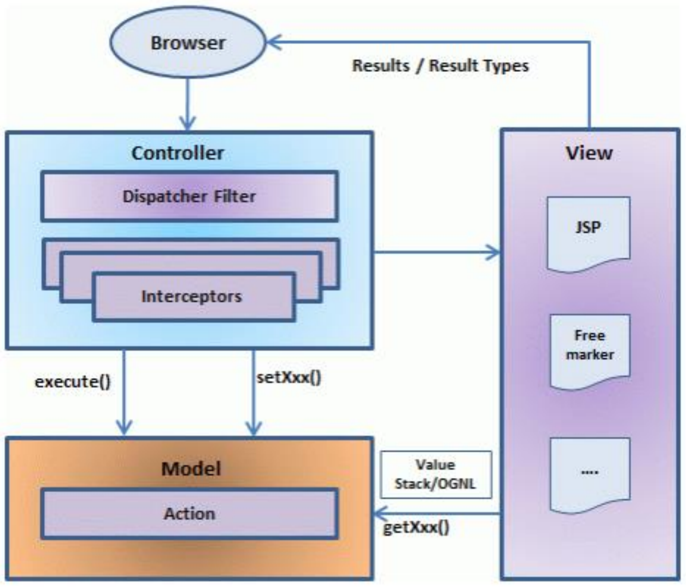
Struts 1	Struts 2
Action Class	
Abstract class	Interface
Threading Model	
Singleton and thread-safe	New instance and no thread safety issues
Servlet Dependency	
Depends on Servlet API	No dependency on Servlet API
Testability	
Unit test complexity	DI makes unit testing easier
Harvesting Input	
Redundant Action Forms	Eliminates redundancy using Action properties. Supports Action Forms too.
Expression Language	
Integrates with JSTL & EL	Integrates with JSTL & OGNL
Binding Values into Views	
Binds objects into the page context using JSP mechanism	Decouples reusable views from values using Value Stack technology
Type Conversion	
common-beanutils for type conversion	OGNL for type conversion
Validation	
Manual validation - validate method on the action form	Xwork Validation framework in addition to manual validation
Control of Action Execution	
Each module- separate Request Processor(lifecycles) - shared by all Actions in the module	Different lifecycles created on a per Action basis via Interceptor Stacks (Custom Stacks)

Struts 2 Architecture

From a high level, Struts2 is a pull-MVC (or MVC2) framework. The Model-View-Controller pattern in Struts2 is realized with following five core components:

1. Actions
2. Interceptors
3. Value Stack / OGNL
4. Results / Result types
5. View technologies

Struts2 is slightly different from a traditional MVC framework in that the action takes the role of the model rather than the controller, although there is some overlap.



The above diagram depicts the Model, View and Controller to the Struts2 high level architecture. The controller is implemented with a Struts2 dispatch servlet filter as well as interceptors; the model is implemented with actions and the view as a combination of result types and results. The value stack and OGNL provide common thread, linking and enabling integration between the other components.

Apart from the above components, there will be a lot of information that relates to configuration. Configuration for the web application, as well as configuration for actions, interceptors, results, etc.

Request Life Cycle

Based on the above diagram, one can explain the user's request life cycle in Struts 2 as follows:

1. User sends a request to the server for requesting for some resource (i.e. pages).
2. The Filter Dispatcher looks at the request and then determines the appropriate Action.
3. Configured interceptors functionalities apply such as validation, file upload etc.
4. Selected action is executed to perform the requested operation.
5. Again, configured interceptors are applied to do any post-processing if required.
6. Finally the result is prepared by the view and returns the result to the user.

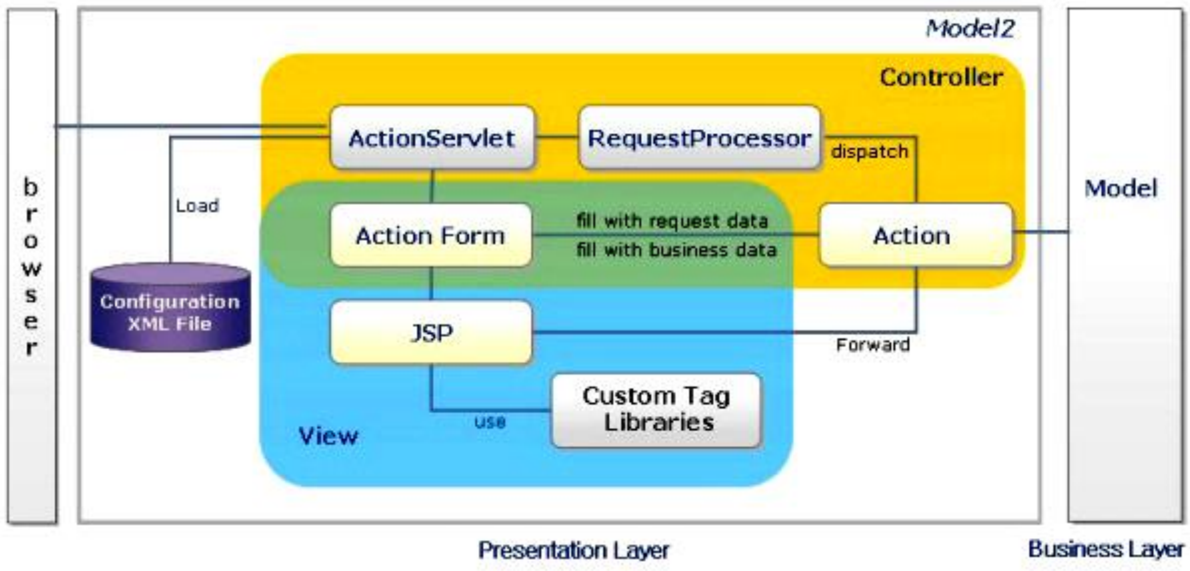
Struts 1 Architecture

Struts components can be categorize into Model, View and Controller:

- **Model:** Components like business logic /business processes and data are the part of model.
- **View:** HTML, JSP are the view components.
- **Controller:** Action Servlet of Struts is part of Controller components which works as front controller to handle all the requests.

Struts is a set of cooperating classes, servlets, and JSP tags that make up a reusable MVC 2 design.

- JavaBeans components for managing application state and behavior.
- Event-driven development (via listeners as in traditional GUI development).
- Pages that represent MVC-style views; pages reference view roots via the JSF component tree.

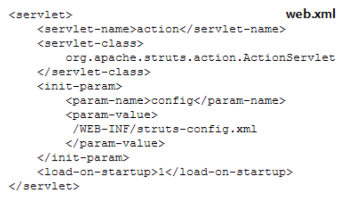


Request Life Cycle

Executed in the following order when request is sent by client

1. When web application starts, Struts configuration file (struts-config.xml) loading is performed using Action Servlet set in web.xml.
2. Decide Action Mapping according to request from Request Processor defined in struts-config.xml.
3. Execute execute() method of Action class according to URL and action mapping information defined in struts-config.xml.
4. Action execute() method calls business logic by connecting to business layer.
5. Return Action Forward according to business logic execution result, and Controller performs forwarding with the appropriate view according to return value.

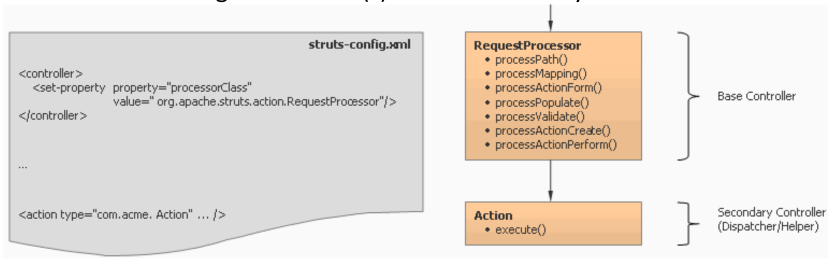
ActionServlet
ActionServlet is a simple servlet which is the backbone of all Struts applications. It is the main Controller component that handles client requests and determines which Action will process each received request. It serves as an Action factory that helps to create specific Action classes based on user's request.



web.xml

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
        org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>
         /WEB-INF/struts-config.xml
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

RequestProcessor
RequestProcessor contains the processing logic that the Struts controller servlet performs as it receives each servlet request from the container. You can customize the request processing behavior by subclassing this class and overriding the method(s) whose behavior you are interested in changing.



struts-config.xml

```
<controller>
   <set-property  property="processorClass"
                  value=" org.apache.struts.action.RequestProcessor"/>
</controller>

...

<action type="com.acme. Action" ... />
```

Action
An Action Class performs a role of an adapter between the contents of an incoming HTTP request and the corresponding business logic that should be executed to process this request.

```
public  ActionForward execute( ActionMapping  mapping,  ActionForm form, HttpServletRequest req, HttpServletResponse res)
                                        throws Exception ;
```

The different kinds of actions in Struts are ForwardAction, IncludeAction, DispatchAction, LookupDispatchAction and SwitchAction.

ActionMapping
Action mapping contains all the deployment information for a particular Action bean. This class is to determine where the results of the Action will be sent once its processing is complete.

ActionForm
ActionForm is javabean which represents the form inputs containing the request parameters from the View referencing the Action bean. The important methods of ActionForm are validate() and reset().

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
```

The lifecycle of ActionForm invoked by the RequestProcessor is as follows:

- Retrieve or Create Form Bean associated with Action
- "Store" FormBean in appropriate scope (request or session)
- Reset the properties of the FormBean
- Populate the properties of the FormBean

- Validate the properties of the FormBean
- Pass FormBean to Action

DynaActionForm
It's a specialized subclass of ActionForm that allows the creation of form beans with dynamic sets of properties (configured in configuration file), without requiring the developer to create a Java class for each type of form bean.

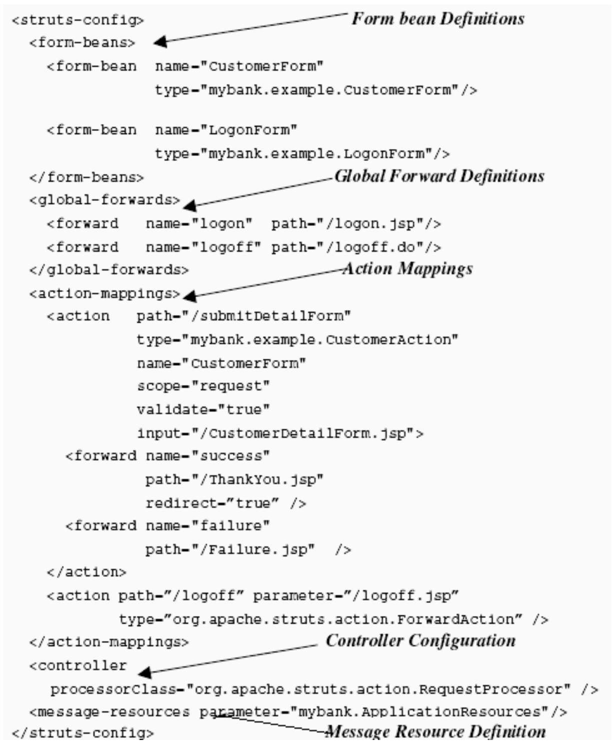
```
<form-bean name="loginForm"type="org.apache.struts.action.DynaActionForm" >
  <form-property name="userName" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String" />
</form-bean>
```

ActionErrors
A class that encapsulates the error messages being reported by the validate() method of an ActionForm. Validation errors are either global to the entire ActionForm bean they are associated with, or they are specific to a particular bean property. Each individual error is described by an ActionMessage object, which contains a message key.

ActionForward
An ActionForward represents a destination to which the controller, RequestProcessor, might be directed to perform a RequestDispatcher.forward or HttpServletResponse.sendRedirect to, as a result of processing activities of an Action class.

Struts Tag Libraries
The various Struts tag libraries are HTML Tags, Bean Tags, Logic Tags, Template Tags, Nested Tags and Tiles Tags.

Struts Configuration File Structure



```
<struts-config>
  <form-beans>
    <form-bean  name="CustomerForm"
                type="mybank.example.CustomerForm"/>

    <form-bean  name="LogonForm"
                type="mybank.example.LogonForm"/>
  </form-beans>
  <global-forwards>
    <forward   name="logon"  path="/logon.jsp"/>
    <forward   name="logoff" path="/logoff.do"/>
  </global-forwards>
  <action-mappings>
    <action   path="/submitDetailForm"
              type="mybank.example.CustomerAction"
              name="CustomerForm"
              scope="request"
              validate="true"
              input="/CustomerDetailForm.jsp">
      <forward name="success"
               path="/ThankYou.jsp"
               redirect="true" />
      <forward name="failure"
               path="/Failure.jsp"  />
    </action>
    <action path="/logoff" parameter="/logoff.jsp"
            type="org.apache.struts.action.ForwardAction" />
  </action-mappings>
  <controller
    processorClass="org.apache.struts.action.RequestProcessor" />
  <message-resources parameter="mybank.ApplicationResources"/>
</struts-config>
```

Chapter 23 – Spring

Inversion of Control (IOC)

The basic concept of the Inversion of Control pattern (also known as dependency injection) is that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (in the case of the Spring framework, the IOC container) is then responsible for hooking it all up.

i.e., Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

Hollywood Principle - Don't call me, I will call you

Different Types of IOC

There are three types of dependency injection:

- **Constructor Injection** (e.g. Pico container, Spring etc): Dependencies are provided as constructor parameters.
- **Setter Injection** (e.g. Spring): Dependencies are assigned through JavaBean properties (ex: setter methods).
- **Interface Injection** (e.g. Avalon): Injection is done through an interface.

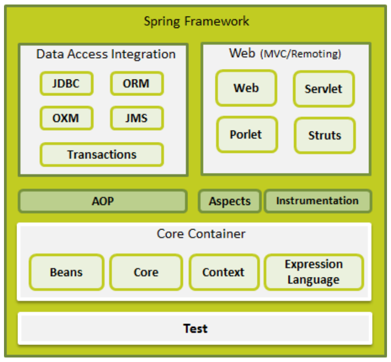
  Note: Spring supports only Constructor and Setter Injection

Benefits of IOC (Dependency Injection)

Benefits of IOC are as follows:

- Minimizes the amount of code in your application.
- Make your application more testable by not requiring any singletons or JNDI lookup mechanisms in your unit test cases.
- Loose coupling is promoted with minimal effort and least intrusive mechanism.
- IOC containers support eager instantiation and lazy loading of services.

Spring is an open source framework created to address the complexity of enterprise application development. One of the chief advantages of the Spring framework is its layered architecture, which allows you to be selective about which of its components you use while also providing a cohesive framework for J2EE application development.



Advantages of Spring

The advantages of Spring framework are as follows:

- Spring has layered architecture. Use what you need and leave you don't need now.
- Spring enables POJO Programming. There is no behind the scene magic here. POJO programming enables continuous integration and testability.
- Dependency Injection and Inversion of Control simplifies JDBC
- Open source and no vendor lock-in.

Features of Spring

- **Lightweight:**
  Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.
- **Inversion of control (IOC):**
  Loose coupling is achieved in spring using the technique Inversion of Control. The objects give their dependencies instead of creating or looking for dependent objects.
- **Aspect oriented (AOP):**
  Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services.
- **Container:**
  Spring contains and manages the life cycle and configuration of application objects.
- **MVC Framework:**
  Spring comes with MVC web application framework, built on core Spring functionality. This framework is highly configurable via strategy interfaces, and accommodates multiple view technologies like JSP, Velocity, Tiles, iText, and POI. But other frameworks can be easily used instead of Spring MVC Framework.
- **Transaction Management:**
  Spring framework provides a **generic abstraction layer** for transaction management. This allows the developer to add the pluggable transaction managers, and making it easy to demarcate

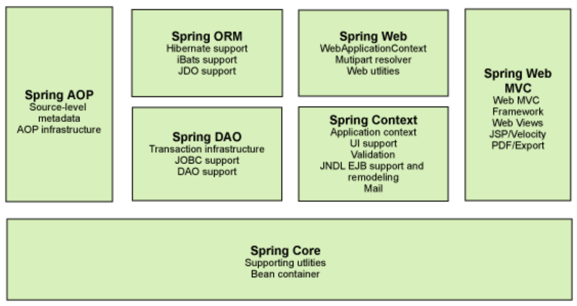
transactions without dealing with low-level issues. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.

- **JDBC Exception Handling:**

  The JDBC abstraction layer of the Spring offers a meaningful **exception hierarchy**, which simplifies the error handling strategy. Integration with Hibernate, JDO, and iBATIS: Spring provides best Integration services with Hibernate, JDO and iBATIS

Seven Modules:

1. Spring Core - IOC-DI/Bean Factory Container
2. Spring AOP – Cross cutting concerns such as logging, auditing, transactions, security and caching
3. Spring MVC – Model View Controller
4. Spring ORM – Integration with Hibernate, iBatis and JDO
5. Spring DAO - JDBC abstraction layer and exception handling
6. Spring Web - Web oriented integration (Web Application Context and Multipart Resolver)
7. Spring Context – Application Context, UI support, Validation, JNDI, EJB and Mail support



Bean Definition

The objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions which you have already seen in previous chapters.

The bean definition contains the information called **configuration metadata** which is needed for the container to know the followings:

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

Properties	Description
class	This attribute is mandatory and specify the bean class to be used to create the bean.
name	This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
scope	This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
constructor-arg	This is used to inject the dependencies using constructor arguments.
properties	This is used to inject the dependencies using the setter methods.
autowiring mode	This is used to inject the dependencies in auto-wiring mode (autowire)
lazy-initialization mode	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup (lazy-init)
initialization method	A callback to be called just after all necessary properties on the bean has been set by the container. It will be discussed in bean life cycle chapter (init-method)
destruction method	A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter (destroy-method)

Bean Scopes

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware Application Context (Web Application only). The first two scopes can be used for both stand-alone and web applications.

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default). This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.
prototype	This scopes a single bean definition to have any number of object instances. If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made. **As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans**.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring Application Context.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring Application Context.

global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring Application Context. Typically only valid when used in a **portlet context**.

Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. There are following three important methods to provide configuration metadata to the Spring Container:

1. XML-based configuration file (Declarative)
2. Annotation-based configuration (Declarative)
3. Java-based configuration (Programmatic)

Bean Life Cycle

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there is a list of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, but this chapter will discuss only two important bean lifecycle callback methods which are required at the time of bean initialization and its destruction.

To define setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.

Initialization callbacks:

The *org.springframework.beans.factory.***InitializingBean** interface specifies a single method.

```
void afterPropertiesSet() throws Exception;
```

So you can simply implement above interface and initialization work can be done inside afterPropertiesSet() method as follows:

```
public class ExampleBean implements InitializingBean {
  public void afterPropertiesSet() {
    // do some initialization work
  }
}
```

In the case of XML-based configuration metadata, you can use the **init-method** attribute to specify the name of the method that has a void no-argument signature. For example:

```
<bean id="exampleBean"  class="examples.ExampleBean" init-method="initialize"/>
```

Following is the class definition:

```
public class ExampleBean {
  public void initialize() {
    // do some initialization work
  }
}
```

Destruction callbacks

The *org.springframework.beans.factory.***DisposableBean** interface specifies a single method.

```
void destroy() throws Exception;
```

So you can simply implement above interface and finalization work can be done inside destroy() method as follows:

```
public class ExampleBean implements DisposableBean {
  public void destroy() {
    // do some destruction work
  }
}
```

In the case of XML-based configuration metadata, you can use the **destroy-method** attribute to specify the name of the method that has a void no-argument signature. For example:

```
<bean id="exampleBean" class="examples.ExampleBean" destroy-method="dispose"/>
```

Following is the class definition:

```
public class ExampleBean {
  public void dispose() {
    // do some destruction work
  }
}
```

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

Bean Post Processor

The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own **instantiation logic, dependency-resolution logic** etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the **order** property provided the BeanPostProcessor implements the **Ordered** interface.

The BeanPostProcessors operate on bean (or object) instances which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An **ApplicationContext** automatically detects any beans that are defined with implementation of the **BeanPostProcessor** interface and registers these beans as post-processors, to be then called appropriately by the container upon bean creation.

```
public class InitHelloWorld implements BeanPostProcessor {

  public Object postProcessBeforeInitialization(Object bean,
           String beanName) throws BeansException {
    System.out.println("BeforeInitialization : " + beanName);
    return bean;  // you can return any other object as well
  }

  public Object postProcessAfterInitialization(Object bean,
           String beanName) throws BeansException {
    System.out.println("AfterInitialization : " + beanName);
    return bean;  // you can return any other object as well
  }

}
```

```
    AbstractApplicationContext context =
                  new ClassPathXmlApplicationContext("Beans.xml");

    HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
    obj.getMessage();
    context.registerShutdownHook();
```

```
  <bean id="helloWorld" class="com.mesonsoft.HelloWorld"
     init-method="init" destroy-method="destroy">
     <property name="message" value="Hello World!"/>
  </bean>

  <bean class="com.mesonsoft.InitHelloWorld" />
```

Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but inheritance concept is same. You can define a parent bean definition as a template and other child beans can inherit required configuration from the parent bean.

When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent** attribute, specifying the parent bean as the value of this attribute.

```
<bean id="helloWorld" class="com.mesonsoft.HelloWorld">
   <property name="message1" value="Hello World!"/>
   <property name="message2" value="Hello Second World!"/>
</bean>

<bean id="helloIndia" class="com.mesonsoft.HelloIndia"
   parent="helloWorld">
   <property name="message1" value="Hello India!"/>
   <property name="message3" value="Namaste India!"/>
</bean>
```

Bean Definition Template

You can create a Bean definition template which can be used by other child bean definitions without putting much effort. While defining a Bean Definition Template, you should not specify **class** attribute and should specify **abstract** attribute with a value of **true** as shown below:

```
<bean id="beanTeamplate" abstract="true">
   <property name="message1" value="Hello World!"/>
   <property name="message2" value="Hello Second World!"/>
   <property name="message3" value="Namaste India!"/>
</bean>

<bean id="helloIndia" class="com.mesonsoft.HelloIndia"
   parent="beanTeamplate">
   <property name="message1" value="Hello India!"/>
   <property name="message3" value="Namaste India!"/>
</bean>
```

Injection Inner Beans

As you know Java inner classes are defined within the scope of other classes, similarly, **inner beans** are beans that are defined within the scope of another bean. Thus, a <bean/> element inside the <**property**/> or <**constructor-arg**/> elements is called inner bean and it is shown below.

```
<bean id="outerBean" class="...">
  <constructor-arg>
       <bean id="innerBean" class="..."/>  //CONSTRUCTOR INJECTION
  </constructor-arg>
   <property name="target">
       <bean id="innerBean" class="..."/>  //SETTER INJECTION
  </property>
</bean>
```

Injecting Collection

You have seen how to configure **primitive data type** using **value** attribute and **object references** using **ref** attribute of the **<property>** tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.

Now what about if you want to pass plural values like Java Collection types **List, Set, Map,** and **Properties**. To handle the situation, Spring offers four types of collection configuration elements which are as follows:

Element	Description
<list>	This helps in wiring i.e. injecting a list of values, **allowing duplicates**.
<set>	This helps in wiring a set of values but **without any duplicates**.
<map>	This can be used to inject a collection of name-value pairs where name and value can be of **any type**.
<props>	This can be used to inject a collection of name-value pairs where the name and value are **both Strings**.

You can use either <list> or <set> to wire any implementation of java.util.**Collection** or an **array**.

```
<!-- Definition for javaCollection -->
<bean id="javaCollection" class="com.mesonsoft.JavaCollection">
  <!-- results in a setAddressList(java.util.List) call -->
  <property name="addressList">
    <list>
       <value>INDIA</value>
       <value>USA</value>
       <value>USA</value>
    </list>
  </property>

 <!-- results in a setAddressSet(java.util.Set) call -->
 <property name="addressSet">
    <set>
       <value>INDIA</value>
       <value>USA</value>
       <value>USA</value>
    </set>
  </property>

 <!-- results in a setAddressMap(java.util.Map) call -->
 <property name="addressMap">
    <map>
       <entry key="1" value="INDIA"/>
       <entry key="3" value="USA"/>
       <entry key="4" value="USA"/>
    </map>
  </property>

 <!-- results in a setAddressProp(java.util.Properties) call -->
 <property name="addressProp">
    <props>
       <prop key="one">INDIA</prop>
       <prop key="three">USA</prop>
       <prop key="four">USA</prop>
    </props>
  </property>
</bean>
```

Spring Beans Auto-Wiring

You have learnt how to declare beans using the <bean> element and inject <bean> with using <constructor-arg> and <property> elements in XML configuration file.

The Spring container can **autowire** relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.

Autowiring Modes:

There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the **autowire** attribute of the <bean/> element to specify autowire mode for a bean definition.

Mode	**Description**
no	This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter.
byName	Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byName* in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
byType	Autowiring by property datatype. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byType* in the XML configuration file. It then tries to match and wire a property if its **type** matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown.
constructor	Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
autodetect	Spring first tries to wire using autowire by ***constructor***, if it does not work, Spring tries to autowire by ***byType***.

You can use **byType** or **constructor** autowiring mode to wire **arrays** and other typed-**collections**.

Limitations with autowiring:

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

Limitations	**Description**
Overriding possibility	You can still specify dependencies using <constructor-arg> and <property> settings which will always override auto-wiring.

Primitive data types	You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
Confusing nature	Auto-wiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

Annotation Based Configuration

Starting from Spring 2.5, it became possible to configure the dependency injection using **annotations**. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So, consider to have following configuration file, in case you want to use any annotation in your Spring application.

```
  <context:annotation-config/>
  <!-- bean definitions go here -->
</beans>
```

Once **<context:annotation-config/>** is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us see few important annotations to understand how they work:

S.N.	Annotation & Description
1	@Required The @Required annotation applies to bean property setter methods.
2	@Autowired The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.
3	@Qualifier The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
4	JSR-250 Annotations Spring supports JSR-250 based annotations which include **@Resource**, **@PostConstruct** and **@PreDestroy** annotations.

Java Based Configuration

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations explained below.

@Configuration & @Bean Annotations:

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

Lifecycle Callbacks:

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element

Specifying Bean Scope:

The default scope is singleton, but you can override this with the @Scope annotation as given below.

The @Import Annotation:

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as given below.

```
@Configuration
@Import(ConfigA.class)
@ImportResource(spring.xml"})
public class HelloWorldConfig {

  @Bean
  @Scope("prototype")
  public HelloWorld helloWorld(){
    return new HelloWorld();
  }

  @Bean(initMethod = "initialize", destroyMethod = "cleanup" )
  public Foo foo() {
    return new Foo();
  }

 public class Foo {
   public void initialize() {
     // initialization logic
   }
   public void cleanup() {
     // destruction logic
   }
 }
}
```

Above Java code will be equivalent to the following XML configuration:

```
  <bean id="helloWorld" class="com.mesonsoft.HelloWorld" scope="prototype" init-method="initialize" destroy-method="cleanup"
/>
```

Injecting Bean Dependencies:

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method calling another as follows:

```
@Configuration
public class AppConfig {
  @Bean
  public Foo foo() {
    Foo foo = new Foo (bar()); // CONSTRUCTOR INJECTION
    Foo foo = new Foo ();
    foo.setBar(bar());           // SETTER INJECTION
    return foo;
  }
  @Bean
  public Bar bar() {
    return new Bar();
  }
}
```

Event Handling in Spring

The core of Spring framework is the **ApplicationContext**, which manages complete life cycle of the beans. The ApplicationContext publishes certain types of events when loading the beans. For example, a *ContextStartedEvent* is published when the context is started and *ContextStoppedEvent* is published when the context is stopped.

Event handling in the *ApplicationContext* is provided through the **ApplicationEvent** class and **ApplicationListener** interface. So if a bean implements the ApplicationListener, then every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

Spring provides the following standard events:

S.N.	Spring Built-in Events & Description
1	**ContextRefreshedEvent** This event is published when the *ApplicationContext* is either initialized or refreshed. This can also be raised using the **refresh**() method on the **ConfigurableApplicationContext** interface.
2	**ContextStartedEvent** This event is published when the *ApplicationContext* is started using the **start**() method on the **ConfigurableApplicationContext** interface. You can poll your database or you can restart any stopped application after receiving this event.
3	**ContextStoppedEvent** This event is published when the *ApplicationContext* is stopped using the **stop**() method on the **ConfigurableApplicationContext** interface.
4	**ContextClosedEvent** This event is published when the *ApplicationContext* is closed using the **close**() method on the **ConfigurableApplicationContext** interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
5	**RequestHandledEvent** This is a web-specific event telling all beans that an HTTP request has been serviced.

Spring's event handling is **single-threaded** so if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. Hence, care should be taken when designing your application if event handling is to be used.

Listening to Context Events:

To listen a context event, a bean should implement the **ApplicationListener** interface which has just one method **onApplicationEvent()**. So let us write an example to see how the events propagates and how you can put your code to do required task based on certain events.

```
public class CStartEventHandler
  implements ApplicationListener<ContextStartedEvent>{

  public void onApplicationEvent(ContextStartedEvent event) {
    System.out.println("ContextStartedEvent Received");
  }
}
```

```
    ConfigurableApplicationContext context =
    new ClassPathXmlApplicationContext("Beans.xml");

    // Let us raise a start event.
    context.start();

    HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

    obj.getMessage();

    // Let us raise a stop event.
    context.stop();
```

```
  <bean id="cStartEventHandler"   class="com.mesonsoft.CStartEventHandler"/>
```

Custom Events in Spring

There are number of steps to be taken to write and publish your own custom events. Follow the instructions given below to **write**, **publish** and **handle** Custom Spring Events.

Step	Description
1	Create an event class, *CustomEvent* by extending **ApplicationEvent**. This class must define a default constructor which should **inherit** constructor from ApplicationEvent class.
2	Once your event class is defined, you can publish it from any class, let us say *CustomEventPublisher* which implements **ApplicationEventPublisherAware**. You will also need to declare these classes in XML configuration file as a bean so that the container can identify the bean as an event publisher because it implements the ApplicationEventPublisherAware interface.
3	A published event can be handled in a class, let us say *CustomEventHandler* which implements **ApplicationListener** interface and implements **onApplicationEvent** method for the custom event.

```
public class CustomEvent extends ApplicationEvent{

  public CustomEvent(Object source) {
    super(source);
  }

  public String toString(){
    return "My Custom Event";
  }
}
```

```
public class CustomEventPublisher
  implements ApplicationEventPublisherAware {

  private ApplicationEventPublisher publisher;

  public void setApplicationEventPublisher
          (ApplicationEventPublisher publisher){
    this.publisher = publisher;
  }

  public void publish() {
    CustomEvent ce = new CustomEvent(this);
    publisher.publishEvent(ce);
  }
}
```

```
public class CustomEventHandler
  implements ApplicationListener<CustomEvent>{

  public void onApplicationEvent(CustomEvent event) {
    System.out.println(event.toString());
  }
}
```

```
    CustomEventPublisher cvp = (CustomEventPublisher) context.getBean("customEventPublisher");
    cvp.publish();
```

```
  <bean id="customEventHandler"  class="com.mesonsoft.CustomEventHandler"/>
  <bean id="customEventPublisher"  class="com.mesonsoft.CustomEventPublisher"/>
```

AOP with Spring Framework

One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. Aspect Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like **logging, auditing, declarative transactions, security, and caching** etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect.

Spring AOP module provides **interceptors** to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

AOP Terminologies:

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

Terms	Description
Aspect	A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
Join point	This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
Advice	This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework.
Pointcut	This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns.
Introduction	An introduction allows you to add new methods or attributes to existing classes.
Target object	The object being advised by one or more aspects, this object will always be a proxied object. Also referred to as the advised object.
Weaving	Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

Types of Advice

Spring aspects can work with five kinds of advice mentioned below:

Advice	Description
before	Run advice before the method execution.
after	Run advice after the method execution regardless of its outcome.
after-returning	Run advice after the method execution only if method completes successfully
after-throwing	Run advice after the method execution only if method exits by throwing an exception.
around	Run advice before and after the advised method is invoked.

Advice	Interface	Overriding Method
before	MethodBeforeAdvice	before(Method method, Object[] args, Object target) throws Throwable
after		
after-returning	AfterReturningAdvice	afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable
after-throwing	ThrowsAdvice	afterThrowing(IllegalArgumentException e) throws Throwable
around	MethodInterceptor	invoke(MethodInvocation methodInvocation) throws Throwable Call methodInvocation.proceed() too

```
<bean id="customerService" class="com.mesonsoft.customer.services.CustomerService"/>
<bean id="customerServiceInterceptor" class="com.mesonsoft.CustomerServiceInterceptor"/>
<bean id="customerServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="customerService" />
        <property name="interceptorNames">
                <list>
                     <value>customerServiceInterceptor</value>
                </list>
        </property>
</bean>
```

To use Spring proxy, you need to add CGLIB2 library.

```
<dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2.2</version>
</dependency>
```

Custom Aspects Implementation

Spring supports the **@AspectJ annotation style** approach and the **schema-based** approach to implement custom aspects.

Approach	Description
XML Schema based	Aspects are implemented using regular classes along with XML based configuration as shown above.
@AspectJ based	@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

Bean Factory (Spring Core Module)

A BeanFactory is like a factory class that contains a collection of beans. The BeanFactory holds bean definitions of multiple beans within itself and then instantiates the bean whenever asked for by the clients.

- BeanFactory is able to create associations between collaborating objects as they are instantiated. This removes the burden of configuration from bean itself and the beans client.
- BeanFactory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.

Application Context (Spring Context Module)

A bean factory is fine to simple applications, but to take advantage of the full power of the Spring framework, you may want to move up to Spring's more advanced container, the Application Context. On the surface, an application context is same as a bean factory. Both load bean definitions, wire beans together, and dispense beans upon request. But it also:

- Provides a means for resolving text messages, including support for internationalization of those messages.
- Provides a generic way to load file resources, such as images.
- Publishes events to beans that are registered as listeners.
- Certain operations on the container or beans in the container can be handled both programmatically and declaratively.

The three commonly used implementation of 'Application Context' are

- **ClassPathXmlApplicationContext :** It loads context definition from an XML file located in the **classpath**, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code.

  ```
  ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
  ```

- **FileSystemXmlApplicationContext :** It loads context definition from an XML file in the **filesystem**. The application context is loaded from the file system by using the code.

  ```
  ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
  ```

- **XmlWebApplicationContext :** It loads context definition from an XML file contained **within a web application**.

Differences between Bean Factory and Application Context

On the surface, an application context is same as a bean factory. But application context offers much more.

- Application contexts provide a means for resolving text messages, including support for i18n of those messages.

- Application contexts provide a generic way to load file resources, such as images.
- Application contexts can publish events to beans that are registered as listeners.
- Certain operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context.
- ResourceLoader support: Spring's Resource interface us a flexible generic abstraction for handling low-level resources. An application context itself is a ResourceLoader, Hence provides an application with access to deployment-specific Resource instances.
- MessageSource support: The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable

Spring JDBC Template

Spring JDBC provides several approaches and correspondingly different classes to interface with the database. I'm going to take classic and the most popular approach which makes use of **JdbcTemplate** class of the framework. This is the central framework class that manages all the database communication and exception handling.

JdbcTemplate Class

The JdbcTemplate class executes SQL queries, updates statements and stored procedure calls, performs iteration over resultsets and extracts the returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the *JdbcTemplate* class are ***threadsafe*** once configured. So you can configure a single instance of a *JdbcTemplate* and then safely inject this shared reference into multiple DAOs.

A common practice when using the JdbcTemplate class is to configure a *DataSource* in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

Configuring Data Source

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
  <property name="username" value="root"/>
  <property name="password" value="password"/>
</bean>
```

Data Access Object (DAO)

DAO stands for data access object which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way.

Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operations on database tables using SQL and jdbcTemplate object.

Querying for an integer:

```
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Querying for a long:

```
String SQL = "select count(*) from Student";
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

A simple query using a bind variable:

```
String SQL = "select age from Student where id = ?";
int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Querying for a String:

```
String SQL = "select name from Student where id = ?";
String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class);
```

Querying and returning an object:

```
String SQL = "select * from Student where id = ?";
Student student = jdbcTemplateObject.queryForObject(SQL,
            new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
  public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
    Student student = new Student();
    student.setID(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));
    return student;
  }
}
```

Querying and returning multiple objects:

```
String SQL = "select * from Student";
List<Student> students = jdbcTemplateObject.query(SQL,
                    new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
  public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
    Student student = new Student();
    student.setID(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));
    return student;
  }
}
```

Inserting a row into the table:

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Updating a row into the table:

```
String SQL = "update Student set name = ? where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Deleting a row from the table:

```
String SQL = "delete Student where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{20} );
```

Executing DDL Statements

You can use the **execute(..)** method from *jdbcTemplate* to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table:

```
String SQL = "CREATE TABLE Student( " +
  "ID   INT NOT NULL AUTO_INCREMENT, " +
  "NAME VARCHAR(20) NOT NULL, " +
  "AGE  INT NOT NULL, " +
  "PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );
```

Context Configuration Annotations

These annotations are used by Spring to guide creation and injection of beans.

Annotation	Use	Description
@Autowired	Constructor, Field, Method	Declares a constructor, field, setter method, or configuration method to be autowired by type. Items annotated with @Autowired do not have to be public.
@Configurable	Type	Used with to declare types whose properties should be injected, even if they are not instantiated by Spring. Typically used to inject the properties of domain objects.
@Order	Type, Method, Field	Defines ordering, as an alternative to implementing the org. springframework.core.Ordered interface.
@Qualifier	Field, Parameter, Type, Annotation Type	Guides autowiring to be performed by means other than by type.
@Required	Method (setters)	Specifies that a particular property must be injected or else the configuration will fail.
@Scope	Type	Specifies the scope of a bean, singleton, prototype, request, session, or some custom scope.

Autowiring without Setter Methods

The XML Configuration file should have <**context:annotation-config**> tag to activate the annotation based configuration. @Autowired can be used on any method (not just setter methods). The wiring can be done through any method, as illustrated here:

```
@Configurable(autowire = Autowire.BY_TYPE)
public class Treasuse {
        @Autowired
        public void directionsToTreasure(TreasureMap treasureMap) {
                this.treasureMap = treasureMap;
        }
}
```

And even on member variables:

```
@Autowired
private TreasureMap treasureMap;
```

To resolve any autowiring ambiguity, use the @Qualifier attribute with @Autowired.

```
@Autowired
@Qualifier("mapToTortuga")
private TreasureMap treasureMap;
```

Ensuring that Required Properties are Set

To ensure that a property is injected with a value, use the @Required annotation:

```
@Required
public void setTreasureMap(TreasureMap treasureMap) {
        this.treasureMap = treasureMap;
}
```

In this case, the "treasureMap" property must be injected or else Spring will throw a BeanInitializationException and context creation will fail.

Stereotyping Annotations

These annotations are used to stereotype classes with regard to the application tier that they belong to. Classes that are annotated with one of these annotations will automatically be registered in the Spring application context if <**context:component-scan**> is in the Spring XML configuration. It is required to activate the Spring MVC annotation capability.

In addition, if a PersistenceExceptionTranslationPostProcessor is configured in Spring, any bean annotated with @Repository will have SQLExceptions thrown from its methods translated into one of Spring's unchecked DataAccessExceptions.

Annotation	Use	Description
@Component	Type	**Generic** stereotype annotation for any Spring-managed component.
@Controller	Type	Stereotypes a component as a Spring MVC controller in the **presentation layer**.
@Repository	Type	Stereotypes a component as a repository in the **persistence layer**. Also indicates that SQLExceptions thrown from the component's methods should be translated into Spring DataAccessExceptions.
@Service	Type	Stereotypes a component as a service in the **service layer**.

Automatically Configuring Beans

In the previous section, you saw how to automatically wire a bean's properties using the @Autowired annotation. But it is possible to take autowiring to a new level by automatically registering beans in Spring. To get started with automatic registration of beans, first annotate the bean with one of the stereotype annotations, such as @Component:

```
@Component
public class Pirate {
        private TreasureMap treasureMap;
        @Autowired
        public void setTreasureMap(TreasureMap treasureMap) {
                this.treasureMap = treasureMap;
        }
}
```

You can specify a name for the bean by passing it as the value of @Component.

```
@Component("jackSparrow")
public class Pirate { … }
```

Specifying Scope For Auto-Configured Beans
By default, all beans in Spring, including auto-configured beans, are scoped as singleton. But you can specify the scope using the @Scope annotation. For example:

```
@Component
@Scope("prototype")
public class Pirate { … }
```

This specifies that the pirate bean be scoped as a prototype bean.

Creating Custom Stereotypes

Autoregistering beans are a great way to cut back on the amount of XML required to configure Spring. But it may bother you that your autoregistered classes are annotated with Spring-specific annotations. If you're looking for a more non-intrusive way to autoregister beans, you have two options:

Create your own custom stereotype annotation. Doing so is as simple as creating a custom annotation that is itself annotated with @Component:

```
@Component
public interface MyComponent {
}
```

Or add a filter to <context:component-scan> to scan for annotations that it normally would not:

```
<context:component-scan  base-package="com.habuma.pirates">
<context:include-filter  type="annotation" expression="com.habuma.MyComponent" />
<context:exclude-filter type="annotation"  expression="org.springframework.stereotype.Component"
/>
</context:component-scan>
```

In this case, the @MyComponent custom annotation has been added to the list of annotations that are scanned for, but @Component has been excluded (that is, @Componentannotated classes will no longer be autoregistered).

Regardless of which option you choose, you should be able to autoregister beans by annotating their classes with the custom annotation:

```
@MyComponent
public class Pirate { … }
```

Spring MVC Annotations

These annotations were introduced in Spring 2.5 to make it easier to create Spring MVC applications with minimal XML configuration and without extending one of the many implementations of the Controller interface.

Annotation	Use	Description
@Controller	Type	Stereotypes a component as a Spring MVC controller.
@InitBinder	Method	Annotates a method that customizes data binding.
@ModelAttribute	Method, Parameter	When applied to a method, used to preload the model with the value returned from the method. When applied to a parameter, binds a model attribute to the parameter.
@RequestMapping	Method, Type	Maps a URL pattern and/or HTTP method to a method or controller type.
@RequestParam	Parameter	Binds a request parameter to a method parameter.
@SessionAttributes	Type	Specifies that a model attribute should be stored in the session.

Setting up Spring for Annotated Controllers

Before we can use annotations on Spring MVC controllers, we'll need to add a few lines of XML to tell Spring that our controllers will be annotation-driven. First, so that we won't have to register each of our controllers individually as <bean>s, we'll need a <context:component-scan>:

```
<context:component-scan  base-package= “com.mesonsoft”/>
```

In addition to autoregistering @Component-annotated beans, <context:component-scan> also autoregisters beans that are annotated with @Controller. We'll see a few examples of @Controller-annotated classes in a moment.

But first, we'll also need to tell Spring to honor the other Spring MVC annotations. For that we'll need <**context:annotation-config**> : <context:annotation-config/>

Transaction Annotations

The @Transactional annotation is used along with the <tx:annotation-driven> element to declare transactional boundaries and rules as class and method metadata in Java.

Annotation	Use	Description
@Transactional	Method, Type	Declares transactional boundaries and rules on a bean and/or its methods.

Annotating Transactional Boundaries

To use Spring's support for annotation-declared transactions, you'll first need to add a small amount of XML to the Spring configuration:

```
<beans><tx:annotation-driven /></beans>
```

The <tx:annotation-driven> element tells Spring to keep an eye out for beans that are annotated with @Transactional. In addition, you'll also need a platform transaction manager bean declared in the Spring context. For example, if your application uses Hibernate, you'll want to include the

HibernateTransactionManager:

```
<bean id="transactionManager" class="org.springframework.orm.hibernate3.
HibernateTransactionManager"><property name="sessionFactory" ref="sessionFactory"/></bean>
```

With the basic plumbing in place, you're ready to start annotating the transactional boundaries:

```
@Transactional (propagation=Propagation.SUPPORTS, readOnly=true)
public class TreasureRepositoryImpl implements TreasureRepository {
        @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
        public void storeTreasure(Treasure treasure) {...}
}
```

At the class level, @Transactional is declaring that all methods should support transactions and be read-only. But, at the method-level, @Transactional declares that the storeTreasure() method requires a transaction and is not read-only.

Aspect Annotations

For defining aspects, Spring leverages the set of annotations provided by AspectJ.

Annotation	Use	Description
@Aspect	Type	Declares a class to be an aspect.
@After	Method	Declares a method to be called after a pointcut completes.
@AfterReturning	Method	Declares a method to be called after a pointcut returns successfully.
@AfterThrowing	Method	Declares a method to be called after a pointcut throws an exception.
@Around	Method	Declares a method that will wrap the pointcut.
@Before	Method	Declares a method to be called before proceeding to the pointcut.
@DeclareParents	Static Field	Declares that matching types should be given new parents, that is, it introduces new functionality into matching types.
@Pointcut	Method	Declares an empty method as a pointcut placeholder method.

What's important to note, however, is that while you can use AspectJ annotations to define Spring aspects, those aspects will be defined in the context of Spring AOP and will not be handled by the AspectJ runtime. This is significant because Spring AOP is limited to proxying method invocations and does not provide for the more exotic pointcuts (constructor interception, field interception, etc.) offered by AspectJ.

Annotating Aspects

To use AspectJ annotations to create Spring aspects, you'll first need to provide a bit of Spring XML plumbing:

```
<beans><aop:aspectj-autoproxy/></beans>
```

The <aop:aspectj-autoproxy> element tells Spring to watch for beans annotated with AspectJ annotations and, if it finds any, to use them to create aspects. Then you can annotate bean classes to be aspects:

```
@Aspect
public class ChantySinger {
        @Pointcut("execution(* Pirate.plunder(..))")
        public void plunderPC() {}
        @Before("plunderPC()")
        public void singYoHo() {
        ...
        }
        @AfterReturning("plunderPC()")
        public void singAPiratesLifeForMe() {
        ...
        }
}
```

This simple annotation-based aspect has a pointcut that is triggered by the execution of a plunder() method on the Pirate class. Before the Pirate.plunder() method is executed, the singYoHo() method is called. Then, after the Pirate.plunder() method returns successfully, the singAPiratesLifeForMe() method is invoked.

Note the rather odd looking plunderPC() method. It is annotated with @Pointcut to indicate that this method is a pointcut placeholder. The key thing here is that the most interesting stuff happens in the annotation itself and not in the method. In fact, pointcut placeholder methods must be empty methods and return void.

JSR-250 Annotations

In addition to Spring's own set of annotations, Spring also supports a few of the annotations defined by JSR-250, which is the basis for the annotations used in EJB 3.

Annotation	Use	Description
@PostConstruct	Method	Indicates a method to be invoked after a bean has been created and dependency injection is complete. Used to perform any initialization work necessary.
@PreDestroy	Method	Indicates a method to be invoked just before a bean is removed from the Spring context. Used to perform any cleanup work necessary.
@Resource	Method, Field	Indicates that a method or field should be injected with a named resource (by default, another bean).

Wiring Bean Properties with @Resource

Using @Resource, you can wire a bean property by name:

```
public class Pirate {
    @Resource
    private TreasureMap treasureMap;
}
```

In this case, Spring will attempt to wire the "treasureMap" property with a reference to a bean whose ID is "treasureMap". If you'd rather explicitly choose another bean to wire into the property, specify it to the name attribute:

```
public class Pirate {
   @Resource(name="mapToSkullIsland")
    private TreasureMap treasureMap;
}
```

Initialization and Destruction Methods

Using JSR-250's @PostConstruct and @PreDestroy methods, you can declare methods that hook into a bean's lifecycle. For example, consider the following methods added to the Pirate class:

```
public class Pirate {
        @PostConstruct
        public void wakeUp() { … }
        @PreDestroy
        public void goAway() { … }
}
```

As annotated, the wakeUp() method will be invoked just after Spring instantiates the bean and goAway() will be invoked just before the bean is removed from the Spring container.

Writing a Spring-Aware JUnit Test

The key to writing a Spring-aware test is to annotate the test class with @RunWith, specifying **SpringJUnit4ClassRunner** as the class runner behind the test:

```
@RunWith(SpringJUnit4ClassRunner.class)
public class PirateTest {
...
}
```

In this case, the Spring test runner will try to load a Spring application context from a file named PirateTest-context.xml. If you'd rather specify one or more XML files to load the application context from, you can do that with @ContextConfiguration:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "pirates.xml" })
public class PirateTest {
        @Autowired
        private Pirate pirate;
        @Autowired
        private TreasureMap treasureMap;
        @Test
        public void annotatedPropertyShouldBeAutowired() {
                assertNotNull(treasureMap);
                assertEquals(treasureMap, pirate.getTreasureMap());
        }
}
```

Accessing the Spring Context in a Test

If you need the Spring application context itself in a test, you can autowire it into the test the same as if it were a bean in the context:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "pirates.xml" })
public class PirateTest {
        @Autowired
        private Pirate pirate;
        @Autowired
        private ApplicationContext applicationContext;
        @Test
        public void annotatedPropertyShouldBeAutowired() {
                assertEquals(applicationContext.getBean("treasureMap"), pirate
                .getTreasureMap());
        }
}
```

Spring Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS oriented enterprise applications to ensure data integrity and consistency. The concept of transactions can be described with following four key properties described as **ACID**:

Atomicity: A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.

Consistency: This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

Isolation: There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.

Durability: Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all the four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows:

- Begin the transaction using *begin transaction* command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform *commit* otherwise *rollback* all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. The Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs requires an application server, but Spring transaction management can be implemented without a need of application server.

Local versus Global Transactions

Local transactions are specific to a **single transactional resource** like a JDBC connection, whereas global transactions can span **multiple transactional resources** like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic versus Declarative

Spring supports two types of transaction management:

Programmatic Transaction Management: This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

Declarative Transaction Management: This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Spring Transaction Abstractions

The key to the Spring transaction abstraction is defined by **PlatformTransactionManager** interface.

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction(TransactionDefinition definition);
  throws TransactionException;
  void commit(TransactionStatus status) throws TransactionException;
  void rollback(TransactionStatus status) throws TransactionException;
}
```

S.N.	Method & Description
1	**TransactionStatus getTransaction(TransactionDefinition definition)** This method returns a currently active transaction or create a new one, according to the specified propagation behavior.
2	**void commit(TransactionStatus status)** This method commits the given transaction, with regard to its status.
3	**void rollback(TransactionStatus status)** This method performs a rollback of the given transaction.

TransactionDefinition

The **TransactionDefinition** is the core interface of the transaction support in Spring and it is defined as below:

```
public interface TransactionDefinition {
  int getPropagationBehavior();
  int getIsolationLevel();
  String getName();
  int getTimeout();
  boolean isReadOnly();
}
```

S.N.	Method & Description
1	**int getPropagationBehavior()** This method returns the propagation behavior. Spring offers the entire transaction propagation options familiar from EJB CMT.
2	**int getIsolationLevel()** This method returns the degree to which this transaction is isolated from the work of other transactions.
3	**String getName()** This method returns the name of this transaction.
4	**int getTimeout()** This method returns the time in seconds in which the transaction must complete.
5	**boolean isReadOnly()** This method returns whether the transaction is read-only.

Isolation Level

S.N.	Isolation & Description
1	**TransactionDefinition.ISOLATION_DEFAULT** This is the default isolation level.
2	**TransactionDefinition.ISOLATION_READ_UNCOMMITTED** Indicates that dirty reads, non-repeatable reads and phantom reads can occur.
3	**TransactionDefinition.ISOLATION_READ_COMMITTED** Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
4	**TransactionDefinition.ISOLATION_REPEATABLE_READ** Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
5	**TransactionDefinition.ISOLATION_SERIALIZABLE** Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

Propagation Behavior

S.N.	Propagation & Description
1	**TransactionDefinition.PROPAGATION_MANDATORY** Support a current transaction; throw an exception if no current transaction exists.
2	**TransactionDefinition.PROPAGATION_NESTED** Execute within a nested transaction if a current transaction exists.
3	**TransactionDefinition.PROPAGATION_NEVER** Do not support a current transaction; throw an exception if a current transaction exists.
4	**TransactionDefinition.PROPAGATION_NOT_SUPPORTED** Do not support a current transaction; rather always execute non-transactionally.
5	**TransactionDefinition.PROPAGATION_REQUIRED** Support a current transaction; create a new one if none exists.
6	**TransactionDefinition.PROPAGATION_REQUIRES_NEW** Create a new transaction, suspending the current transaction if one exists.
7	**TransactionDefinition.PROPAGATION_SUPPORTS** Support a current transaction; execute non-transactionally if none exists.
8	**TransactionDefinition.TIMEOUT_DEFAULT** Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

The **TransactionStatus** interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
  boolean isNewTransaction();
  boolean hasSavepoint();
  void setRollbackOnly();
  boolean isRollbackOnly();
  boolean isCompleted();
}
```

S.N.	Method & Description
1	**boolean hasSavepoint()** This method returns whether this transaction internally carries a savepoint.
2	**boolean isCompleted()** This method returns whether this transaction is completed (either committed or rolled back).
3	**boolean isNewTransaction()** This method returns true in case the present transaction is new.
4	**boolean isRollbackOnly()** This method returns whether the transaction has been marked as rollback-only.
5	**void setRollbackOnly()** This method sets the transaction rollback-only.

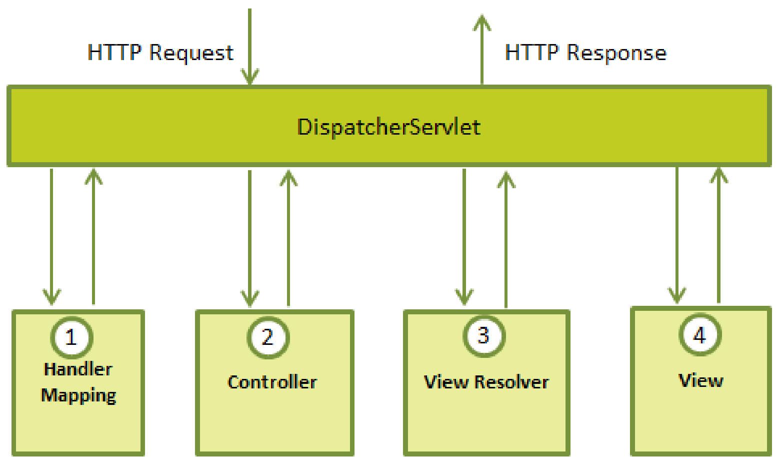
Spring MVC Framework

The Spring web MVC framework provides **model-view-controller architecture** and ready components that can be used to develop **flexible and loosely coupled web applications**. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The **Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram:



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*:

1. After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
2. The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
3. The *DispatcherServlet* will take help from *ViewResolver* to pick up the defined view for the request.
4. Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.
5. All the above mentioned components ie. HandlerMapping, Controller and ViewResolver are parts of **WebApplicationContext** which is an extension of the plain *ApplicationContext* with some extra features necessary for web applications.

Spring MVC Configuration

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file. The **web.xml** file will be kept *WebContent/WEB-INF* directory of your web application. OK, upon initialization of **HelloWeb** *DispatcherServlet*, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's *WebContent/WEB-INF* directory. In this case our file will be **HelloWeb-servlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by the which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.
If you do not want to go with default filename as *[servlet-name]-servlet.xml* and default location as*WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener*ContextLoaderListener* in your web.xml file as follows:

```
<web-app>
  <servlet>
    <servlet-name>HelloWeb</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
```

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory:

```
<context:component-scan  base-package="com.mesonsoft" />

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Following are the important points about **HelloWeb-servlet.xml** file:

- The *[servlet-name]-servlet.xml* file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
- The <***context:component-scan***...> tag will be used to **activate Spring MVC annotation scanning capability** which allows to make use of annotations like **@Controller** and **@RequestMapping** etc.
- The **InternalResourceViewResolver** will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at ***/WEB-INF/jsp/****hello****.jsp***.

Defining a Controller

DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The**@Controller** annotation indicates that a particular class serves the role of a controller. The**@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
@RequestMapping("/hello")
public class HelloController{
   @RequestMapping(method = RequestMethod.GET)
  public String printHello(ModelMap model) {
    model.addAttribute("message", "Hello Spring MVC Framework!");
    return "hello";
  }
}
```

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of**@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation **@RequestMapping(method=RequestMethod.GET)** is used to declare the *printHello()* method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.
You can write above controller in another form where you can add additional attributes in*@RequestMapping* as follows:

```
@Controller
public class HelloController{

  @RequestMapping(value = "/hello", method = RequestMethod.GET)
  public String printHello(ModelMap model) {
    model.addAttribute("message", "Hello Spring MVC Framework!");
    return "hello";
  }
}
```

Spring MVC Handler Interceptor

Spring MVC allows you to intercept the web request and response through handler interceptors. The handler interceptor has to implement the HandlerInterceptor interface, which contains three methods:

1. **preHandle()** – Called before the handler execution. It returns a Boolean value. True means continue the handler execution chain; otherwise, stop the execution chain and return it.
2. **postHandle()** – Called after the handler execution. It allows you to manipulate the ModelAndView object before render it to the view page.
3. **afterCompletion()** – Called after the user request has been processed. Seldom use, we can’t find any use case.

It’s recommended to extend the HandlerInterceptorAdapter class for the convenient default implementations.

Chapter 24 – Hibernate

ORM (Object/Relational Mapping)

ORM stands for Object Relational Mapping. ORM is the automated persistence of objects in a Java application to the tables in a relational database.

An ORM solution consists of the following four pieces:

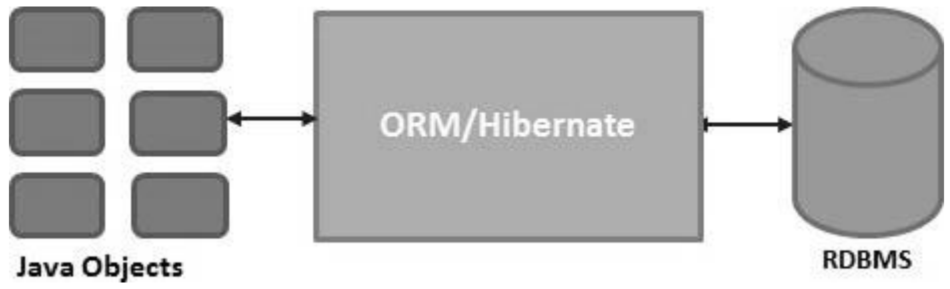
- API for performing basic CRUD operations
- API to express queries referring to classes
- Facilities to specify metadata
- Optimization facilities such as dirty checking, lazy associations fetching and so on

The ORM levels are:

- Pure relational (stored procedure)
- Light object mapping (JDBC)
- Medium object mapping
- Full object Mapping (composition, inheritance, polymorphism, persistence by reachability)

Hibernate Framework

Hibernate is a pure Java object relational mapping (ORM) and persistence framework that allows you to map plain old java objects to relational database tables using (XML) configuration files.



The main advantage of ORM like hibernate is that it shields developers from messy SQL. Apart from this, ORM provides following benefits:

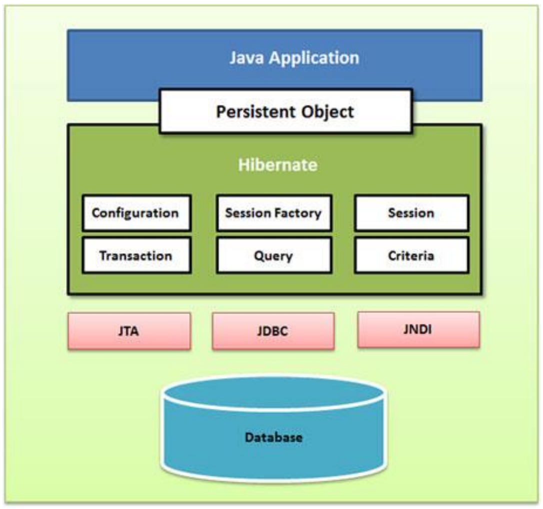
- **Improved productivity**
  - High-level object-oriented API
  - Less Java code to write
  - No SQL to write
- **Improved performance**
  - Sophisticated caching
  - Lazy loading
  - Eager loading
- **Improved maintainability**
  - A lot less code to write and maintain
- **Improved portability**
  - ORM framework generates database-specific SQL for you

Hibernate simplifies:

- Saving and retrieving your domain objects
- Making database column and table name changes
- Centralizing pre save and post retrieve logic
- Complex joins for retrieving related items
- Schema creation from object model

Core Interfaces of Hibernate Framework:

1. Session Interface
2. SessionFactory Interface
3. Configuration Interface
4. Transaction Interface
5. Query and Criteria Interface



General Flow:

The general flow of Hibernate communication with RDBMS is:

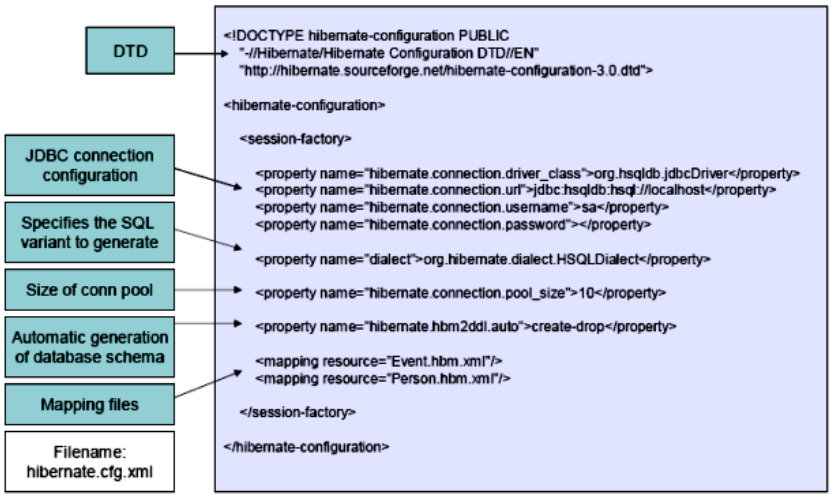
1. Load the Hibernate configuration file and create configuration object. It will automatically load all hbm mapping files
2. Create session factory from configuration object
3. Get one session from this session factory
4. Begin the transaction, and create HQL query
5. Execute the query to get a list containing Java objects
6. Commit or Rollback the transaction based on the query execution outcome

Configuration Interface

The most common methods of Hibernate configuration are:

- Programmatic configuration
- Declarative configuration (hibernate.cfg.xml)

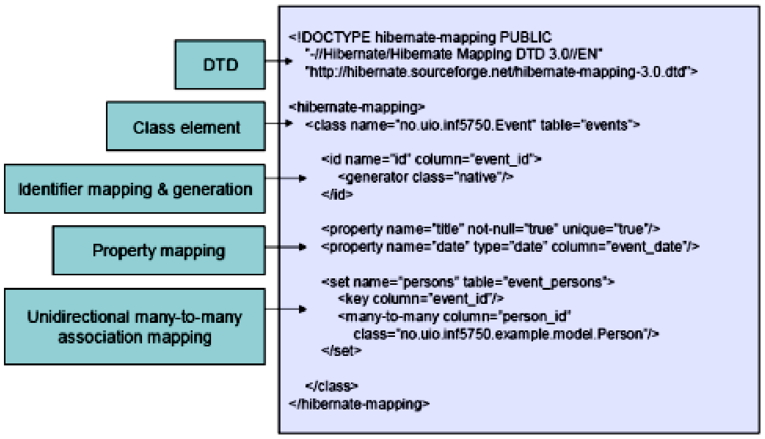
Following are the important tags of the Hibernate configuration file (hibernate.cfg.xml):



DTD
JDBC connection configuration
Specifies the SQL variant to generate
Size of conn pool
Automatic generation of database schema
Mapping files
Filename: hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

  <session-factory>

    <property name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="hibernate.connection.url">jdbc:hsqldb:hsql://localhost</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>

    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <property name="hibernate.connection.pool_size">10</property>

    <property name="hibernate.hbm2ddl.auto">create-drop</property>

    <mapping resource="Event.hbm.xml"/>
    <mapping resource="Person.hbm.xml"/>

  </session-factory>

</hibernate-configuration>
```

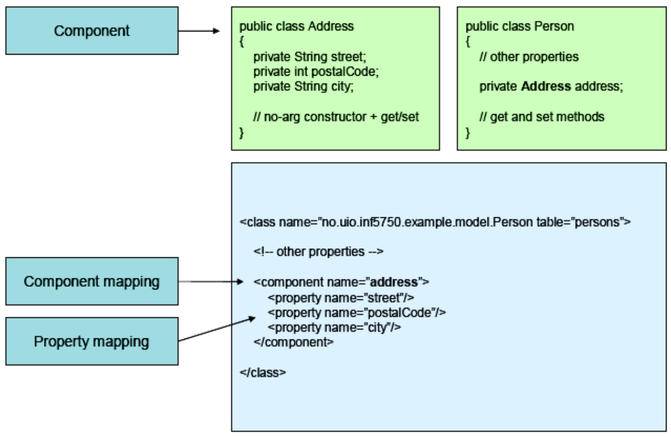
Hibernate mapping file tells Hibernate which tables and columns to use to load and store objects. Some of the important tags of a typical mapping file (person.hbm.xml) are given below:



DTD
Class element
Identifier mapping & generation
Property mapping
Unidirectional many-to-many association mapping

```
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="no.uio.inf5750.Event" table="events">

    <id name="id" column="event_id">
      <generator class="native"/>
    </id>

    <property name="title" not-null="true" unique="true"/>
    <property name="date" type="date" column="event_date"/>

    <set name="persons" table="event_persons">
      <key column="event_id"/>
      <many-to-many column="person_id"
        class="no.uio.inf5750.example.model.Person"/>
    </set>

  </class>
</hibernate-mapping>
```

Component Mapping in Hibernate



```
public class Address
{
    private String street;
    private int postalCode;
    private String city;

    // no-arg constructor + get/set
}
```

```
public class Person
{
    // other properties

    private Address address;

    // get and set methods
}
```

```
<class name="no.uio.inf5750.example.model.Person table="persons">

    <!-- other properties -->

    <component name="address">
        <property name="street"/>
        <property name="postalCode"/>
        <property name="city"/>
    </component>

</class>
```

Derived Properties

The properties that are not mapped to a column, but calculated at runtime by evaluation of an expression are called derived properties. The expression can be defined using the **formula** attribute of the element.

```
<property name="_LastRevisionDate"
    formula="(select MAX(rev.start_date) from trevision rev where rev.id_detectable = id_detectable and rev.status != 'DRAFT')"
    type="date" />
```

Hibernate DataSource Connection Properties:

Following is the list of important properties you would require to configure for the databases in a **standalone** situation:

S.N.	Properties and Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.
4	**hibernate.connection.username** The database username.
5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.
7	**hibernate.connection.autocommit** Allows autocommit mode to be used for the JDBC connection.

If you are using a database along with an **application server and JNDI** then you would have to configure the following properties:

S.N.	Properties and Description
1	**hibernate.connection.datasource** The JNDI name defined in the application server context you are using for the application.
2	**hibernate.jndi.class** The InitialContext class for JNDI.
3	**hibernate.jndi.<JNDIpropertyname>** Passes any JNDI property you like to the JNDI *InitialContext*.
4	**hibernate.jndi.url** Provides the URL for JNDI.
5	**hibernate.connection.username** The database username.
6	**hibernate.connection.password** The database password.

Hibernate Mapping Configuration

```
<hibernate-mapping
        schema="databaseSchemaName"                          1
        catalog="databaseCatalogName"                        2
        default-cascade="none|cascade_style"                 3
        default-access="field|property|ClassName"            4
        default-lazy="true|false"                            5
        auto-import="true|false"                             6
        package="package.name"                               7
 />
```

1. schema (optional): the name of a database schema.
2. catalog (optional): the name of a database catalog.
3. default-cascade (optional - defaults to none): a default cascade style.
4. default-access (optional - defaults to property): the strategy Hibernate should use for accessing all properties. It can be a custom implementation of PropertyAccessor. By mapping the property with default-access="field" in Hibernate metadata. These forces hibernate to bypass the setter method and access the instance variable directly while initializing a newly loaded object.
5. default-lazy (optional - defaults to true): the default value for unspecified lazy attributes of class and collection mappings.
6. auto-import (optional - defaults to true): specifies whether we can use unqualified class names of classes in this mapping in the query language.
7. package (optional): specifies a package prefix to use for unqualified class names in the mapping document.

If you have two persistent classes with the same unqualified name, you should set auto-import="false". An exception will result if you attempt to assign two classes to the same "imported" name.

The hibernate-mapping element allows you to nest several persistent <class> mappings, as shown above. It is, however, good practice (and expected by some tools) to map only a single persistent class, or a single class hierarchy, in one mapping file and name it after the persistent superclass. For example, Cat.hbm.xml, Dog.hbm.xml, or if using inheritance, Animal.hbm.xml.

Class Configuration

You can declare a persistent class using the class element. For example:

```
<class
        name="ClassName"                                  ①
        table="tableName"                                 ②
        discriminator-value="className"                   ③
        mutable="true|false"                              ④
        schema="databaseSchemaName"                       ⑤
        catalog="databaseCatalogName"                     ⑥
        proxy="ProxyInterface"                            ⑦
        dynamic-update="true|false"                       ⑧
        dynamic-insert="true|false"                       ⑨
        select-before-update="true|false"                 ⑩
        polymorphism="implicit|explicit"                  ⑪
        where="arbitrary sql where condition"             ⑫
        persister="PersisterClass"                        ⑬
        batch-size="1"                                    ⑭
        optimistic-lock="none|version|dirty|all"          ⑮
        lazy="true|false"                                 (16)
        entity-name="ClassName as Default"                (17)
        check="arbitrary sql check condition"             (18)
        rowid="rowid"                                     (19)
        subselect="SQL expression"                        (20)
        abstract="true|false"                             (21)
        node="elementName"                                (22)
/>
```

① name (optional): the fully qualified Java class name of the persistent class or interface. If this attribute is missing, it is assumed that the mapping is for a non-POJO entity.

② table (optional - defaults to the unqualified class name): the name of its database table.

③ discriminator-value (optional - defaults to the class name): a value that distinguishes individual subclasses that is used for polymorphic behavior. Acceptable values include null and not null.

④ mutable (optional - defaults to true): specifies that instances of the class are (not) mutable. Immutable classes may not be updated or deleted by the application.

⑤ schema (optional): overrides the schema name specified by the root <hibernate-mapping>element.

(6) catalog (optional): overrides the catalog name specified by the root <hibernate-mapping>element.

(7) proxy (optional): specifies an interface to use for lazy initializing proxies. You can specify the name of the class itself.

(8) dynamic-update (optional - defaults to false): specifies that UPDATE SQL should be generated at runtime and can contain only those columns whose values have changed.

(9) dynamic-insert (optional - defaults to false): specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.

(10) select-before-update (optional - defaults to false): specifies that Hibernate should *never* perform an SQL UPDATE unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using update(), will Hibernate perform an extra SQL SELECT to determine if an UPDATE is actually required.

(11) polymorphism (optional - defaults to implicit): determines whether implicit or explicit query polymorphism is used. Use polymorphism="explicit" in the class mapping. This will cause queries to return only instances of the named class and not its subclasses.

(12) where (optional): specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class.

(13) persister (optional): specifies a custom ClassPersister.

(14) batch-size (optional - defaults to 1): specifies a "batch size" for fetching instances of this class by identifier.

(15) optimistic-lock (optional - defaults to version): determines the optimistic locking strategy.

(16) lazy (optional): lazy fetching can be disabled by setting lazy="false".

(17) entity-name (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity.

(18) check (optional): an SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

(19) rowid (optional): Hibernate can use ROWIDs on databases. On Oracle, for example, Hibernate can use the rowid extra column for fast updates once this option has been set to rowid. A ROWID is an implementation detail and represents the physical location of a stored tuple.

(20) subselect (optional): maps an immutable and read-only entity to a database subselect. This is useful if you want to have a view instead of a base table. See below for more information.

(21) abstract (optional): is used to mark abstract superclasses in <union-subclass> hierarchies.

Cascade and Inverse option in One to Many mapping

Cascade enables operations to cascade to child entities. The vaild values are all, none, save-update, delete and all-delete-orphan. The default value is none.

Inverse indicates which end of a bi-directional association relationship should be ignored. It determines whether ask a parent for their children or a child for their parents when persisting a parent who has a collection of children. The valid values are true and false. The default value is false.

Dynamic Insert versus Dynamic Update

- dynamic-update (defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed.
- dynamic-insert (defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only those columns whose values are not null.

Hibernate Proxy

The proxy attribute enables lazy initialization of persistent instances of the class. Hibernate will initially return CGLIB proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked.

Callback Interface

Callback interfaces allow the application to receive a notification when something interesting happens to an object. It happens when an object is loaded, saved, or deleted. Hibernate applications don't need to implement these callbacks, but they're useful for implementing certain kinds of generic functionality.

HibernateCallback is the callback interface for Hibernate application. To be used with HibernateTemplate's execution methods, often as anonymous classes within a method implementation. The typical implementation will call Session.load/find/update to perform some operations on persistent objects. It can also perform direct JDBC operations via Hibernate's Session.connection(), operating on a JDBC Connection. The only method that needs to be overridden is **doInHibernate(session)** that gets called by HibernateTemplate's execute method with an active Hibernate session.

Session Factory

The application obtains Session instances from a SessionFactory. There is typically a single SessionFactory for the whole application created during application initialization. The SessionFactory caches generated SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work.

Session Factory is a Hibernate's concept of a single data store and is **thread safe** so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database. It usually only built once at startup. It should be wrapped in some kind of singleton.

XML Configuration

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

Annotated Configuration

```
SessionFactory sessionFactory = new
AnnotationConfiguration().configure().addPackage("package.name").addAnnotatedClass(ClassName.class).buildSessionFactory();
```

Session

The Session interface is the primary interface used by Hibernate applications. It is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It allows you to create query objects to retrieve persistent objects.

```
Session session = sessionFactory.openSession();
```

Session interface role:

- Wraps a JDBC connection
- Factory for Transaction
- Holds a mandatory first-level cache of persistent objects, used when navigating the object graph or looking up objects by identifier

Session is a light weight and **non-thread safe** object that represents a single unit of work with the database. Sessions are created by a Session Factory and they are closed when all work is complete. Session is the primary interface of the persistence service. A session obtains a database connection **lazily**. To avoid creating too many sessions, ThreadLocal class can be used.

```
ThreadLocal local = new ThreadLocal().get();
Session session = sessionFactory.openSession();
local.set(session);
```

Ending a session usually involves four distinct phases:

1. Flush the session
2. Commit the transaction
3. Handle exceptions and Rollback the transaction
4. Finally, close the session

Session Interface Methods:

There are number of methods provided by the **Session** interface but I'm going to list down a few important methods only.

#	Session Methods and Description
1	**boolean isOpen()** Check if the session is still open.
2	**boolean isConnected()** Check if the session is currently connected.
3	**boolean isDirty()** Does this session contain any changes which must be synchronized with the database?
4	**void refresh(Object object)** Re-read the state of the given instance from the underlying database.
5	**void flush()** Flushing is the process of synchronizing the underlying persistent store with persistable state held in memory.
6	**void clear()** Completely clear the session.
7	**Connection close()** End the session by releasing the JDBC connection and cleaning up.
8	**SessionFactory getSessionFactory()** Get the session factory which created this session.

9	**Serializable getIdentifier(Object object)** Return the identifier value of the given entity as associated with this session.
10	**Transaction beginTransaction()** Begin a unit of work and return the associated Transaction object.
11	**Transaction getTransaction()** Get the Transaction instance associated with this session.
12	**Criteria createCriteria(Class persistentClass)** Create a new Criteria instance, for the given entity class, or a superclass of an entity class.
13	**Criteria createCriteria(String entityName)** Create a new Criteria instance, for the given entity name.
14	**Query createFilter(Object collection, String hqlQueryString)** Create a new instance of Query for the given collection and filter string.
15	**Query createQuery(String hqlQueryString)** Create a new instance of Query for the given HQL query string.
16	**SQLQuery createSQLQuery(String queryString)** Create a new instance of SQLQuery for the given SQL query string.
17	**void cancelQuery()** Cancel the execution of the current query.
18	**Object load(String entityName, Serializable id, LockOptions lockOptions)** Return the persistent instance of the given entity class with the given identifier, obtaining the specified lock mode, assuming the instance exists.
19	**Session get(String entityName, Serializable id)** Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.
20	**Serializable save(Object object)** Persist the given transient instance, first assigning a generated identifier.
21	**void saveOrUpdate(Object object)** Either save(Object) or update(Object) the given instance.
22	**void update(Object object)** Update the persistent instance with the identifier of the given detached instance.
23	**void update(String entityName, Object object)** Update the persistent instance with the identifier of the given detached instance.
24	**Object merge(String entityName, Object object)** Copy the state of the given object onto the persistent object with the same identifier in session.
25	**void delete(Object object)** Remove a persistent instance from the datastore.
26	**void delete(String entityName, Object object)** Remove a persistent instance from the datastore.

Differences between load and get methods

Load	Get
Only use the load() method if you are sure that the object exists.	If you are not sure that the object exists, then use one of the get() methods.
load() method will throw **ObjectNotFoundException** if the unique id is not found in the database.	get() method will return null if the unique id is not found in the database.
load() just returns a proxy by default and database won't be hit until the proxy is first invoked.	get() will hit the database immediately.

Differences between update and merge methods

Update	Merge
Use update method, if you are sure that the session does not contain an already persistent instance with the same identifier.	Use merge method, if you want to merge your modifications at any time without consideration of the state of the session.
It will throw **NonUniqueObjectException** if a persistent object with the same identifier is already in the session	It will act like update but if a persistent object with the same identifier is already in the session it will copy the state of the given object onto the persistent object with the same identifier in session.

Hibernate Annotations:

@Entity Annotation:

The EJB 3 standard annotations are contained in the **javax.persistence** package, so we import this package as the first step. Secondly, we used the **@Entity** annotation to the Employee class which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation:

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now we are using just table name which is EMPLOYEE.

@Id and @GeneratedValue Annotations:

Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.
By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the **@GeneratedValue** annotation which takes two parameters **strategy** and **generator**. We can use the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation:

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

- **name** attribute permits the name of the column to be explicitly specified.
- **length** attribute permits the size of the column used to map a value particularly for a String value.
- **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.
- **unique** attribute permits the column to be marked as containing only unique values.

@OneToOne, @OneToMany, @ManyToOne and @ManyToMany Annotations

The parameters used by these annotations are fetch (FetchType.LAZY or FetchType.EAGER), cascade (CascadeType.ALL or CascadeType.NONE), inverse (true or false) and orphanRemoval (true or false).

Every relationship that finishes with **@Many** (@OneToMany and @ManyToMany) will be **lazy loaded by default**. Every relationship that finishes with **@One** (@ManyToOne and @OneToOne) will be **eagerly loaded by default**. If you want to set a basic field (e.g. String name) with lazy loading just does: @Basic (fetch=FetchType.LAZY)

Every basic field (e.g. String, int, double) that we can find inside a class will be eagerly loaded if the developer does not set it as lazy.

Entity Class:

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
  @Id @GeneratedValue
  @Column(name = "id")
  private int id;

  @Column(name = "emp_name")
  private String employeeName;

  @Basic(fetch = FetchType.LAZY)
  @Column(name = "salary")
  private int salary;
}
```

Application Class:

```
private static SessionFactory factory;
    try{
      factory = new AnnotationConfiguration(). configure().
             addPackage("com.xyz"). //add package if used.
             addAnnotatedClass(Employee.class). buildSessionFactory();
    }catch (Throwable ex) {
      System.err.println("Failed to create sessionFactory object." + ex);
      throw new ExceptionInInitializerError(ex);
}
    /* Method to CREATE an employee in the database */
```

```
Session session = factory.openSession();
Transaction tx = null;
Integer employeeID = null;
try{
  tx = session.beginTransaction();
  Employee employee = new Employee();
  employee.setFirstName(fname);
  employeeID = (Integer) session.save(employee);
  session.flush();
  tx.commit();
}catch (HibernateException e) {
  if (tx!=null) tx.rollback();
  e.printStackTrace();
}finally {
  session.close();
}
```

```
/* Method to READ all the employees */
Session session = factory.openSession();
Transaction tx = null;
try{
  tx = session.beginTransaction();
  session.setFlushMode(FlushMode.COMMIT);
  List employees = session.createQuery("FROM Employee").list();
  for (Iterator iterator =  employees.iterator(); iterator.hasNext();){
    Employee employee = (Employee) iterator.next();
  }
  tx.commit(); //flush occurs here
}catch (HibernateException e) {
  if (tx!=null) tx.rollback();
  e.printStackTrace();
}finally {
  session.close();
}
```

```
/* Method to UPDATE salary for an employee */
Session session = factory.openSession();
Transaction tx = null;
try{
  tx = session.beginTransaction();
  Employee employee =  (Employee)session.get(Employee.class, EmployeeID);
  employee.setSalary( salary );
  session.update(employee);
  session.flush();
  tx.commit();
}catch (HibernateException e) {
  if (tx!=null) tx.rollback();
  e.printStackTrace();
}finally {
  session.close();
}
/* Method to DELETE an employee from the records */
Session session = factory.openSession();
Transaction tx = null;
```

```
   try{
     tx = session.beginTransaction();
     session.setFlushMode(FlushMode.COMMIT);
     Employee employee =  (Employee)session.get(Employee.class, EmployeeID);
     session.delete(employee);
     tx.commit(); //flush occurs here
   }catch (HibernateException e) {
     if (tx!=null) tx.rollback();
     e.printStackTrace();
   }finally {
     session.close();
   }
```

Query Interface

HQL (Hibernate Query Language)

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries which in turn perform actions on database tables.

Although you can use SQL statements directly with Hibernate using Native SQL but I would recommend using HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT, FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL.

FROM Clause

You will use **FROM** clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

If you need to fully qualify a class name in HQL, just specifies the package and class name as follows:

```
String hql = "FROM com.mesonsoft.criteria.Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

AS Clause

In HQL, the **AS** clause can be used to assign aliases to the classes, especially when you have long queries. For instance, our previous simple example would be the following:

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

The **AS** keyword is optional and you can also specify the alias directly after the class name, as follows:

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

SELECT Clause

The **SELECT** clause provides more control over the result set than the FROM clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first_name field of the Employee object:

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

WHERE Clause

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause:

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

ORDER BY Clause

To sort your HQL query's results, you will need to use the **ORDER BY** clause. You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC). Following is the simple syntax of using ORDER BY clause:

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows:

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

GROUP BY Clause

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, uses the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E  GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

Using Named Parameters

Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to **defend against SQL injection attacks.** Following is the simple syntax of using named parameters:

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

Bulk Updates

Bulk updates are new to HQL with Hibernate 3, and deletes work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a new method called executeUpdate() for executing HQL UPDATE, INSERT or DELETE statements.

UPDATE Clause

The **UPDATE** clause can be used to update one or more properties of one or more objects. Following is the simple syntax of using UPDATE clause:

```
String hql = "UPDATE Employee set salary = :salary WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE Clause

The **DELETE** clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause:

```
String hql = "DELETE FROM Employee WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT Clause

HQL supports **INSERT INTO** clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause:

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
          "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

S.N.	Functions	Description
1	avg(property name)	The average of a property's value
2	count(property name or *)	The number of times a property occurs in the results
3	max(property name)	The maximum value of the property values
4	min(property name)	The minimum value of the property values
5	sum(property name)	The sum total of the property values

DISTINCT

The **distinct** keyword only counts the unique values in the row set. The following query will return only unique count:

```
String hql = "SELECT count(DISTINCT E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

Pagination using Query

There are two methods of the Query interface for pagination.

S.N.	Method & Description
1	**Query setFirstResult(int startPosition)** This method takes an integer that represents the first row in your result set, starting with row 0.
2	**Query setMaxResults(int maxCount)** This method tells Hibernate to retrieve a fixed number **maxCount** of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example which you can extend to fetch 10 rows at a time:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

Ways to express joins in HQL

HQL provides four ways of expressing equi (inner and outer) joins:-

- An **implicit** association join
- An ordinary JOIN in the FROM clause (RIGHT JOIN)
- A FETCH JOIN in the FROM clause. (LEFT JOIN FETCH)
- A **theta-style** join in the WHERE clause. (to find all children of an entity)

  Select c from Entity e, Child c **where e = c.parent**

  Such joins are called theta style joins - a cartesian product + a where condition to restrict the result.

Criteria Interface

'Criteria' is a simplified API for retrieving entities by composing Criterion objects and build up a criteria query object programmatically where you can apply **filtration rules and logical conditions**. This is a very convenient approach for functionality like "search" screens where there are a variable number of conditions to be placed upon the result set.

The Hibernate **Session** interface provides **createCriteria()** method which can be used to create a **Criteria** object that returns instances of the persistence object's class when your application executes a criteria query.

Following is the simplest example of a criteria query is one which will simply return every object that corresponds to the Employee class.

```
Criteria cr = session.createCriteria(Employee.class);
List results = cr.list();
```

Restrictions with Criteria:

You can use **add()** method available for **Criteria** object to add restriction for a criteria query. Following is the example to add a restriction to return the records with salary is equal to 2000:

```
Criteria cr = session.createCriteria(Employee.class);
cr.add(Restrictions.eq("salary", 2000));
List results = cr.list();
```

Following are the few more examples covering different scenarios and can be used as per requirement:

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary equals to 2000
cr.add(Restrictions.eq("salary", 2000));

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

You can create AND or OR conditions using LogicalExpression restrictions as follows:

```
Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt("salary", 2000);
Criterion name = Restrictions.ilike("firstNname","zara%");

// To get records matching with OR condistions
LogicalExpression orExp = Restrictions.or(salary, name);
cr.add( orExp );

// To get records matching with AND condistions
LogicalExpression andExp = Restrictions.and(salary, name);
cr.add( andExp );

List results = cr.list();
```

Though, all the above conditions can be used directly with HQL as explained before.

Pagination using Criteria:

There are two methods of the Criteria interface for pagination.

S.N.	Method & Description
1	**public Criteria setFirstResult(int startPosition)** This method takes an integer that represents the first row in your result set, starting with row 0.
2	**public Criteria setMaxResults(int maxCount)** This method tells Hibernate to retrieve a fixed number **maxResults** of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example which you can extend to fetch 10 rows at a time:

```
Criteria cr = session.createCriteria(Employee.class);
cr.setFirstResult(1);
cr.setMaxResults(10);
List results = cr.list();
```

Sorting the Results:

The Criteria API provides the **org.hibernate.criterion.Order** class to sort your result set in either ascending or descending order, according to one of your object's properties. This example demonstrates how you would use the Order class to sort the result set:

```
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order
```

```
crit.addOrder(Order.desc("salary"));

// To sort records in ascending order
crit.addOrder(Order.asc("salary"));

List results = cr.list();
```

Projections & Aggregations:

The Criteria API provides the **org.hibernate.criterion.Projections** class which can be used to get average, maximum or minimum of the property values. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining **Projection** instances.

Following are the few examples covering different scenarios and can be used as per requirement:

```
Criteria cr = session.createCriteria(Employee.class);

// To get total row count.
cr.setProjection(Projections.rowCount());

// To get average of a property.
cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.
cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.
cr.setProjection(Projections.max("salary"));

// To get minimum of a property.
cr.setProjection(Projections.min("salary"));

// To get sum of a property.
cr.setProjection(Projections.sum("salary"));
```

Hibernate Native SQL

You can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

Your application will create a native SQL query from the session with the **createSQLQuery()** method on the Session interface.:

```
public SQLQuery createSQLQuery(String sqlStatement) throws HibernateException
```

After you pass a string containing the SQL query to the createSQLQuery() method, you can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using addEntity(), addJoin(), and addScalar() methods respectively.

Scalar queries:

The most basic SQL query is to get a list of scalars (values) from one or more tables. Following is the syntax for using native SQL for scalar values:

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List results = query.list();
```

Entity queries:

The above queries were all about returning scalar values, basically returning the "raw" values from the resultset. The following is the syntax to get entity objects as a whole from a native sql query via addEntity().

```
String sql = "SELECT * FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List results = query.list();
```

Named SQL queries:

The following is the syntax to get entity objects from a native sql query via addEntity() and using named SQL query.

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list();
```

Named SQL queries are defined in the mapping XML document and called wherever required.
Example:

```
<sql-query name = "empdetails">
  <return alias="emp" class="com.test.Employee"/>
    SELECT emp.EMP_ID AS {emp.empid},
           emp.EMP_ADDRESS AS {emp.address},
           emp.EMP_NAME AS {emp.name}
    FROM Employee EMP WHERE emp.NAME LIKE :name
</sql-query>
```

Invoke Named Query :

```
List people = session.getNamedQuery("empdetails")
                                   .setString("TomBrady", name)
                                   .setMaxResults(50)
                                   .list();
```

Invoke Stored Procedure

```
<sql-query name="selectAllEmployees_SP" callable="true">
 <return alias="emp" class="employee">
   <return-property name="empid" column="EMP_ID"/>
   <return-property name="name" column="EMP_NAME"/>
   <return-property name="address" column="EMP_ADDRESS"/>
    { ? = call selectAllEmployees() }
 </return>
</sql-query>
```

Hibernate Template

org.springframework.orm.hibernate.HibernateTemplate is a helper class which provides different methods for querying and retrieving data from the database. It also converts the checked HibernateExceptions into unchecked DataAccessExceptions (Runtime Exceptions).

The benefits of HibernateTemplate are:

- HibernateTemplate, a Spring Template class simplifies interactions with Hibernate session.
- Common functions are simplified to a single method call.
- Sessions are automatically closed.
- Exceptions are automatically caught and converted into runtime exceptions.

Tips:

- Using Hibernate SQL Dialects, we can switch databases. Hibernate will generate appropriate hql queries based on the dialect defined.
- In order to print the Hibernate generated SQL statements on Console, the configuration file has to be updated with the following

```
<property name="show_sql">true</property>
```

Collection Types in Hibernate

Collection Types are:

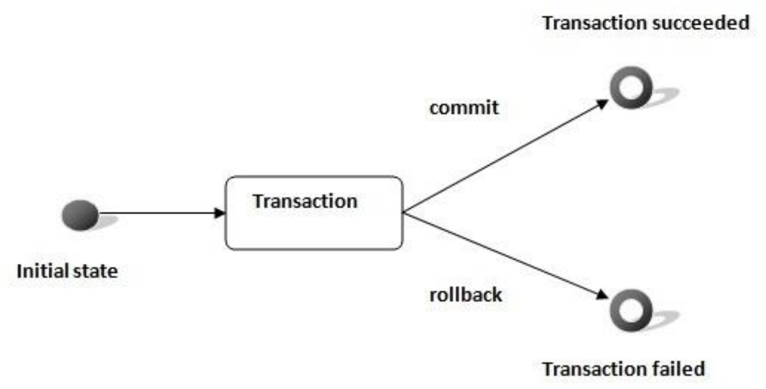
- Bag
- Set
- List
- Array
- Map

Sorted Collection versus Ordered Collection

Sorted Collection	**Ordered Collection**
A sorted collection is sorting a collection by utilizing the sorting features provided by the Java collections framework. The sorting occurs in the memory of JVM that runs Hibernate, after the data being read from database using java comparator.	Order collection is sorting a collection by specifying the ORDER BY clause for sorting this collection when retrieval.
If your collection is not large, it will be more efficient way to sort it.	If your collection is very large, it will be more efficient way to sort it.

Hibernate Transaction Management

A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



Transaction Interface

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA, JDBC, so on).

A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

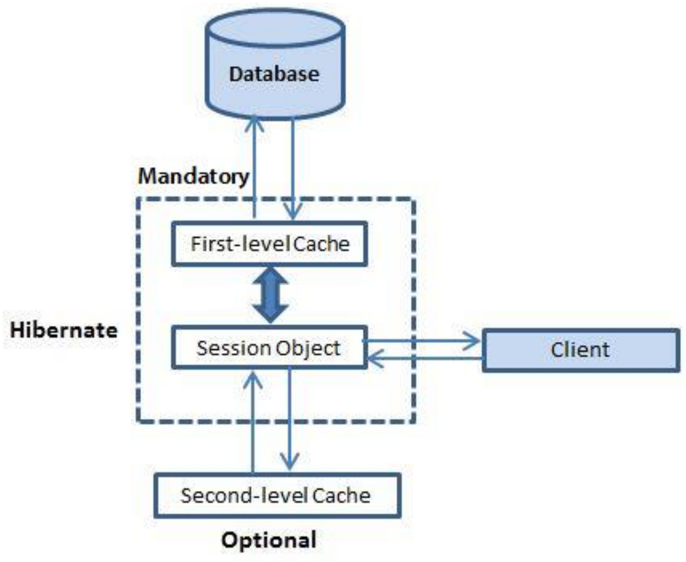
The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in **FlushMode.NEVER**.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.
6. **boolean isAlive()** checks if the transaction is still alive.
7. **boolean wasCommited()** checks if the transaction is commited successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

Caching

Caching is all about application performance optimization and it sits between your application and the database to avoid the number of database hits as many as possible to give a better performance for performance critical applications.

Caching is important to Hibernate as well which utilizes multilevel caching schemes as explained below:



First-level cache:

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Second-level cache:

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-level cache can be configured on a **per-class** and **per-collection basis** and mainly responsible for **caching objects across sessions**.

Any third-party cache can be used with Hibernate. An **org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

Query-level cache:

Hibernate also implements a cache for query result sets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the **cached query results and the timestamps** when a table was last updated. This is only useful for queries that are **run frequently with the same parameters**.

To use the query cache, you must first activate it using the **hibernate.cache.use_query_cache = "true"** property in the configuration file. By setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Next, to use the query cache, you use the setCacheable(Boolean) method of the Query class. For example:

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
List users = query.list();
SessionFactory.closeSession();
```

Hibernate also supports very fine-grained cache support through the concept of a cache region. A cache region is part of the cache that's given a name.

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
query.setCacheRegion("employee");
List users = query.list();
SessionFactory.closeSession();
```

This code uses the method to tell Hibernate to store and look for the query in the employee area of the cache.

The Second Level Cache:

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.

Concurrency strategies:

A concurrency strategy is a mediator which responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

- **Transactional:** Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- **Read-write:** Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- **Non-strict-read-write:** This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.
- **Read-only:** A concurrency strategy suitable for data which never changes. Use it for reference data only.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <cache usage="read-write"/>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

The usage="read-write" attribute tells Hibernate to use a read-write concurrency strategy for the defined cache.

Cache provider:

Your next step after considering the concurrency strategies you will use for your cache candidate classes is to pick a cache provider. **Hibernate forces you to choose a single cache provider for the whole application.**

Every cache provider is not compatible with every concurrency strategy. The following compatibility matrix will help you choose an appropriate combination.

Strategy/Provider	Read-only	Non-strict-read-write	Read-write	Transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBoss Cache	X			X

Batch Processing

Consider a situation when you need to upload a large number of records into your database using Hibernate. Following is the code snippet to achieve this using Hibernate:

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
   Employee employee = new Employee(.....);
   session.save(employee);
}
tx.commit();
session.close();
```

Because by default, Hibernate will cache all the persisted objects in the session-level cache and ultimately your application would fall over with an **OutOfMemoryException** somewhere around the 50,000th row. You can resolve this problem if you are using **batch processing** with Hibernate.

To use the batch processing feature, first set **hibernate.jdbc.batch_size** as batch size to a number either at 20 or 50 depending on object size. This will tell the hibernate container that every X rows to be inserted as batch. To implement this in your code we would need to do little modification as follows:

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
   Employee employee = new Employee(.....);
   session.save(employee);
    if( i % 50 == 0 ) { // Same as the JDBC batch size
          // Flush a batch of inserts
          session.flush();
         // Release memory
          session.clear();
   }
}
tx.commit();
session.close();
```

Above code will work fine for the INSERT operation, but if you are willing to make UPDATE operation then you can achieve using the following code:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults employeeCursor = session.createQuery("FROM EMPLOYEE")
                         .scroll();
int count = 0;

while ( employeeCursor.next() ) {
   Employee employee = (Employee) employeeCursor.get(0);
   employee.updateEmployee();
   seession.update(employee);
   if ( ++count % 50 == 0 ) {
      session.flush();
      session.clear();
   }
}
tx.commit();
session.close();
```

Batch Processing Example:

Let us modify configuration file as to add **hibernate.jdbc.batch_size** property:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
  <property name="hibernate.dialect">
     org.hibernate.dialect.MySQLDialect
  </property>
  <property name="hibernate.connection.driver_class">
     com.mysql.jdbc.Driver
  </property>

  <!-- Assume students is the database name -->
  <property name="hibernate.connection.url">
     jdbc:mysql://localhost/test
  </property>
  <property name="hibernate.connection.username">
     root
  </property>
  <property name="hibernate.connection.password">
     root123
  </property>
  <property name="hibernate.jdbc.batch_size">
     50
  </property>

  <!-- List of XML mapping files -->
  <mapping resource="Employee.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

Hibernate Interceptor

As you have learnt that in Hibernate, an object will be created and persisted. Once the object has been changed, it must be saved back to the database. This process continues until the next time the object is needed, and it will be loaded from the persistent store.

Thus an object passes through different stages in its life cycle and **Interceptor Interface** provides methods which can be called at different stages to perform some required tasks. These methods are callbacks from the session to the application, allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. Following is the list of all the methods available within the Interceptor interface:

S.N.	Method and Description
1	**instantiate()** This method is called when a persisted class is instantiated.
2	**onLoad()** This method is called before an object is initialized.
3	**onSave()** This method is called before an object is saved.
4	**onDelete()** This method is called before an object is deleted.
5	**findDirty()** This method is be called when the **flush()** method is called on a Session object.
6	**isUnsaved()** This method is called when an object is passed to the **saveOrUpdate()** method.
7	**preFlush()** This method is called before a flush.
8	**onFlushDirty()** This method is called when Hibernate detects that an object is dirty (ie. have been changed) during a flush i.e. update operation.
9	**postFlush()** This method is called after a flush has occurred and an object has been updated in memory.

Hibernate Interceptor gives us total control over how an object will look to both the application and the database.

How to use Interceptors?

To build an interceptor you can either implement **Interceptor** interface directly or extend **EmptyInterceptor** class. Following will be the simple steps to use Hibernate Interceptor functionality.

```
/* Method to CREATE an employee in the database */
  Session session = factory.openSession( new MyInterceptor() );
  Transaction tx = null;
  Integer employeeID = null;
  try{
    tx = session.beginTransaction();
    Employee employee = new Employee(fname, lname, salary);
    employeeID = (Integer) session.save(employee);
    tx.commit();
  }catch (HibernateException e) {
    if (tx!=null) tx.rollback();
    e.printStackTrace();
  }finally {
    session.close();
  }
  return employeeID;
```

Inheritance models in Hibernate

There are three types of inheritance models in Hibernate:

1. Table per class hierarchy
2. Table per subclass
3. Table per concrete class

Fetching Strategy

A fetching strategy is the strategy that Hibernate will use for retrieving associated objects if the application needs to navigate the association. Fetch strategies may be declared in the O/R mapping metadata, or over-ridden by a particular HQL or Query By Criteria (QBC).

Hibernate Fetching Strategies

1. fetch="join", @**Fetch**(FetchMode.**JOIN**) or **@Basic**(fetch=FetchType.**EAGER**)
Disable the **lazy loading**, always load all the collections and entities.

2. fetch="select", (**default**) @**Fetch**(FetchMode.**SELECT**) or **@Basic**(fetch=FetchType.**LAZY**)
Lazy load all the collections and entities when necessary

3. fetch="subselect", @**Fetch**(FetchMode.**SUBSELECT**)
Group its collection into a sub select statement.

4. batch-size="10", @**BatchSize**(size=10)
Fetch up to 'N' collections or entities, Not Record.

Automatic Dirty Checking

Automatic dirty checking is a feature that saves us the effort of explicitly asking Hibernate to update the database when we modify the state of an object inside a transaction.

Transactional Write-Behind

Hibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called transactional write-behind.

Hibernate Instance States

Three types of instance states are:
Transient (New) - The instance is not associated with any persistence context
Persistent - The instance is associated with a persistence context
Detached - The instance was associated with a persistence context which has been closed and currently not associated

Detached Objects

It can be passed across layers all the way up to the presentation layer without having to use any DTOs. It can be later on re-attached to another session.

Distinguish between Transient (New) Objects and Detached Objects

- Use version property if exists
- Use the identifier value. No value for new objects. Natural and assigned surrogate keys don't work.
- Own strategy with Interceptor.isUnsaved() that will be invoked when saveOrUpdate is called.

Lazy Initialization Exception

This error means that you're trying to access a lazily-loaded property or collection, but the hibernate session is closed or not available. Lazy loading in Hibernate means that the object will not be populated (via a database query) until the property or collection is accessed in code. Hibernate accomplishes this by creating a dynamic proxy object that will hit the database only when you first use the object. In order for this to work, your object must be attached to an open Hibernate session throughout its lifecycle.

One way to fix this problem is to force Hibernate to perform an Eager fetching by using 'fetch = FetchType.EAGER' on the collection of the entity class. Once again, if you are building real world applications this kind of quickfix will not work.

For example, if your entity has one collection of objects, and each one of these objects has more than one collection of objects then writing 'fetch = FetchType.EAGER' next to each collection will probably carry you to have the PersistenceException: HibernateException: cannot simultaneously fetch multiple bags. Now, instead of trying to fix our brand new bug, let's try to fix our configuration to avoid having the first LazyInitializationException in a clean way.

How to avoid the LazyInitializationException:

All you need is to mantain the session of your sessionFactory open while the processing the request. This is possible by adding an **OpenSessionInViewFilter** filter in your web.xml that will bind the session to the thread of the entire processing of the request. By doing this, **the filter will mantain the session opened** (during the request handling) **and allows the Lazy fetching**.

```
<filter>
  <filter-name>OpenSessionInViewFilter</filter-name>
  <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>OpenSessionInViewFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Pros and Cons of Lazy Loading

Lazy Initialization can help prevent unnecessary SELECT queries being executed when retrieving an object graph. But, with Lazy Loading, we have Select N+1 problem with one-to-many (parent-child) relationships while retrieving an object graph using HQL or Query by Criteria (QBC).

Solution: Eager loading by means of proper database table joins. But, the problem with Eager Loading is decreased performance.

Select N+1 Problem

Select N + 1 is a **data access anti-pattern** where the database is accessed in a suboptimal way. Take a look at this code sample, say you want to show the user all comments from all posts so that they can delete all of the nasty comments. The naive implementation would be something like:

```
// SELECT * FROM Posts
foreach (Post post in session.CreateQuery("from Post").List()) {
        //lazy loading of comments list causes:
        // SELECT * FROM Comments where PostId = @p0
        foreach (Comment comment in post.Comments)   {
                //print comment...
        }
}
```

In this example, we can see that we are loading a list of posts (the first select) and then traversing the object graph. However, we access the collection in a lazy fashion, causing Hibernate to go to the database and bring the results back one row at a time. This is incredibly inefficient.

The solution for this problem is simple: Force an eager load of the collection up front.

Using HQL:

```
var  posts = session.CreateQuery("FROM Post p LEFT JOIN FETCH p.Comments").List();
```

Using the criteria API:

```
session.CreateCriteria(typeof(Post)).setFetchMode("Comments", FetchType.EAGER).List();
```

In each case, we will get a join and only a **single query** to the database.

Hibernate – Pointers

1. **Introduction to problems**
    - Mapping member variables to columns
    - Mapping relationships
    - Handling Datatypes
    - Managing changes to object state

2. **JDBC vs Hibernate**
    - JDBC Database Configuration – Hibernate Configuration file (hibernate.cfg.xml)
    - The Model object – Annotations
    - Service method to create model objects – Hibernate API
    - Database Design – Not Needed
    - DAO Method – Not Needed

3. **Model class with Annotations**
    - @Entity - Must
    - @Id - Must

4. **Saving model objects**
   Hibernate Flow (Configuration → Session Factory → Session → Begin Transaction → Operations → Commit Transaction →Error Handling →Rollback Transaction →Finally, Close Session

5. **HBM2DDL Configuration and Name Annotations**
    - @Entity (name) - optional
    - @Column (name) – optional
    - Create and Update options in HBM2DDL configuration

6. **Other Model Annotations**
    - @Table – Customize Table Attributes
    - @Column – Customize Column Attributes
    - @Basic – Customize Fetch Type
    - @Transient – Exclude Field
    - @Temporal – Date
    - @LOB – Large Objects (Determine Blob or Clob based on the datatype)

7. **Object Retrieval**
    - Session.get()
    - Session.get() vs Session.load()

8. **Primary Keys**
    - Natural vs Surrogate
    - Surrogate - @GeneratedValue(StrategyType. AUTO, IDENTITY, SEQUENCE and TABLE)

9. **Value Types or Value Objects and Embedded Objects**
    - @Embeddable
    - @Embedded

10. **Attribute Overrides and Embedded Object Keys**
    - @AttributeOverrides and @AttributeOverride – Overrides the default @Column attributes
    - @EmbeddedId – Composite Primary Key

11. **Saving Collections**
    - @ElementCollection

12. **Configuring Collections and Adding Keys**
    - @JoinTable
    - @JoinColumn
    - @CollectionId – Primary Key Column – Hibernate Feature (Not in JPA specification)
    - @GenericGenerator – Key Generation (hilo)
    - @Type

13. **Proxy Objects and Eager/Lazy Fetch Types**
    - @Basic(FetchType.EAGER or LAZY)
    - @FetchMode(FetchType.EAGER or LAZY)
    - First Level – Eager and Sub Levels – Lazy
    - Lazy Initialization Exception – Fetching the association after the session is closed
    - OpenSessionInViewFilter – Prevents Lazy Initialization Exception

14. **One to One Mapping**
    - @Entity @OnetoOne – Same Table
    - @JoinColumn – Customize the join column attributes

15. **One to Many Mapping or Many to One Mapping**
    - @Entity @OnetoMany or @ManytoOne – Separate Table
    - @JoinTable – Customize the join table attributes
    - @OnetoMany (mappedBy = “TableName”) – No Separate Join Table
    - @JoinColumn – Customize the join column attributes

16. **Many to Many Mapping**
    - @Entity @ManytoMany – Two Separate Tables
    - @JoinTable – Customize the join table attributes
    - @ManytoMany (mappedBy =”TableName”) – One Separate Join Table – Prevent two tables
    - @JoinColumn – Customize the join column attributes
    - @InverseJoinColumn – Customize the inverse join column attributes

17. **Cascade Types & Entity Relationships**
    - @Entity(CascadeType.ALL, PERSIST (Multiple Saves), DETACH, MERGE, REFRESH, REMOVE and INVERSE)
    - Session.persist()
    - @NotFound (ActionType.IGNORE) – Ignores when the associated object is missing

18. **Hibernate Collections**
    - Bag (ArrayList)
    - Bag with Id (ArrayList)
    - List – Ordered and Duplicates(ArrayList)
    - Set – Unordered and No Duplicates
    - Map – Key/Value Pairs

19. **Inheritance in Hibernate**
    - Entity Class extends another Entity Class – DTYPE (Column) – Sub Class Names (Values)
    - Strategies – Single Table (Default), Table per Class and Joined

20. **Single Table Inheritance**
    - Super Class @Entity @Inheritance (StrategyType.SINGLE_TABLE)
    - Disadvantage: Null Columns
    - @Discriminator Column (name, type) – Customize the discriminator column attributes (DTYPE)
    - @DiscriminatorValue - Customize the discriminator value (Sub Class Name)

21. **Table per Class Inheritance**
    - Separate tables per class – Avoid blank columns but introduce Redundancy with multiple duplicate columns
    - @Entity @Inheritance (StrategyType.TABLE_PER_CLASS)

22. **Inheritance with Joined**

    @Entity @Inheritance (StrategyType.JOINED) – Prevent Redundancy with duplicate columns

23. CRUD Operations

- save or persist – Track down the object state and issue updates wisely (Automatic Dirty Checking)
- get or load – Automatic Dirty Checking
- update or merge
- delete

24. Hibernate Instance States

- Transient (New) – not associated with any persistent context
- Persistent (In Session) – associated with a persistent context
- Detached (Removed) – was associated with a persistent context, closed or currently not associated

25. Understanding Instance States

- Transient → Persistent → Detached
- New → Transient → Save → Persistent → Close → Detached
- Get → Persistent → Close → Detached
- Transient ← Delete ← Persistent → Close → Detached
- Transient → Open Session → Persistent → Close Session → Detached

26. Persisting Detached Objects

- New Session - No changes will also issue UPDATE
- @Entity @Entity (selectBeforeUpdate = true) – Special Hibernate Annotation - Executes UPDATE query only if the SELECT indicates a change

27. Hibernate Query Language (HQL)

- More control like Pagination, Filter, etc.
- Session.createQuery(query).list()
- Session.createQuery(query).uniqueResult() with group functions such as COUNT, DISTINCT, AVG, SUM, MAX and MIN

28. HQL Pagination and Filtration

- setFirstResult (startPosition) – Offset
- setMaxResults (maxCount) – Limit
- SELECT new Map (key, value) FROM ClassName

29. Parameter Binding and SQL Injection

- Appending parameters in a simple statement will led to SQL Injection Attack – "QUERY WHERE COLNAME =" + value – Allows attacker to inject 'value OR 1 = 1' – Exposes the protected and confidential data
- Use Prepared Statement with either '?' or ':name' – Named Parameter, get(index) or get(parameterName) respectively

30. Named Queries

- Best Practice – One common place for all queries related to the entity – Entity Class is the best place to group them – Reusable
- @NamedQuery (name, HQL)
- @NamedNativeQuery (name, SQL, ResultClass)
- Session.getNamedQuery(queryName)

31. Criteria API

- No need for SQL or HQL – Third way to extract data – Best among three – HQL uses Query no different from SQL which is complex too
- Criteria c = Session.createCriteria(Class)

32. Restrictions with Criteria

- Chain criterions with add methods – Criteria.add(c1).add(c2)…
- Criterion c1 = Restrictions.gt(colName, value)
- Restrictions.eq(), ge(), le(), lt(), gt(), like(), ilike(), isNull(), isNotNull(), isEmpty(), isNotEmpty and between() are some of them
- Logical Expressions are Restrictions.and(c1, c2) and Restrictions.or(c3,c4)
- Pagination methods – setFirstResult (startPosition) and setMaxResults (maxCount)
- Sorting the results – Criteria.addOrder(Order.asc/desc(fieldName))

33. Projections, Aggregations and Query by Example

- Criteria.setProjection(Projections.rowCount())
- property() and aggregate (group) functions such as countDistinct(), rowCount(), avg(), max(), min() and sum() are some of the projections
- Query by Example – Create Example object with default values – Bunch of properties to set in a query – Fetches the records that satisfies the property values in Example object
- Criteria.add(Example.create(exampleObject))
- Ignores null values and primary key
- Exclude the property using Example.excludeProperty(propertName) method
- Set the property value with % and enable LIKE capability using Example.enableLike() method

34. **Hibernate Cache**

- First Level Cache (Default) – Session Cache – Mandatory - Minimizes the query (SELECT and UPDATE) execution
- Second Level Cache – Optional – Third Party Vendor
- Query Cache – Optional – Third Party Vendor

35. **Second Level Cache**

- Caching across sessions, applications and clusters
- Configurable - hibernate.cfg.xml – cache.provider_class – Determined by Concurrency Strategies - EHCacheProvider/OSCacheProvider/SwarmCacheProvider/JBossCacheProvider
- Turn on the second level cache (optional) – cache.use_second_level_cache – true
- Annotations - @Entity @Cacheable @Cache(usage=CacheConcurrencyStrategy.NONE, TRANSACTIONAL, READ_WRITE, NON_STRICT_READ_WRITE, READ_ONLY)

36. **Query Cache**

- HQL and Criteria are stored in a different memory location – Second Level Caching mechanism will not cache them
- Configurable - hibernate.cfg.xml – cache.use_query_cache – true
- Query.setCacheable(true) or Criteria.setCacheable(true) for different sessions
- Use them with care – End up querying a lot of data – Cache something not required - Caching them make the system inefficient

Chapter 25 – iBatis

iBatis

ORM - object model is not in sync with data model - objects are mapped to result set of a query or stored procedure.

JDBC Framework

- developers write SQL
- no try/catch/finally

SQL Mapper

- automatically maps object properties to prepared statement parameters
- automatically maps result sets to objects
- support for getting rid of N+1 queries

Transaction Manager

- provides transaction management for database operations
- support transaction management (Spring, EJB CMT)

Great integration with Spring

It's an ORM

- does not generate SQL
- does not have a proprietary query language
- does not know about object identity
- does not transparently persist objects
- does not build an object cache

Chapter 26 – Framework Configuration and Integration

Framework Configuration Files

JSF
Config File: faces-config.XML

Struts
Config File: struts-config.XML

Spring
Config File: applicationContext.XML

Hibernate
Config File: hibernate.cfg.XML
Mapping File: xxx.hbm.XML

iBatis
Config File: SqlMapConfig.XML
Mapping File: SqlMap.XML

Web Application
Config File: web.XML

Framework Integration

Struts+Spring+Hibernate (Web Framework+ DI Framework + ORM Framework)

Web Page -> DI -> DAO -> DB
1+2 = Spring's Struts Plug-in "ContextLoaderPlugin"
- Action class extends Spring's ActionSupport class
- Get Spring bean via getWebApplicationContext()
- struts-config.XML
- spring-web.jar & spring-struts.jar

2+3= Spring's "LocalSessionFactoryBean"
applicationContext.XML
- Spring's DriverManagerDataSource, DataSource.XML & PropertyPlaceHolderConfigurer, DB.properties
- Spring's HibernateSessionFactory.XML

- DAO class extends HibernateDAOSupport class

JSF+Spring+Hibernate (Web Framework+ DI Framework + ORM Framework)

1+2= faces-config.XML
- SpringBeanFacesELResolver

web.XML
- Spring's ContextLoadListener
- Spring's RequestContextListener
- JSF's FacesServlet

JSF+Spring+iBatis (Web Framework+ DI Framework + ORM Framework)

2+3=Minimum Three Configuration files + web.XML
1. Spring Config -
applicationContext.XML
Spring's SqlMapClientFactoryBean
Instance of ClassPathXmlApplicationContext(applicationContext.XML).getBean(beanName)
2. SQL Map Config - SqlMapConfig.XML
SqlMapClient.queryForObject(sqlName, input) or update(same)
3. SQL Map - domainObject.XML

Struts+Hibernate (Web Framework + ORM Framework)

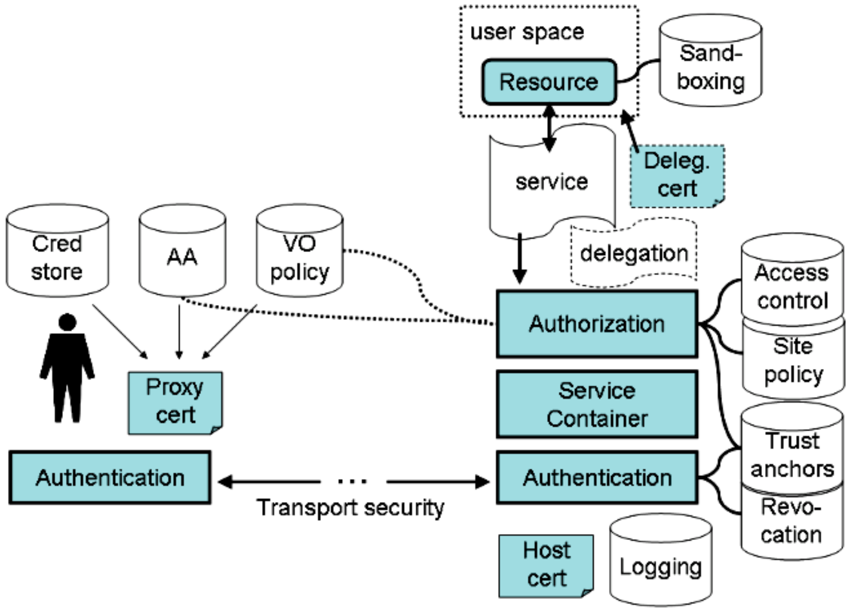
- Create Hibernate Struts Plug-in
- Get Hibernate Session Factory
- Store it into the Servlet context
- Include it in struts-config.XML
  <plug-in>

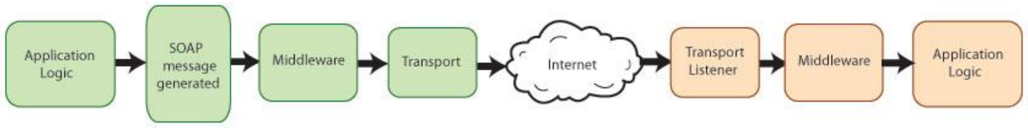
Implements Plugin
Init(ActionServlet, ModuleConfig)

Chapter 27 – Apache Axis

Apache Axis

Apache Axis is an execution of Simple Object Access Protocol also called as SOAP submission to W3C. SOAP is a lightweight protocol meant for the purpose of exchange of controlled information in a distributed and decentralized environment. It is a protocol based on Extensible Markup Language, which consists of three parts: a standard for representing remote procedure responses and calls, a combination of encoding rules to express examples of application-defined data types and an envelope defining a structure for describing the content of a message and the method of processing it.





Apache Axis 2

The Apache Axis2 project is a Java-based implementation of both the client and server sides of the Web services equation. Designed to take advantage of the lessons learned from Apache Axis 1.0, Apache Axis2 provides a complete object model and a modular architecture that makes it easy to add functionality and support for new Web services-related specifications and recommendations.

- complete redesign and rewrite
- more efficient, more modular and more XML-oriented
- easy addition of Plugin "modules" such as Security and Reliability

Axis2 enables you to easily perform the following tasks:

1. Send SOAP messages
2. Receive and process SOAP messages
3. Create a Web service out of a plain Java class
4. Create implementation classes for both the server and client using WSDL
5. Easily retrieve the WSDL for a service
6. Send and receive SOAP messages with attachments
7. Create or utilize a REST-based Web service
8. Create or utilize services that take advantage of the WS-Security, WS-ReliableMessaging, WS-Addressing, WS-Coordination, and WS-Atomic Transaction recommendations
9. Use Axis2's modular structure to easily add support for new recommendations as they emerge

Enhancements

Apache Axis2 not only supports SOAP 1.1 and SOAP 1.2, but it also has integrated support for the widely popular REST style of Web services. The same business logic implementation can offer both a WS-* style interface as well as a REST/POX style interface simultaneously.

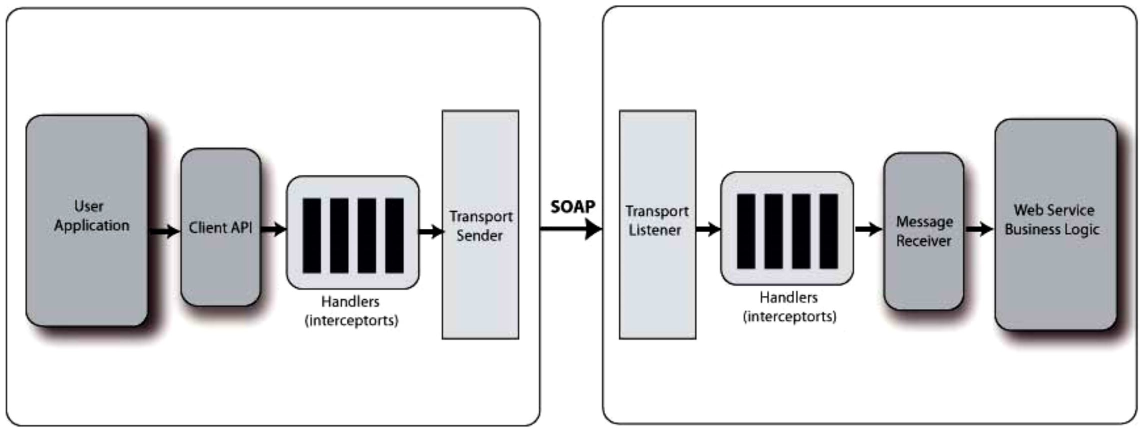
Axis2 has support for Spring Framework

Axis2 comes with many new features, enhancements and industry specification implementations. The key features offered are as follows:

- Speed
- Low memory foot print
- AXIOM - **AXI**s2 comes with its own light-weight **O**bject **M**odel
- Hot Deployment
- Asynchronous Web services
- Message Exchange Patterns (MEPs) Support
- Flexibility
- Stability
- Component-oriented Deployment
- Transport Framework
- WSDL support - supports version 1.1 and 2.0
- Add-ons - WSS4J for security (Apache Rampart), Sandesha for reliable messaging, Kandula which is an encapsulation of WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity.
- Composition and Extensibility

How Axis2 Handles SOAP Messages

Axis2 can handle processing for both the sender and the receiver in a transaction. From the Axis2 perspective, the structure looks like this:

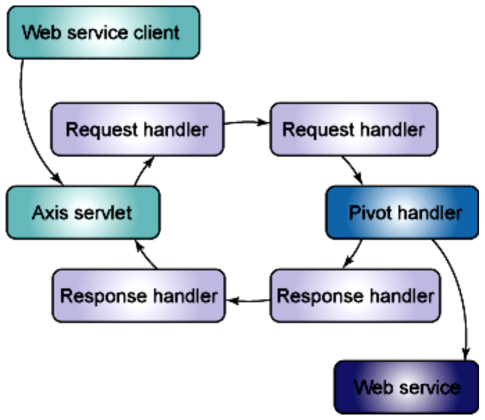


On each end, you have an application designed to deal with the (sent or received) messages. In the middle, you have Axis2, or rather, you can have Axis2. The value of Web services is that the sender and receiver (each of which can be either the server or the client) don't even have to be on the same platform, much less running the same application. Assuming that Axis2 is running on both sides, the process looks like this:
The sender creates the SOAP message.
Axis "handlers" perform any necessary actions on that message such as encryption of WS-Security related messages.
The transport sender sends the message.
On the receiving end, the transport listener detects the message.
The transport listener passes the message on to any handlers on the receiving side.
Once the message has been processed in the "pre-dispatch" phase, it is handed off to the dispatchers, which pass it on to the appropriate application.
In Axis2, these actions are broken down into "phases", with several pre-defined phases, such as the "pre-dispatch", "dispatch," and "message processing", being built into Axis2. Each phase is a collection of "handlers". Axis2 enables you to control what handlers go into which phases, and the order in which the handlers are executed within the phases. You can also add your own phases and handlers.

Handlers come from "modules" that can be plugged into a running Axis2 system. These modules, such as Rampart, which provides an implementation of WS-Security, and Sandesha, which provides an implementation of WS-ReliableMessaging, are the main extensibility mechanisms in Axis2.

Axis Flow Architecture

Apache's first SOAP implementation relied on the concept of a provider. This bit of code decoded the object and method from a SOAPenvelope, instantiated the local object, and returned the result to be wrapped up in a return SOAP envelope. Axis has a provider equivalent, but often it is referred to as the pivot point. This is because Axis provides an architecture that you can use to define transaction flows. You define the flows through handlers. Each handler is able to act on the current incoming and outgoing SOAP envelope. Handlers provide the ability to perform methods and services as part of a flow for a larger service, as shown in Figure 1. A pivot point is just another handler, but it indicates the point at which the request becomes a response. The pivot point is one place where you can insert your cache.



Message Exchange Patterns (MEPs)

Although all SOAP messages carry the same structure, the ways in which they are used can be combined into a number of different "message exchange patterns", or MEPs. The two major message exchange patterns are:

In-Out: in this MEP, the client sends a SOAP message to the server, which processes the message and sends a response back. This is probably the most commonly used MEP, and is useful for tasks such as searching for information or submitting information in situations in where acknowledgment is important.

In-Only: In this MEP, the client sends a message to the server without expecting a response. You may use this MEP for activities such as pinging a server to wake it up, reporting logging information for which you do not need an acknowledgment and so on.
Within these two MEPs, you also have several variables to consider:

Blocking versus non-blocking: When the client sends a message, the application may wait to receive a response before moving on, or it may simply send a message and move on by specifying a callback action to be completed when the response is received.

Number of parameters: Ultimately, a message sent from a client to server is intended to execute a particular action. That action may not require any parameters, or it may require one or more parameters. These parameters must be encoded as part of the payload of the message.

Taking all these options into consideration, you can create virtually any MEP. For example, you can create an Out-Only system by reversing roles for the In-Only MEP. Apache Axis2 also includes support for less prominent MEPs, such as Robust-In-Only.

Choosing a Client Generation Method

Axis2 gives you several options when it comes to mapping WSDL to objects when generating clients. Three of these options are Axis2 Data Binding Framework, XMLBeans, and JiBX data binding. All of these methods involve using data binding to create Java objects out of the XML structures used by the service, and each has its pros and cons. You can also generate XML in-out stubs that are not based on data binding.

Axis2 DataBinding Framework (ADB)

ADB is probably the simplest method of generating an Axis2 client. In most cases, all of the pertinent classes are created as inner classes of a main stub class. ADB is very easy to use, but it does have limitations. It is not meant to be a full schema binding application, and has difficulty with structures such as XML Schema element extensions and restrictions.

XMLBeans

Unlike ADB, XMLBeans is a fully functional schema compiler, so it doesn't carry the same limitations as ADB. It is, however, a bit more complicated to use than ADB. It generates a huge number of files, and the programming model, while being certainly usable, is not as straightforward as ADB.

JiBX

JiBX is a complete data binding framework that actually provides not only WSDL-to-Java conversion, as covered in this document, but also Java-to-XML conversion. In some ways, JiBX provides the best of both worlds. JiBX is extremely flexible, enabling you to choose the classes that represent your entities, but it can be complicated to set up. On the other hand, once it is set up, actually using the generated code is as easy as using ADB.

In the end, for simple structures, ADB will likely be enough for you. If, on the other hand you need more power or flexibility, whether you choose XMLBeans or JiBX depends on how much power or flexibility you need and your tolerance for complexity.

Choosing a Service Creation Method

Axis2 provides a number of ways to create a service, such as:

- Create a service and build it from scratch. In this case, you build your service class to specifically access AXIOM OMElement objects, then create the services.xml file and package it for deployment.
- Deploy Plain Old Java Objects (POJOs) as a service.
- Generate the service from WSDL. Just as you can generate clients with WSDL, you can also generate the skeleton of a service.

Writing client using Apache SOAP API:
Service service = new Service();
Call call = (Call)service.createCall();
//Set the target service host and service location
call.setTargetEndpointAddress(new URL(http://localhost:8080/axis/services));

```
//Invoke the operation
call.setOperationName(new QName("EchoService", "echo"));
call.addParameter("in0", XMLType.XSD_STRING, ParameterMode.IN);
call.setReturnType(XMLType.XSD_STRING);
Object ret = call.invoke(new Object[]{in0});
//The object ret will either be a String on success or RemoteException on failure.
```

Four Client files generated by WSDL2Java:
1. Echo.java
2. EchoService.java
3. EchoServiceLocator.java
4. EchoSoapBindingStub.java

Writing client using generated stubs:

```
//Create Echo client object and point to the service you want to use
Echo echo = new EchoServiceLocator().getEcho(new URL(http://server:port/axis/services/Echo));
//Invoke the method as if local
String backTalk = echo.echo("Hi");
```

Chapter 28 – Jersey

Jersey Framework

Jersey framework is more than the JAX-RS Reference Implementation. Jersey provides its own API that extends the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development.

Jersey Annotations (javax.ws.rs)

Annotation	Description	Usage

@GET	Indicates that the following method will answer to an HTTP GET/Retrieve request.	**@GET**
		public String getHTML() {
		  ...
		}

@POST	Indicates that the following method will answer to an HTTP POST/Create request.	**@POST**
		@Consumes("application/json")
		@Produces("application/json")
		public RestResponse<Contact> create(Contact contact) {
		...
		}

@PUT	Indicates that the following method will answer to an HTTP PUT/Update request.	**@PUT**
		@Consumes("application/json")
		@Produces("application/json")
		@Path("{contactId}")
		public RestResponse<Contact> update(Contact contact) {
		...
		}

@DELETE	Indicates that the following method will answer to an HTTP DELETE/Remove request.	**@DELETE**
		@Produces("application/json")
		@Path("{contactId}")
		public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {
		...
		}

@Consumes (MediaType.APPLICATION_XML[, more-types])	'Consumes' defines which MIME type is consumed by a method annotated with @POST, @PUT or @DELETE.	@PUT **@Consumes("application/json")** @Produces("application/json") @Path("{contactId}") public void delete(Contact contact) { ... }

@Produces (MediaType.TEXT_PLAIN[, more-types])	‘Produces’ defines which MIME type is delivered or prodcued by a method annotated with @GET. The standard outputs are "text/plain", “text/html”, "application/xml" and "application/json".	@GET
		@Produces("application/json")
		public Contact getJSON() {
		  ...
		}

@Path(your_path)	Specifies the URL path on which this method will be invoked. The base URL is based on your application name, the servlet and the URL pattern from the web.xml configuration file.	@GET
		@Produces("application/xml")
		@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
		public Contact getXML() {
		  ...
		}

@PathParam	Used to inject values from the URL into a method parameter. We can bind REST-style URL parameters to method arguments using @PathParam annotation.	@GET
		@Produces("application/xml")
		@Path("xml/{firstName}")
		public Contact getXML(**@PathParam("firstName") String firstName**) {
		  Contact contact = contactService.findByFirstName(firstName);
		  return contact;
		}

@QueryParam	Request parameters in query string can be accessed using @QueryParam annotation.	@GET
		@Produces("application/json")
		@Path("json/companyList")
		public CompanyList getJSON(**@QueryParam("start") int start**, @QueryParam("limit") int limit) {
		  CompanyList list = new CompanyList(companyService.listCompanies(start, limit));
		  return list;
		}

@FormParam	The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly using @FormParam.	@POST
		public String save(**@FormParam("firstName") String firstName**,
		    @FormParam("lastName") String lastName) {
		      ...
		 }

@HeaderParam	Binds the value(s) of a HTTP header to a resource method parameter, resource class field, or resource class bean property.	public class MyBeanParam {
		    @HeaderParam("header")
		    private String headerParam;
		}

@CookieParam	Binds the value of a HTTP cookie to a resource method parameter, resource class field, or resource class bean property.	public class MyBeanParam {
		    @CookieParam("cookieName")
		    private String cookieParam;
		}

@MatrixParam	Binds the value(s) of a URI matrix parameter to a resource method parameter, resource class field, or resource class bean property.	public class MyBeanParam {
		    @MatrixParam("m")
		    @Encoded
		    @DefaultValue("default")
		    private String matrixParam;
		}

@BeanParam	It can contain all parameter injections. The JAX-RS runtime will instantiate the object and inject all its fields and properties annotated with either one of the @XxxParam annotation or the @Context annotation.	public class MyBean {
		  @FormParam("myData")
		  private String data;
		  @HeaderParam("myHeader")
		  private String header;
		  @PathParam("id")
		  public void setResourceId(String id) {...}
		 }

		@Path("myresources")
		public class MyResources {
		@POST
		@Path("{id}")
		public void post**(@BeanParam MyBean myBean**) {...}
		}

@DefaultValue	Defines the default value of request meta-data that is bound using either one of PathParam, QueryParam, MatrixParam, CookieParam, FormParam or HeaderParam.	@Path("{id:\\d+}")
		public class InjectedResource {
		// Injection onto field
		@DefaultValue("q") @QueryParam("p")
		private String p;
		}

@Encoded	Disables automatic decoding of parameter values bound using QueryParam, PathParam, FormParam or MatrixParam.	public class MyBeanParam {
		@MatrixParam("m")
		@Encoded
		@DefaultValue("default")
		private String matrixParam;
		}

@Context	This annotation is used to inject information into a class field, bean property or method parameter. When deploying a JAX-RS application using servlet then ServletConfig, ServletContext, HttpServletRequest and HttpServletResponse are available using @Context.	@GET
		public String get(**@Context UriInfo ui**) {
		MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
		MultivaluedMap<String, String> pathParams = ui.getPathParameters();
		}
		
		@GET
		public String get**(@Context HttpHeaders hh**) {
		MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
		Map<String, Cookie> pathParams = hh.getCookies();
		}
		
		@Context
		public void setRequest(**Request request**) {
		String[] requestParams = request.getRequestParams();
		}
		
		public MySingletonResource**(@Context SecurityContext securityContext**) {
		...
		}

@Singleton	In this scope there is only one instance per jax-rs application.	**@Singleton**
		@Path("/printers")
		public class PrintersResource {
		...
		}

@RequestScoped	Default lifecycle (applied when no annotation is present). In this scope the resource instance is created for each new request and used for processing of this request.	**@RequestScoped**
		@Path("/pathUrl")
		public class UserAction {
		...
		}

@PerLookup	In this scope the resource instance is created every time it is needed for the processing even it handles the same request.	**@PerLookup**
		@Path("/pathUrl")
		public class GranularTransaction {
		...
		}

Chapter 29 – Messaging Service

Java Messaging Service (JMS)

The Java Message Service (JMS) defines the standard for reliable Enterprise Messaging.

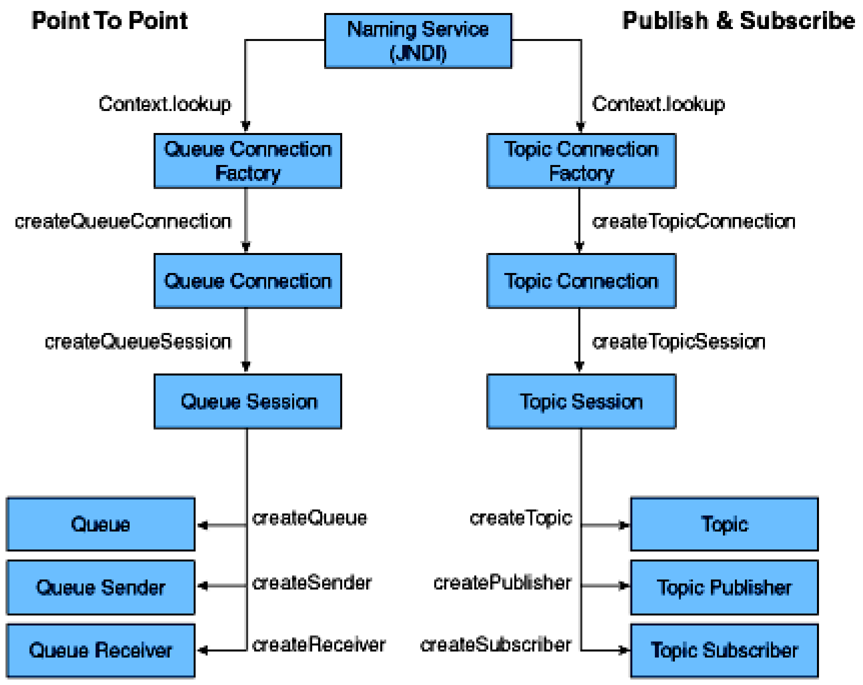
The JMS Elements are:

1. JMS Provider
2. JMS Client
3. JMS Producer/Publisher
4. JMS Consumer/Subscriber
5. JMS Message
6. JMS Queue
7. JMS Topic

The Messaging Models are:

1. Point-to-Point (P2P) or Queuing (Push - Online - Persistence possible)
2. Publish and Subscribe or Topic (Pull - Offline - Store and Forward)

JMS class hierarchy



Chapter 30 – Enterprise Bean

Enterprise Java Beans (EJB)

EJB is a managed, server-side component architecture for modular construction of enterprise applications. It's a server-side model that encapsulates the business logic of an application.

The application server provides:

1. Transaction Processing (XA-1)
2. Integrated with Persistence services (JPA)
3. Concurrent Control
4. Events using JMS
5. Naming and Directory Service (JNDI)
6. Security (JCE & JAAS)
7. RPCs using RMI-IIOP
8. Exposing Business methods as Web Services.

EJB Types

Session Beans
Stateful, Stateless or Singleton accessed via either a Local or Remote or directly without an interface. Support asynchronous execution for all views. It represents a single unit of work or operation.

Message Driven Beans
Message Beans support asynchronous execution, but via a messaging paradigm.

Legacy (Deprecated) Beans
Entity beans were distributed objects having persistent state. **CMP and BMP are two types of Entity beans** that were replaced by the Java Persistence API in EJB 3.0. It represents a single record in the database table wrapped with the business logic.

EJB History

EJB 3.1

- Local view without interface
- .war EJB packaging
- Singleton Session Beans
- Application & Shutdown events
- EJB Timer Service Enhancements
- Asynchronous for session beans

EJB 3.0

- EJBs using annotations rather than complex deployment descriptors
- Use of home, remote interfaces and EJB-jar.XML no longer required.
- Replaced with business interface and a bean that implements it.

EJB 2.1

- Web service support - Stateless session bean invoked over SOAP/HTTP
- EJB Timer Service
- MDBs accepts message from sources other than JMS
- EJB-QL additions - ORDER BY, AVG, SUM, MIN, MAX, COUNT, and MOD
- XML schema replaces DTD as deployment descriptor

EJB 2.0

- Be compatible with CORBA protocols (RMI-IIOP)

EJB 1.1

- XML deployment descriptors
- Default JNDI contexts
- RMI over IIOP
- Security - role driven, not method driven
- Entity Bean Support – mandatory

EJB 1.0 1998

- Defined responsibilities of EJB container, roles, client view and developer's view

Transactions

Six Transaction Attributes

1. Required
2. Requires New
3. Supports
4. Not Supported
5. Mandatory - Transaction Required Exception
6. Never - Remote or EJB Exception

ACID Properties of a Transaction

- Atomicity - Do all or Nothing
- Consistency - Stable Data before/after/failed transactions (Data Integrity)
- Isolation - Isolate current transaction from others
- Durability - Persist the data

Distributed Transactions (XA) & Two-phase Commit Protocol

1. Commit Request Phase (Voting)
2. Commit Phase

Three Different Read Problems

Dirty Reads
One transaction reads rolled back data (stale record) of another transaction.

Non Repeatable Reads
One transaction reads two different values of the same record that was updated by another transaction between two successive reads.

Phantom Reads
One transaction finds different record count between two successive executions due to the record inserted by another transaction during execution.

Transaction Isolation Levels

1. TRANSACTION_READ_UNCOMMITTED – has **all three read problems**
2. TRANSACTION_READ_COMMITTED - prevents **Dirty Reads**
3. TRANSACTION_REPEATABLE_READ - prevents **Dirty Reads and Non Repeatable Reads**
4. TRANSACTION_SERIALIZABLE - prevents **Dirty Reads, Non Repeatable Reads and Phantom Reads**

Performance decreases as the level increases.

Chapter 31 – Persistence

Java Persistence API (JPA)

- javax.persistence package
- JPQL (Java Persistence Query Language)
- Object/Relational metadata

JPA Annotations for Classes
JPA defines three types of persistable classes which are set by the following annotations:

@javax.persistence.Embeddable
@javax.persistence.Entity
@javax.persistence.MappedSuperclass

Entity and mapped super classes can be further configured by annotations that specify cache preferences and lifecycle event listener policy (as explained in chapter 3):

@javax.persistence.Cacheable
@javax.persistence.EntityListeners
@javax.persistence.ExcludeDefaultListeners
@javax.persistence.ExcludeSuperclassListeners

Another JPA class annotation defines an ID class:

@javax.persistence.IdClass

ID classes are useful in representing composite primary keys as explained in the Primary Key section of the ObjectDB manual.

JPA Annotations for JPQL Queries

The following annotations are used to define static named JPA queries:

@javax.persistence.NamedQueries

@javax.persistence.NamedQuery
@javax.persistence.QueryHint

The JPA Named Queries section of the ObjectDB Manual explains and demonstrates how to use these annotations to define named JPQL queries.

JPA Annotations for Fields

The way a field of a persistable class is managed by JPA can be set by the following annotations:

@javax.persistence.Basic
@javax.persistence.Embedded
@javax.persistence.ElementCollection
@javax.persistence.Id
@javax.persistence.EmbeddedId
@javax.persistence.Version
@javax.persistence.Transient

Additional annotations (and enum) are designated for enum fields:

@javax.persistence.Enumerated
@javax.persistence.MapKeyEnumerated
@javax.persistence.EnumType

Other additional annotations (and enum) are designated for date and calendar fields:

@javax.persistence.Temporal
@javax.persistence.TemporalType
@javax.persistence.MapKeyTemporal

JPA Annotations for Relationships

Relationships are persistent fields in persistable classes that reference other entity objects. The four relationship modes are represented by the following annotations:

@javax.persistence.ManyToMany
@javax.persistence.ManyToOne
@javax.persistence.OneToMany
@javax.persistence.OneToOne

Unlike ORM JPA implementations, ObjectDB does not enforce specifying any of the annotations above. Specifying a relationship annotation enables configuring cascade and fetch policy, using the following enum types:

@javax.persistence.CascadeType
@javax.persistence.FetchType

Additional annotations are supported by ObjectDB for the inverse side of a bidirectional relationship (which is calculated by a query):

@javax.persistence.OrderBy
@javax.persistence.MapKey

JPA Annotations for Access Modes

Persistence fields can either be accessed by JPA directly (as fields) or indirectly (as properties and get/set methods). JPA 2 provides an annotation and an enum for setting the access mode:

@javax.persistence.Access
@javax.persistence.AccessType

JPA Annotations for Value Generation

Automatically generated values are mainly useful for primary key fields, but are supported by ObjectDB also for regular (non primary key) persistent fields.

At the field level, the @GeneratedValue with an optional GenerationType strategy is specified:

@javax.persistence.GeneratedValue
@javax.persistence.GenerationType

The @GeneratedValue annotation can also reference a value generator, which is defined at the class level by using one of the following annotations:

@javax.persistence.SequenceGenerator
@javax.persistence.TableGenerator

JPA Annotations for Callback Methods

The following annotations can mark methods as JPA callback methods:

@javax.persistence.PrePersist
@javax.persistence.PreRemove
@javax.persistence.PreUpdate
@javax.persistence.PostLoad
@javax.persistence.PostPersist
@javax.persistence.PostRemove
@javax.persistence.PostUpdate

The Lifecycle Events section of the ObjectDB Manual explains how to use all these annotations on callback methods and with listener classes.

JPA Annotations for Java EE

The following JPA annotations are in use to integrate JPA into a Java EE application and are managed by the Java EE container:

@javax.persistence.PersistenceContext
@javax.persistence.PersistenceContextType
@javax.persistence.PersistenceContexts
@javax.persistence.PersistenceProperty
@javax.persistence.PersistenceUnit
@javax.persistence.PersistenceUnits

Java API for Database Connectivity (JDBC)

Types of JDBC Drivers:

1. JDBC-ODBC Bridge Driver (Bridge) – Type 1
2. Native-API/Partly Java Driver (Native) – Type 2
3. Pure (All) Java, Net-Protocol Driver (Middleware – w3:) – Type 3
4. Pure (All) Java, Native-Protocol Driver (Thin – mysql:) – Type 4

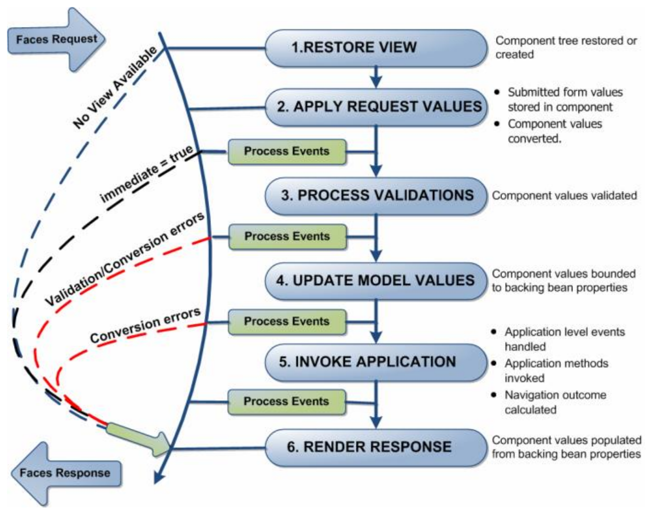
Chapter 32 – User Interface

Java Server Faces (JSF)

A server side user interface component framework for Java™ technology-based web applications. Java Server Faces (JSF) is an industry standard and a framework for building component-based user interfaces for web applications.

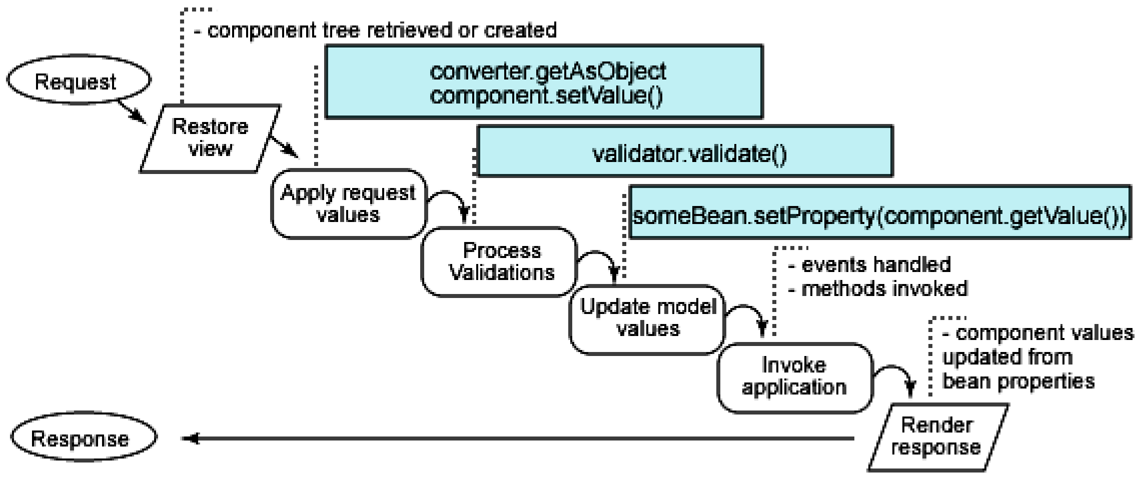
JSF contains an API for representing UI components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features.

JSF Life Cycle


Faces Request
No View Available
immediate = true
Validation/Conversion errors
Conversion errors
Faces Response
1.RESTORE VIEW
Component tree restored or created
2. APPLY REQUEST VALUES
Submitted form values stored in component
Component values converted.
Process Events
3. PROCESS VALIDATIONS
Component values validated
Process Events
4. UPDATE MODEL VALUES
Component values bounded to backing bean properties
Process Events
5. INVOKE APPLICATION
Application level events handled
Application methods invoked
Navigation outcome calculated
Process Events
6. RENDER RESPONSE
Component values populated from backing bean properties

The phases of the JSF application lifecycle are as follows:

1. Restore view
2. Apply request values; process events
3. Process validations; process events
4. Update model values; process events
5. Invoke application; process events
6. Render response



The major benefits of Java Server Faces technology are:

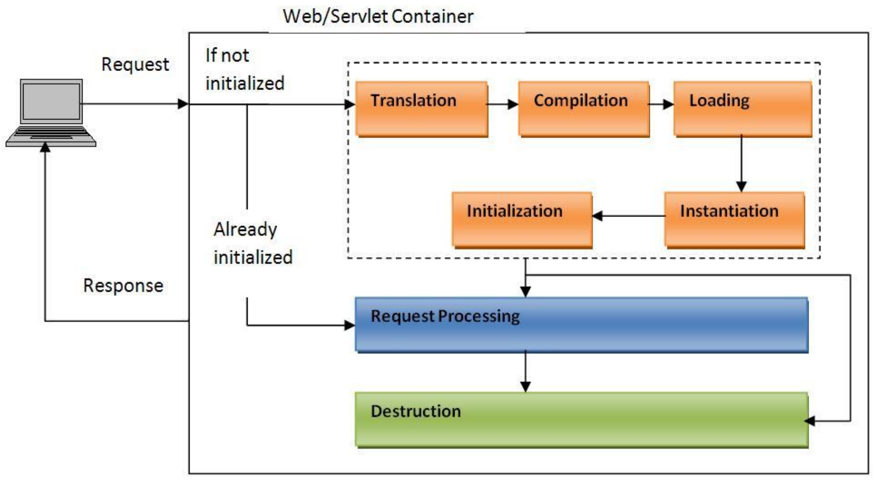
- Java Server Faces architecture makes it easy for the developers to use. In Java Server Faces technology, user interfaces can be created easily with its built–in UI component library, which handles most of the complexities of user interface management.
- Java Server Faces technology offers a clean separation between behavior and presentation.
- Java Server Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.
- Robust event handling mechanism.
- Events easily tied to server-side code.
- Component-level control over stateful-ness
- Render kit support for different clients
- JSF also supports internationalization and accessibility
- Highly 'pluggable' – components, view handler, etc.
- Offers multiple, standardized vendor implementations

Java Server Pages (JSP)

A server-side technology, Java Server Pages are an extension to the Java servlet technology that was developed by Sun to develop interactive web applications based on HTML, XML, or other document types.

JSP Life Cycle

1. Page translation
2. JSP page compilation
3. Load class
4. Create instance
5. Call jspInit
6. Call _jspService
7. Call jspDestroy



Basically, there are two phases, translation and execution. During the translation phase the container locates or creates the JSP page implementation class that corresponds to a given JSP page. This process is determined by the semantics of the JSP page. The container interprets the standard directives and actions, and the custom actions referencing tag libraries used in the page. A tag library may optionally provide a validation method to validate that a JSP page is correctly using the library. A JSP container has flexibility in the details of the JSP page implementation class that can be used to address quality-of-service and most notably, the performance issues.

During the execution phase the JSP container delivers events to the JSP page implementation object. The container is responsible for instantiating request and response objects and invoking the appropriate JSP page implementation object. Upon completion of processing, the response object is received by the container for communication to the client.

JSP Tag Types

1) Directive Tags
These tags control translation and compilation phases.
<%@Directive {attribute=”value”}* %>
<%@page …%> <jsp:directive.page …/>
<%@taglib uri=http://my.com/my.tld prefix=”my” %>
<%@include page=”enterRecord.jsp” flush=true”%>

2) Declaration Tags
These tags declare an instance java variable or method. No output produced
<%! JavaDeclaration %> equivalent to XML tag <jsp:declaration> … </jsp:declaration>

3) Scriptlets
These tags have valid java statements. No output produced unless out is used.
<% JavaStatements %> equivalent to XML tag <jsp:scriptlet> … </jsp:scriptlet>

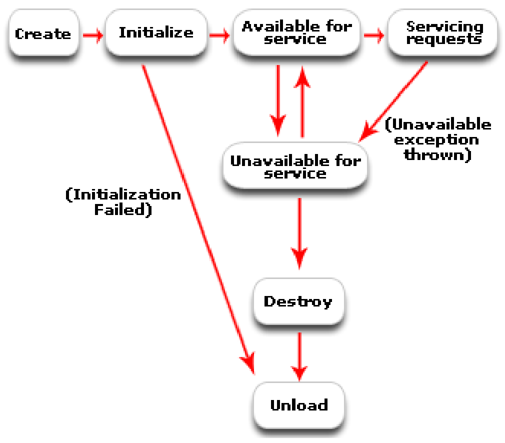
4) Expressions
These tags evaluate a Java expression to a String and then included in the output.
<%= JavaExpression %> equivalent to XML tag <jsp:expression> … </jsp:expression>

5) Action Tags (Bean <jsp:useBean>, Include <jsp:include> Forward <jsp:forward>) and

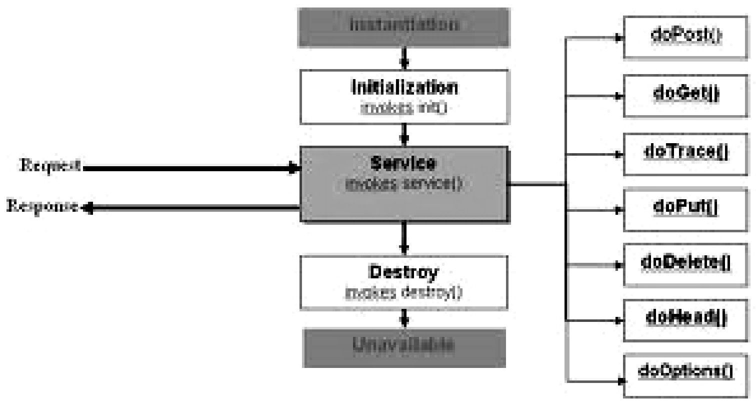
6) Comments <%-- --%>

Implicit Objects Variables

Implicit Object	Type	Purpose/Useful methods	Scope
request	Subclass of ServletRequest	The HTTP request sent to the server from the client	request
response	Subclass of ServletResponse	The HTTP response sent to the client from the server	page
out	JspWriter	Object for writing to the output stream. flush, getBufferSize	page
session	HttpSession	created as long as <% page session=”false” %> is *not* used. Valid for Http protocol only. getAttribute, setAttribute	session
config	ServletConfig	Specific to a servlet instance. Same as Servlet.getServletConfig() getInitParameter, getInitParameterNames	page
application	ServletContext	Available to all Servlets (application wide). Provides access to resources of the servlet engine (resources, attributes, context params, request dispatcher, server info, URL & MIME resources).	application
page	Object	this instance of the page (equivalent servlet instance ‘this’).	page
pageContext	PageContext	provides a context to store references to objects used by the page, encapsulates implementation-dependent features, and provides convenience methods.	page
exception	java.lang.Throwable	created only if <%@ page isErrorPage=”true” %>	page



Servlet Life Cycle



Chapter 33 – Security

J2EE Security Model

The J2EE platform architecture provides for the secure deployment of application components. It emphasizes the declarative approach wherein the application components' security structure, roles, access control, authentication and authorization requirements - as well as the other characteristics pertaining to transactions, persistence, and more - are expressed and managed outside the application code.

J2EE server products provide deployment tools that support the declarative customization of application components for the operational runtime environment. An application component provider isn't expected to implement security services, as it's the responsibility of the J2EE container and server. The security functions provided by the J2EE platform include authentication, access authorization, and secure communication with clients.

Authentication

Authentication is the process by which an entity (such as a user, organization, or program) proves and establishes its identity with the system by supplying authentication data. For users this typically means a user name and password. (In general, authentication data could be comprised of a digital certificate, or even biometric data such as a fingerprint, iris, or retina scan.) A principal is an entity that can be validated by an authentication mechanism; it's identified using a principal name and verified using authentication data.

Types of Authentication:

1. Basic
2. Form-Based
3. Digest
4. SSL and Client Certification

Web Container's Support for Authentication

The <auth-method> element in the Web deployment descriptor is used to configure the type of authentication mechanism such as "BASIC", and "FORM".

<auth-method>FORM</auth-method>

The deployment tools provided with the J2EE server product insulate us from having to manually write the deployment descriptors; the elements are provided here for reference. The following are some of the authentication mechanisms made available by the Web container and server.

Basic Authentication

HTTP basic authentication is the simplest. When a user attempts to access any protected Web resource, the Web container checks if the user has already been authenticated. If the user hasn't, the browser's built-in login screen is used to solicit the user name and password from the user so that the Web server can perform authentication. If the login fails, the browser's built-in screens and messages will be used. The user name and password are sent using simple base 64 encoding.

Form-Based Authentication

Form-based authentication is used if an application-specific login screen is required, and the browser's built-in authentication screen isn't adequate. The Web deployment descriptor needs to specify the login form page and the error page to be used with this mechanism. When the user attempts to access any protected Web resources, the Web container checks if the user has already been authenticated. If the user hasn't, the container presents the login form as specified in the deployment descriptor. If authentication fails, the error page, as specified in the deployment descriptor, is displayed.

The <form-login-page> and <form-error-page> elements are respectively used to specify the location of the login and error pages that need to be displayed. Further, the login page must contain fields named precisely j_username and j_password to represent the user name and password, respectively, as shown in Listing 1.

HTTPS Authentication

HTTPS (HTTP over SSL) authentication is a strong authentication mechanism. It requires the user to possess a public key certificate and is ideal for e-commerce applications as well as single sign-ons from within the browser in an enterprise.

Hybrid Authentication

In both the HTTP basic and the form-based authentication, passwords aren't adequately protected for confidentiality. This deficiency can be overcome by running HTTP basic and HTTP form-based authentication mechanisms over SSL. Generally, the use of the CONFIDENTIAL flag in the <transport-guarantee> element of the Web deployment descriptor ensures the use of SSL for data transmission.

Authorization and Access Control

Authorization and access control mechanisms ensure that only authenticated principals who have the required permissions to access application components are able to do so. In general, there are two fundamental approaches to controlling access - capabilities and permissions. The capabilities-based approach focuses on what resources a given user can access. The permissions-based approach on the other hand focuses on which users can access a given resource. The J2EE authorization model uses role-based permissions. A role is a logical grouping of users that's used to define a logical security view for the application. Protected application resources have associated authorization rules - roles that are allowed to access a given component are specified in the application component's deployment descriptor. The deplorer maps the roles to actual users using the J2EE server's deployment tools. The J2EE server enforces the prescribed security policies at runtime and ensures that only those users who belong to the appropriate role are able to access the protected components' functionality - while those who don't belong are denied access.

J2EE Containers

A container is part of the J2EE server. It provides deployment and runtime support for application components and is responsible for infrastructural services including security. There are three types of J2EE containers that are meant to house different J2EE application components:

EJB container: Houses EJB components
Web container: Houses servlets, JSPs, static HTML, JPEG files, and more
J2EE application client container: Houses J2EE application clients

Each of these containers has its associated deployment descriptor.

Obtaining the Initiating Security Context

In the J2EE model, secure applications require that the client programs be authenticated. An end user can be authenticated using either a Web or an application client. Once the user is authenticated, an initial security context is generated and maintained by the J2EE platform. This security context is the encapsulation of the identity of the authenticated user principal.

Chapter 34 – SOAP and REST

Simple Object Access Protocol (SOAP)

SOAP is a simple XML-based protocol designed to let applications exchange information over HTTP. It is a protocol for accessing the Web Services.

RPC (DCOM or CORBA) is used to communicate between distributed applications. But, it possesses compatibility and security problems. It gets blocked by firewalls and proxy servers. Better way is HTTP because it is supported by all browsers and servers. But, it wasn't designed for this purpose. So, SOAP was created to accomplish this.

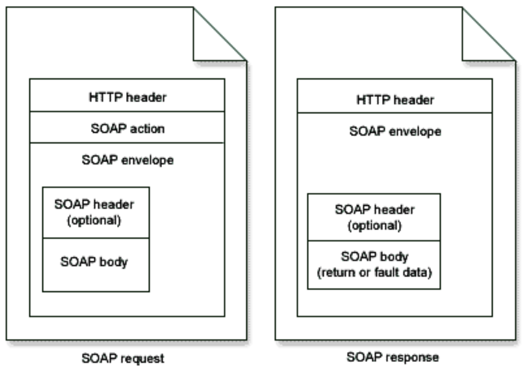
SOAP provides a way to communicate between **applications** running on **different operating systems**, with **different technologies** and **programming languages**.

A SOAP message consists of the following four parts:

ENVELOPE - it is a mandatory part defining the beginning and end of the message.
HEADER - It is optional and if included, may contain any optional properties (attributes) that were included in the building of the message and which may later be required in processing the message.
BODY - It is a mandatory part and contains the XML containing the message to be sent.
FAULT - Again, this is an optional component, providing additional information about errors that occurred whilst processing the message.

SOAP Data Types

1. Scalar types: Contains exactly one value, e.g., a last name, price.
2. Compound types: Contain multiple values, such as a purchase order



For message transmission, SOAP makes use of an 'internet application layer' protocol as a 'message transfer protocol'. Both SMTP and HTTP are widely used as these application layer protocols with HTTP being the more preferred one, as it works better with network firewalls.

```
<soap:Envelope xmlns:soap soap:encodingStyle>
   <soap:Header>
      <m:Trans xmlns:m soap:mustUnderstand=0|1/>
    </soap:Header>
   <soap:Body>
       <soap:Fault>
          <faultcode>
          <faultstring>
          <faultactor>
          <detail>
       </soap:Fault>
   <soap:Body>
</soap:Envelope>
```

SOAP Building Blocks

- an Envelope element that identifies the XML document as a SOAP message
- a Header element that contains header information (authentication,payment, and so on)
- a Body element that contains call and response information
- a Fault element containing error and status information

Syntax Rules of a SOAP Message

1. Must be encoded using XML
2. Must use SOAP Envelope namespace
3. Must use SOAP Encoding namespace
4. Must not contain a DTD reference
5. Must not contain XML Processing Instructions

SOAP Fault Codes

1. VersionMismatch
2. MustUnderstand
3. Client
4. Server

SOAP HTTP Binding

HTTP + XML = SOAP
Content-Type: application/soap+XML

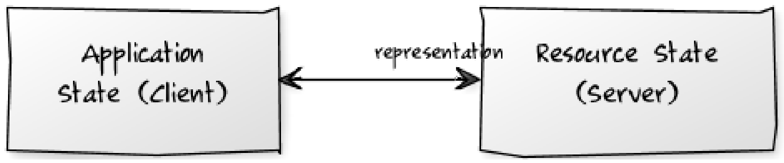
Representational State Transfer (REST)

RESTFul web services are based on the HTTP methods and concept of REST. REST follows one-to-one mapping between CRUD (Create, Read, Update & Delete) operations and HTTP methods.

- To create a resource on the server, use POST
- To retrieve a resource, use GET
- To change the state of a resource or to update it, use PUT
- To remove or delete a resource, use DELETE

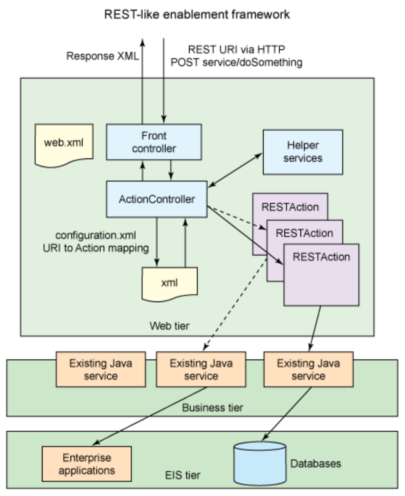
REpresentational State Transfer (REST) is an architecture principle in which the web services are viewed as resources and can be uniquely identified by their URLs.

Representational State Transfer refers to transferring "representations". You are using a "representation" of a resource to transfer resource state which lives on the server into application state on the client.



The most important concept in REST is resources, which are identified by unique global IDs— typically using URIs. Client applications use HTTP methods (GET/ POST/ PUT/ DELETE) to manipulate the resource or collection of resources. A RESTful Web service is a web service implemented using HTTP and the principles of REST. Typically, a RESTful Web service should define the following aspects:

- The base/root URI for the Web service such as http://host/<appcontext>/resources.
- The MIME type of the response data supported, which are JSON/XML/ATOM and so on.
- The set of operations supported by the service. (for example, POST, GET, PUT or DELETE).



The key characteristic of a RESTful Web service is the explicit use of HTTP methods to denote the invocation of different operations.

The basic REST design principle uses the HTTP protocol methods for typical CRUD operations:

- POST - Create a resource
- GET - Retrieve a resource
- PUT – Update a resource
- DELETE - Delete a resource

The major advantages of REST-services are:

- They are **highly reusable across platforms** (Java, .NET, PHP, etc) since they rely on basic **HTTP** They use **basic XML** instead of the complex SOAP XML and are easily consumable

REST-based web services are increasingly being preferred for integration with backend enterprise services. In comparison to SOAP based web services, the programming model is simpler and the use of native XML instead of SOAP reduces the serialization and deserialization complexity as well as the need for additional third-party libraries for the same.

Features of REST:

- Light-weight
- No WSDL file
- Only XML file and no SOAP envelope or header that reduces the processing overhead
- Supports only HTTP and not FTP or SMTP
- Supports MIME types such as text/XML, text/JSON, and so on
- If enabled, then both REST and SOAP endpoints are enabled

Chapter 35 – JAXB and JAXP

Java Architecture for XML Binding (JAXB)

Java Architecture for XML Binding (JAXB) is a Java standard that defines how Java objects are converted from and to XML. It uses a standard set of mappings. JAXB defines an API for reading and writing Java objects to and from XML documents.

Key Annotations

Annotation	Description
@XmlSchema	Maps a package name to a XML namespace.
@XmlRootElement (namespace = "namespace")	Defines the root element for an XML tree. It maps a class or an enum type to an XML element.
@XmlAccessorOrder	Controls the ordering of fields and properties in a class.
@XmlType	Maps a class or an enum type to a XML Schema type.
@XmlType(propOrder = { "field2", "field1",.. })	Allows defining the order in which the fields are written in the XML file.
@XmlElement(name = "neuName")	Maps a JavaBean property to a XML element derived from property name.
@XmlElementWrapper(name = "bookList")	Generates a wrapper element around XML representation.
@XmlElements	A container for multiple @XmlElement annotations.
@XmlAttribute	Maps a JavaBean property to a XML attribute.
@XmlAttachmentRef	Marks a field/property that its XML form is an URI reference to mime content.
@XmlMimeType	Associates the MIME type that controls the XML representation of the property.
@XmlEnum	Maps an enum type Enum to XML representation.
@XmlEnumValue	Maps an enum constant in Enum type to XML representation.

Code Snippet that maps the Java Bean to XML document

```
@XmlRootElement(name = "book")
// If you want you can define the order in which the fields are written (Optional)
@XmlType(propOrder = { "author", "name", "publisher", "isbn" })
public class BookStore {

 private String storeName;

 // XmLElementWrapper generates a wrapper element around XML representation
 @XmlElementWrapper(name = "bookList")
 // XmlElement sets the name of the entities
 @XmlElement(name = "book")
 private ArrayList<Book> bookList;

 // If you like the variable name, e.g. "name", you can easily change this name for your XML-Output:
 @XmlElement(name = "title")
 public String getStoreName() {
  return storeName;
 }

}
```

Code Snippet that converts the Java Object from and to XML document

```
private static final String BOOKSTORE_XML = "./bookstore-jaxb.xml";

 try{

  // Create JAXB context and load the book store object
  JAXBContext context = JAXBContext.newInstance(Bookstore.class);

  // Instantiate Marshaller and Write to XML file
  Marshaller m = context.createMarshaller();
  m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
  m.marshal(bookstore1, new File(BOOKSTORE_XML));

  // Instantiate Unmarshaller and Read from XML file
  Unmarshaller um = context.createUnmarshaller();
  Bookstore bookstore2 = (Bookstore) um.unmarshal(new FileReader(BOOKSTORE_XML));

} catch (JAXBException jaxbe, IOException ioe) {}
```

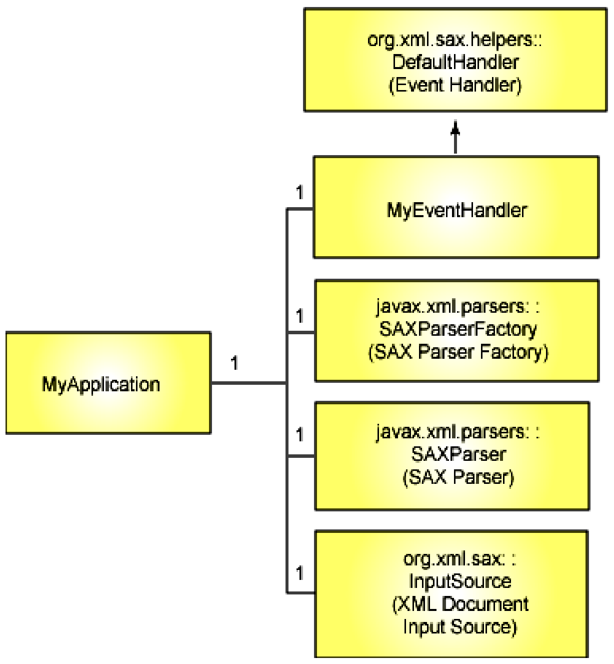
Types of XML Parsers

1. SAX (Simple API for XML)
2. DOM (Document Object Model)

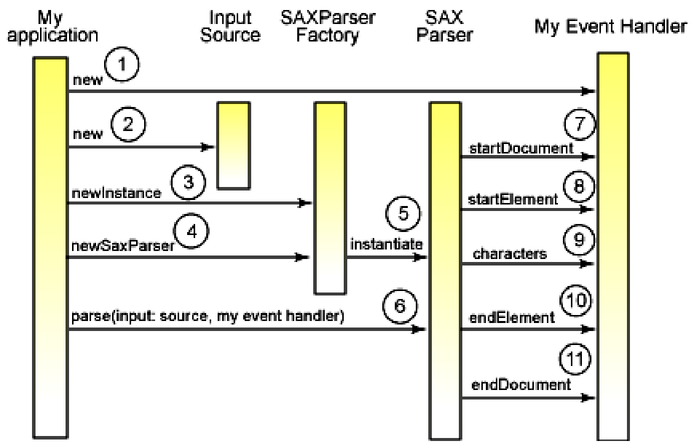
Simple API for XML (SAX)

1. Event based model
2. Serial access (flow of events)
3. Low memory usage
4. To process parts of the document
5. To process the document only once.

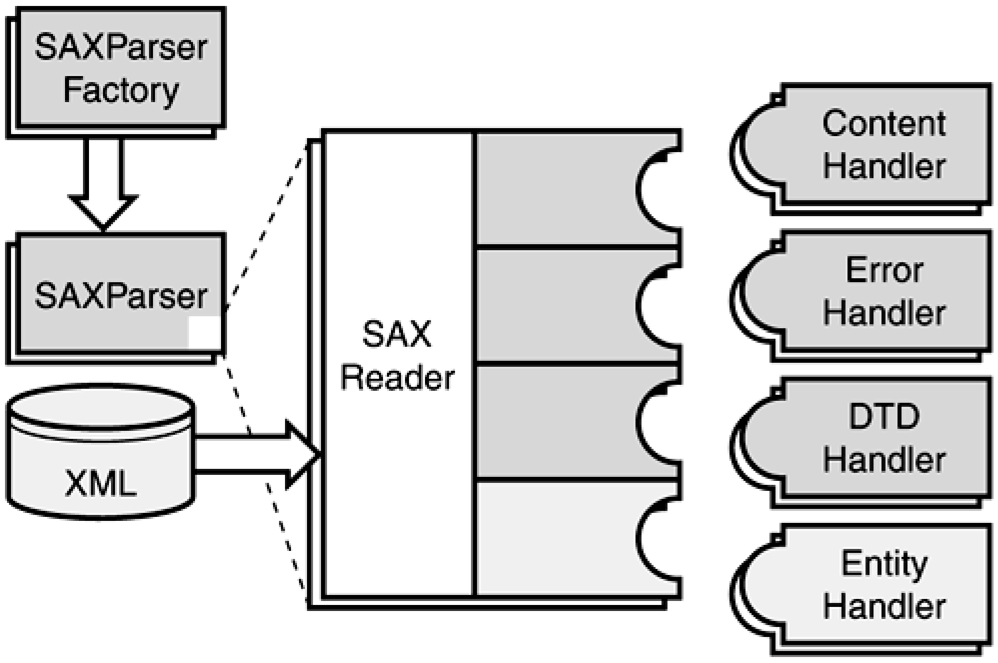
Typical elements of a JAXP application



JAXP Sequence Diagram



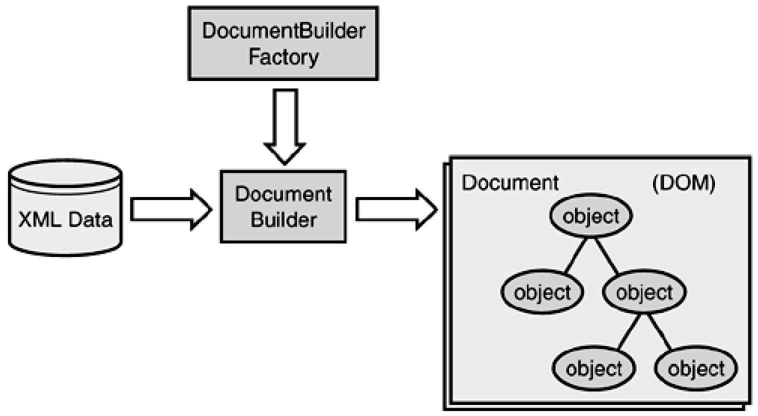
SAX parsing architecture



Document Object Model (DOM)

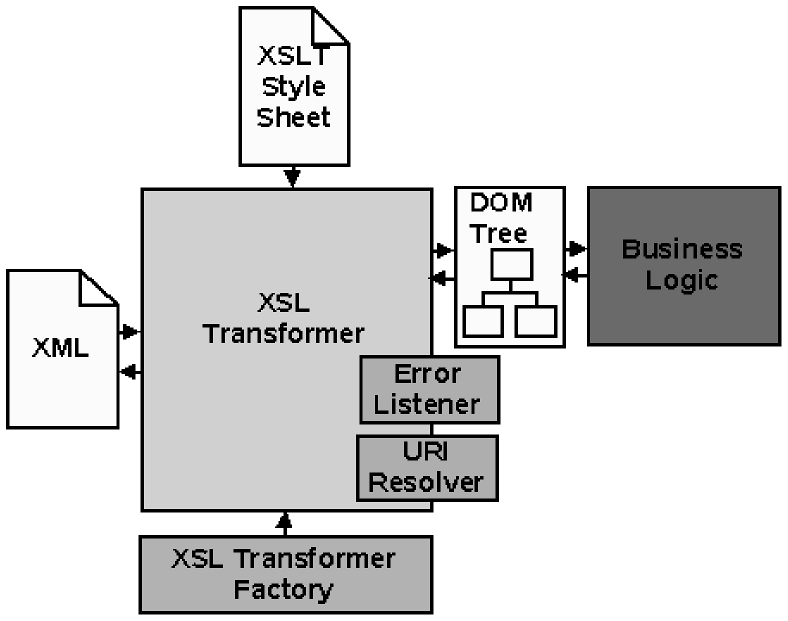
1. Tree data structure
2. Random access
3. High memory usage
4. To edit the document
5. To process the document multiple times

DOM parsing architecture



XSLT - XML (Extensible Markup Language) Stylesheet Language for Transformations

1. Based on SAX or DOM
2. Xpath patterns
3. XSL statements



Chapter 36 – Web Service

A web service is a method of communication between two electronic devices over the web (internet).

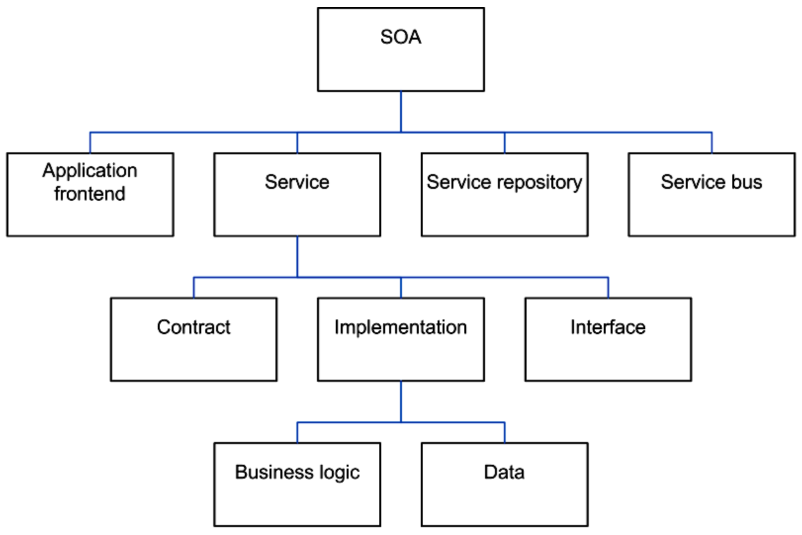
Web services are a set of tools that can be used in a number of ways. There are three most common styles of use. They are:

- Remote procedure calls (RPC) – Java API for XML based RPC (JAX-RPC)
- Service-Oriented Architecture (SOA) – Java API for XML based Web Services (JAX-WS)
- Representational State Transfer (REST) – Java API for Restful Web Services (JAX-RS)

Service Oriented Architecture (SOA)

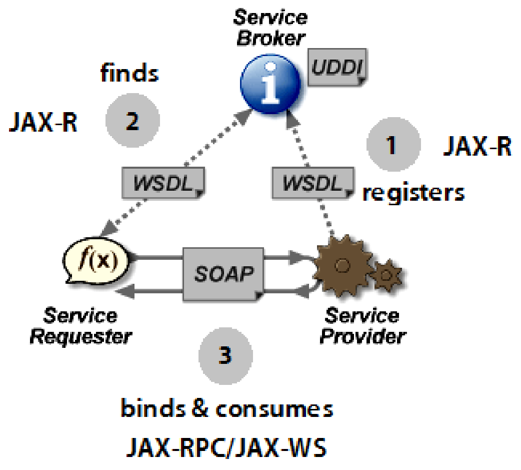
In software engineering, a Service-Oriented Architecture (SOA) is a set of principles and methodologies for designing and developing software in the form of interoperable services.

Elements of Service-Oriented Architecture



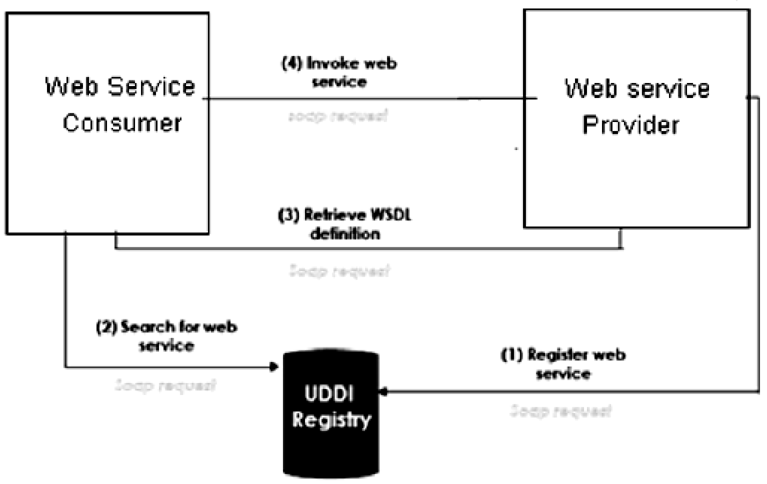
Three critical roles in SOA:

1. Service Provider
2. Service Broker
3. Service Consusmer or Requester



General Flow:

1. Service Provider **registers** the service contract (WSDL) with Service Broker (UDDI registry) using JAX-R
2. Service Consumer **finds** the service contract (WSDL) registered with Service Broker (UDDI registry) using JAX-R
3. Service Consumer or Requester (Client) **binds** and **consumes** the service provided by Service Provider using SOAP/SAAJ protocol and JAX-RPC/JAX-WS APIs



Glossary

Term	Description
WSDD	Web Service Deployment Descriptor (services.xml)
SAAJ	SOAP with Attachments API for Java
UDDI	Universal Description, Discovery and Integration
WSDL	Web Service Description Language (for SOAP based web services)
SOAP	Simple Object Access Protocol
REST	Representational State Transfer
JAX-RPC	Java API for XML-based Remote Procedure Call
JAX-WS	Java API for XML-based Web Services
JAX-RS	Java API for RESTful web Services
JAX-R	Java API for XML-based Registry
JAXB	Java Architecture for XML Binding
WS-S	Web Services - Security
MTOM	Message Transmission Optimization Mechanism
WS-I BP	Web Services-Interoperability (Basic Profile)
WADL	Web Application Description Language (for RESTful web services)

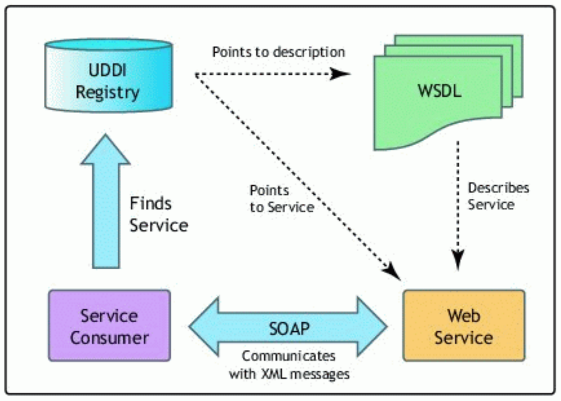
Web Service Deployment Descriptor (WSDD)

WSDD (services.xml) contains

- service provider information (ex: Java: RPC)
- class name
- allowable method names
- scope

Universal Description, Discovery and Integration (UDDI)

UDDI is a platform-independent, XML-based registry and directory service where businesses can register and search for web services.



Web Service Approaches:

1. Top Down Development (Contract First)
2. Bottom Up Development (Code First)

Top Down Development

Top-down web service development involves creating a web service from a WSDL file. By creating the WSDL file first you will ultimately have more control over the web service.

In top-down approach, the WSDL is **designed from a business point of view** and is not driven by a service consumer or an existing application. The “WSDL to Java” tool generates java code according to JAX-RPC or JAX-WS. These classes can then adapt the existing business logic.

Bottom Up Development

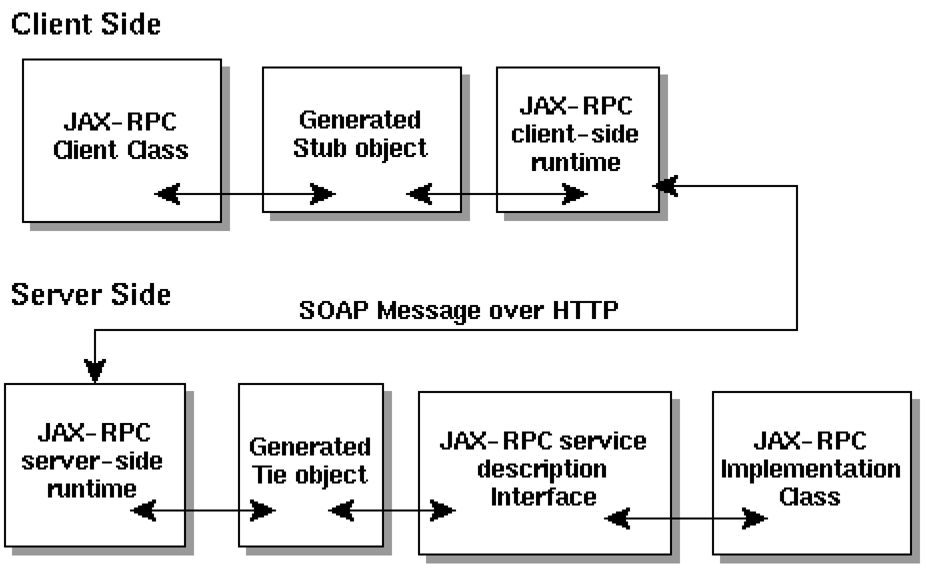
In bottom-up approach, first you create a POJO or EJB and then use the web services wizard to create the WSDL file and Web service.

Although bottom-up web service development may be faster and easier, especially if you are new to Web services, the top-down approach is the recommended way of creating a Web service. The bottom up approach generates web services for which the design is driven from an **application point of view** (developer driven) and will not address the need for other consumers.

Java API for XML based RPC (JAX-RPC)

It allows a Java application to invoke Java-based Web Services with a known description while still being consistent with its WSDL description. It's an API for building Web Services and clients that used RPC and XML.

JAX-RPC Flow Diagram


Client Side
JAX-RPC Client Class
Generated Stub object
JAX-RPC client-side runtime
Server Side
SOAP Message over HTTP
JAX-RPC server-side runtime
Generated Tie object
JAX-RPC service description Interface
JAX-RPC Implementation Class

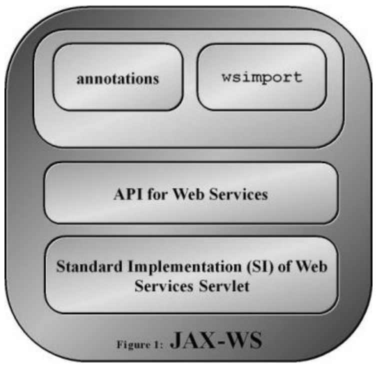
It works as follows:

1. A Java program invokes a method on a stub
2. The stub invokes routines in the JAX-RPC Client-side Runtime System (RS)
3. The RS converts the remote method invocation into a SOAP message
4. The RS transmits the message as an HTTP request
5. The JAX-RPC Server-side Runtime System (RS) receives the SOAP message and converts the request into server method call
6. The RS invokes the appropriate method on a tie object
7. The tie object invokes business method in the service implementation class through the service description interface

Thus, Servlet, EJB, JB or POJO are made available through Web Services.

Java API for XML based Web Services (JAX-WS)

JAX-RPC 2.0 was renamed JAX-WS 2.0 for Message-oriented Web Services and also supports SOAP 1.1 & 1.2 with WS-I BP 1.1, JAXB2.0 for mapping, MTOM and Java 5.0.



Java API for RESTful web Services (JAX-RS)

JAX-RS uses annotations to define the REST relevance of Java classes. The package name is **javax.ws.rs.**

Annotation	Description	Usage

@GET	Indicates that the following method will answer to an HTTP GET/Retrieve request.	**@GET**
		public String getHTML() {
		  ...
		}

@POST	Indicates that the following method will answer to an HTTP POST/Create request.	**@POST**
		@Consumes("application/json")
		@Produces("application/json")
		public RestResponse<Contact> create(Contact contact) {
		...
		}

@PUT	Indicates that the following method will answer to an HTTP PUT/Update request.	**@PUT**
		@Consumes("application/json")
		@Produces("application/json")
		@Path("{contactId}")
		public RestResponse<Contact> update(Contact contact) {
		...
		}

@DELETE	Indicates that the following method will answer to an HTTP DELETE/Remove request.	**@DELETE**
		@Produces("application/json")
		@Path("{contactId}")
		public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {
		...
		}

@Consumes (MediaType.APPLICATION_XML[, more-types])	'Consumes' defines which MIME type is consumed by a method annotated with @POST, @PUT or @DELETE.	@PUT
		@Consumes("application/json")
		@Produces("application/json")
		@Path("{contactId}")
		public void delete(Contact contact) {
		...
		}

@Produces (MediaType.TEXT_PLAIN[, more-types])	'Produces' defines which MIME type is delivered or prodcued by a method annotated with @GET. The standard outputs are "text/plain", "text/html", "application/xml" and "application/json".	@GET
		@Produces("application/json")
		public Contact getJSON() {
		  ...
		}

@Path(your_path)	Specifies the URL path on which this method will be invoked. The base URL is based on your application name, the servlet and the URL pattern from the web.xml configuration file.	@GET
		@Produces("application/xml")
		@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
		public Contact getXML() {
		  ...
		}

@PathParam	Used to inject values from the URL into a method parameter. We can bind REST-style URL parameters to method arguments using @PathParam annotation.	@GET
		@Produces("application/xml")
		@Path("xml/{firstName}")
		public Contact getXML(**@PathParam("firstName") String firstName**) {
		  Contact contact = contactService.findByFirstName(firstName);
		  return contact;
		}

@QueryParam	Request parameters in query string can be accessed using @QueryParam annotation.	@GET
		@Produces("application/json")
		@Path("json/companyList")
		public CompanyList
		getJSON(**@QueryParam("start") int start**,
		@QueryParam("limit") int limit) {
		  CompanyList list = new
		CompanyList(companyService.listCompanies(start,
		limit));
		  return list;
		}

@FormParam	The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly using @FormParam.	@POST
		public String save(**@FormParam("firstName")
		String firstName**,
		    @FormParam("lastName") String lastName) {
		      ...
		 }

@HeaderParam	Binds the value(s) of a HTTP header to a resource method parameter, resource class field, or resource class bean property.	public class MyBeanParam {
		    @HeaderParam("header")
		    private String headerParam;
		}

@CookieParam	Binds the value of a HTTP cookie to a resource method parameter, resource class field, or resource class bean property.	public class MyBeanParam {
		    @CookieParam("cookieName")
		    private String cookieParam;
		}

@MatrixParam	Binds the value(s) of a URI matrix parameter to a resource method parameter, resource class field, or resource class bean property.	public class MyBeanParam {
		    @MatrixParam("m")
		    @Encoded
		    @DefaultValue("default")
		    private String matrixParam;
		}

@BeanParam	It can contain all parameter injections. The JAX-RS runtime will instantiate the object and inject all its fields and properties annotated with either one of the @XxxParam annotation or the @Context annotation.	public class MyBean {
		   @FormParam("myData")
		   private String data;
		   @HeaderParam("myHeader")
		   private String header;
		   @PathParam("id")
		   public void setResourceId(String id) {...}
		 }
		
		 @Path("myresources")
		 public class MyResources {
		   @POST
		   @Path("{id}")
		   public void post(**@BeanParam MyBean
		myBean**) {...}
		 }

@DefaultValue	Defines the default value of request meta-data that is bound using either one of PathParam, QueryParam, MatrixParam, CookieParam, FormParam or HeaderParam.	@Path("{id:\\d+}")
		public class InjectedResource {
		    // Injection onto field
		    @DefaultValue("q") @QueryParam("p")
		    private String p;
		}

@Encoded	Disables automatic decoding of parameter values bound using QueryParam, PathParam, FormParam or MatrixParam.	public class MyBeanParam {
		    @MatrixParam("m")
		    @Encoded
		    @DefaultValue("default")
		    private String matrixParam;
		}

@Context	This annotation is used to inject information into a class field, bean property or method parameter. When deploying a	@GET
		public String get(**@Context UriInfo ui**) {
		    MultivaluedMap<String, String> queryParams =
		ui.getQueryParameters();

	JAX-RS application using servlet then ServletConfig, ServletContext, HttpServletRequest and HttpServletResponse are available using @Context.	MultivaluedMap<String, String> pathParams = ui.getPathParameters();
		}
		
		@GET
		public String get**(@Context HttpHeaders hh**) {
		MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
		Map<String, Cookie> pathParams = hh.getCookies();
		}
		
		@Context
		public void setRequest(**Request request**) {
		String[] requestParams = request.getRequestParams();
		}
		
		public MySingletonResource**(@Context SecurityContext securityContext**) {
		...
		}

@Singleton	In this scope there is only one instance per jax-rs application.	**@Singleton**
		@Path("/printers")
		public class PrintersResource {
		...
		}

@RequestScoped	Default lifecycle (applied when no annotation is present). In this scope the resource instance is created for each new request and used for processing of this request.	**@RequestScoped**
		@Path("/pathUrl")
		public class UserAction {
		...
		}

@PerLookup	In this scope the resource instance is created every time it is needed for the processing even it handles the same request.	**@PerLookup**
		@Path("/pathUrl")
		public class GranularTransaction {
		...
		}

Chapter 37 – Java Script

What is Java Script?

JavaScript is a lightweight programming language and it is designed to add interactivity to HTML pages and usually embedded directly into HTML pages. It is an interpreted language (means that scripts execute without preliminary compilation).

- **JavaScript gives HTML designers a programming tool** - HTML authors are normally not programmers, but JavaScript is a scripting language with a very simple syntax! Almost anyone can put small "snippets" of code into their HTML pages
- **JavaScript can react to events** - A JavaScript can be set to execute when something happens, like when a page has finished loading or when a user clicks on an HTML element
- **JavaScript can read and write HTML elements** - A JavaScript can read and change the content of an HTML element
- **JavaScript can be used to validate data** - A JavaScript can be used to validate form data before it is submitted to a server. This saves the server from extra processing
- **JavaScript can be used to detect the visitor's browser** - A JavaScript can be used to detect the visitor's browser, and - depending on the browser - load another page specifically designed for that browser
- **JavaScript can be used to create cookies** - A JavaScript can be used to store and retrieve information on the visitor's computer

Key Notes

- Unlike HTML, JavaScript is case sensitive - therefore watch your capitalization closely when you write JavaScript statements, create or call variables, objects and functions.

- The HTML <script> tag is used to insert a JavaScript into an HTML page.

- Try to avoid using document.write() in real life JavaScript code. The entire HTML page will be overwritten if document.write() is used inside a function, or after the page is loaded. However, document.write() is an easy way to demonstrate JavaScript output in a tutorial.

- To manipulate HTML elements JavaScript uses the DOM method getElementById(). This method accesses the element with the specified id.
- External script cannot contain the <script></script> tags
- Using semicolons makes it possible to write multiple statements on one line.
- Comments can be added to explain the JavaScript, or to make the code more readable. Single line comments start with //. Multi line comments start with /* and end with */.
- Variable names are case sensitive (y and Y are two different variables)
- Variable names must begin with a letter, the $ character, or the underscore character
- Because JavaScript is case-sensitive, variable names are case-sensitive.
- When you assign a text value to a variable, put quotes around the value.
- If you re-declare a JavaScript variable, it will not lose its value.
- = is used to assign values.
- + is used to add values.
- JavaScript has three kinds of popup boxes: Alert box, Confirm box, and Prompt box.
- An alert box is often used if you want to make sure information comes through to the user.
- A confirm box is often used if you want the user to verify or accept something.
- A prompt box is often used if you want the user to input a value before entering a page.
- A function will be executed by an event or by a call to the function.
- Loops execute a block of code a specified number of times, or while a specified condition is true.
- Events are actions that can be detected by JavaScript.
- Events are normally used in combination with functions, and the function will not be executed before the event occurs
- The try...catch statement allows you to test a block of code for errors.
- The throw statement allows you to create an exception.
- In JavaScript you can add special characters to a text string by using the backslash sign.
- JavaScript ignores extra spaces.
- You can break up a code line within a text string with a backslash.

Chapter 38 – jQuery

jQuery is a **cross-platform, cross-browser JavaScript library** designed to simplify the client-side scripting of HTML. This page lists down all the jQuery APIs at one place for your easy access.

jQuery – Selectors

Selector	Description
Name	Selects all elements which match with the given **elementName**.
#ID	Selects a single element which matches with the given **ID**
.Class	Selects all elements which match with the given **Class**.
Universal (*)	Selects all elements available in a DOM.
Multiple Elements B, C	Selects the combined results of all the specified selectors **B or C**.

Similar to above syntax and examples, following examples would give you understanding on using different type of other useful selectors:

- **$('*'):** This selector selects all elements in the document.
- **$("p > *"):** This selector selects all elements that are children of a paragraph element.
- **$("#specialID"):** This selector function gets the element with id="specialID".
- **$(".specialClass"):** This selector gets all the elements that have the class of*specialClass*.
- **$("li:not(.myclass)"):** Selects all elements matched by <li> that do not have class="myclass".
- **$("a#specialID.specialClass"):** This selector matches links with an id of *specialID* and a class of *specialClass*.
- **$("p a.specialClass"):** This selector matches links with a class of *specialClass* declared within <p> elements.
- **$("ul li:first"):** This selector gets only the first <li> element of the <ul>.
- **$("#container p"):** Selects all elements matched by <p> that are descendants of an element that has an id of *container*.
- **$("li > ul"):** Selects all elements matched by <ul> that are children of an element matched by <li>
- **$("strong + em"):** Selects all elements matched by <em> that immediately follow a sibling element matched by <strong>.
- **$("p ~ ul"):** Selects all elements matched by <ul> that follow a sibling element matched by <p>.

- **$("code, em, strong"):** Selects all elements matched by <code> or <em> or <strong>.
- **$("p strong, .myclass"):** Selects all elements matched by <strong> that are descendants of an element matched by <p> as well as all elements that have a class of*myclass*.
- **$(":empty"):** Selects all elements that have no children.
- **$("p:empty"):** Selects all elements matched by <p> that have no children.
- **$("div[p]"):** Selects all elements matched by <div> that contain an element matched by <p>.
- **$("p[.myclass]"):** Selects all elements matched by <p> that contain an element with a class of *myclass*.
- **$("a[@rel]"):** Selects all elements matched by <a> that have a rel attribute.
- **$("input[@name=myname]"):** Selects all elements matched by <input> that have a name value exactly equal to *myname.*
- **$("input[@name^=myname]"):** Selects all elements matched by <input> that have a name value beginning with *myname*.
- **$("a[@rel$=self]"):** Selects all elements matched by <p> that have a class value ending with *bar*
- **$("a[@href*=domain.com]"):** Selects all elements matched by <a> that have an href value containing domain.com.
- **$("li:even"):** Selects all elements matched by <li> that have an even index value.
- **$("tr:odd"):** Selects all elements matched by <tr> that have an odd index value.
- **$("li:first"):** Selects the first <li> element.
- **$("li:last"):** Selects the last <li> element.
- **$("li:visible"):** Selects all elements matched by <li> that are visible.
- **$("li:hidden"):** Selects all elements matched by <li> that are hidden.
- **$(":radio"):** Selects all radio buttons in the form.
- **$(":checked"):** Selects all checked boxex in the form.
- **$(":input"):** Selects only form elements (input, select, textarea, button).
- **$(":text"):** Selects only text elements (input[type=text]).
- **$("li:eq(2)"):** Selects the third <li> element
- **$("li:eq(4)"):** Selects the fifth <li> element
- **$("li:lt(2)"):** Selects all elements matched by <li> element before the third one; in other words, the first two <li> elements.
- **$("p:lt(3)"):** selects all elements matched by <p> elements before the fourth one; in other words the first three <p> elements.
- **$("li:gt(1)"):** Selects all elements matched by <li> after the second one.
- **$("p:gt(2)"):** Selects all elements matched by <p> after the third one.
- **$("div/p"):** Selects all elements matched by <p> that are children of an element matched by <div>.
- **$("div//code"):** Selects all elements matched by <code>that are descendants of an element matched by <div>.
- **$("//p//a"):** Selects all elements matched by <a> that are descendants of an element matched by <p>
- **$("li:first-child"):** Selects all elements matched by <li> that are the first child of their parent.
- **$("li:last-child"):** Selects all elements matched by <li> that are the last child of their parent.
- **$(":parent"):** Selects all elements that are the parent of another element, including text.
- **$("li:contains(second)"):** Selects all elements matched by <li> that contain the text second.

jQuery - DOM Attributes

Methods	Description
attr(properties)	Set a key/value object as properties to all matched elements.
attr(key, fn)	Set a single property to a computed value, on all matched elements.
removeAttr(name)	Remove an attribute from each of the matched elements.
hasClass(class)	Returns true if the specified class is present on at least one of the set of matched elements.
removeClass(class)	Removes all or the specified class(es) from the set of matched elements.
toggleClass(class)	Adds the specified class if it is not present, removes the specified class if it is present.
html( )	Get the html contents (innerHTML) of the first matched element.
html(val)	Set the html contents of every matched element.
text( )	Get the combined text contents of all matched elements.
text(val)	Set the text contents of all matched elements.
val( )	Get the input value of the first matched element.
val(val)	Set the value attribute of every matched element if it is called on <input> but if it is called on <select> with the passed <option> value then passed option would be selected, if it is called on check box or radio box then all the matching check box and radiobox would be checked.

Similar to above syntax and examples, following examples would give you understanding on using various attribute methods in different situation:

- **$("#myID").attr("custom") :** This would return value of attribute *custom* for the first element matching with ID myID.
- **$("img").attr("alt", "Sample Image"):** This sets the **alt** attribute of all the images to a new value "Sample Image".
- **$("input").attr({ value: "", title: "Please enter a value" }); :** Sets the value of all <input> elements to the empty string, as well as sets the title to the string *Please enter a value*.
- **$("a[href^=http://]").attr("target","_blank"):** Selects all links with an href attribute starting with *http://* and set its target attribute to *_blank*
- **$("a").removeAttr("target") :** This would remove *target* attribute of all the links.
- **$("form").submit(function() {$(":submit",this).attr("disabled", "disabled");}); :** This would modify the disabled attribute to the value "disabled" while clicking Submit button.
- **$("p:last").hasClass("selected"):** This return true if last <p> tag has associated class*selected*.
- **$("p").text():** Returns string that contains the combined text contents of all matched <p> elements.
- **$("p").text("<i>Hello World</i>"):** This would set "<I>Hello World</I>" as text content of the matching <p> elements
- **$("p").html() :** This returns the HTML content of the all matching paragraphs.

- **$("div").html("Hello World") :** This would set the HTML content of all matching <div> to *Hello World*.
- **$("input:checkbox:checked").val() :** Get the first value from a checked checkbox
- **$("input:radio[name=bar]:checked").val():** Get the first value from a set of radio buttons
- **$("button").val("Hello") :** Sets the value attribute of every matched element <button>.
- **$("input").val("on") :** This would check all the radio or check box button whose value is "on".
- **$("select").val("Orange") :** This would select Orange option in a dropdown box with options Orange, Mango and Banana.
- **$("select").val("Orange", "Mango") :** This would select Orange and Mango options in a dropdown box with options Orange, Mango and Banana.

jQuery - DOM Traversing

Selector	Description
eq(index)	Reduce the set of matched elements to a single element.
filter(selector)	Removes all elements from the set of matched elements that do not match the specified selector(s).
filter(fn)	Removes all elements from the set of matched elements that do not match the specified function.
is(selector)	Checks the current selection against an expression and returns true, if at least one element of the selection fits the given selector.
map(callback)	Translate a set of elements in the jQuery object into another set of values in a jQuery array (which may, or may not contain elements).
not(selector)	Removes elements matching the specified selector from the set of matched elements.
slice(start, [end])	Selects a subset of the matched elements.
add(selector)	Adds more elements, matched by the given selector, to the set of matched elements.
andSelf( )	Add the previous selection to the current selection.
children([selector])	Get a set of elements containing all of the unique immediate children of each of the matched set of elements.
closest(selector)	Get a set of elements containing the closest parent element that matches the specified selector, the starting element included.
contents( )	Find all the child nodes inside the matched elements (including text nodes), or the content document, if the element is an iframe.
end( )	Revert the most recent 'destructive' operation, changing the set of matched elements to its previous state .
find(selector)	Searches for descendent elements that match the specified selectors.
next([selector])	Get a set of elements containing the unique next siblings of each of the given set of elements.
nextAll([selector])	Find all sibling elements after the current element.
offsetParent( )	Returns a jQuery collection with the positioned parent of the first

	matched element.
parent([selector])	Get the direct parent of an element. If called on a set of elements, parent returns a set of their unique direct parent elements.
parents([selector])	Get a set of elements containing the unique ancestors of the matched set of elements (except for the root element).
prev([selector])	Get a set of elements containing the unique previous siblings of each of the matched set of elements.
prevAll([selector])	Find all sibling elements in front of the current element.
siblings([selector])	Get a set of elements containing all of the unique siblings of each of the matched set of elements.

jQuery - CSS Methods

Method	Description
css(name)	Return a style property on the first matched element.
css(name, value)	Set a single style property to a value on all matched elements.
css(properties)	Set a key/value object as style properties to all matched elements.
height(val)	Set the CSS height of every matched element.
height( )	Get the current computed, pixel, height of the first matched element.
innerHeight( )	Gets the inner height (excludes the border and includes the padding) for the first matched element.
innerWidth( )	Gets the inner width (excludes the border and includes the padding) for the first matched element.
offset( )	Get the current offset of the first matched element, in pixels, relative to the document
offsetParent( )	Returns a jQuery collection with the positioned parent of the first matched element.
outerHeight([margin])	Gets the outer height (includes the border and padding by default) for the first matched element.
outerWidth([margin])	Get the outer width (includes the border and padding by default) for the first matched element.
position( )	Gets the top and left position of an element relative to its offset parent.
scrollLeft(val)	When a value is passed in, the scroll left offset is set to that value on all matched elements.
scrollLeft( )	Gets the scroll left offset of the first matched element.
scrollTop(val)	When a value is passed in, the scroll top offset is set to that value on all matched elements.

scrollTop( )	Gets the scroll top offset of the first matched element.
width(val)	Set the CSS width of every matched element.
width( )	Get the current computed, pixel, width of the first matched element.

jQuery - DOM Manipulation Methods

Method	Description
after(content)	Insert content after each of the matched elements.
append(content)	Append content to the inside of every matched element.
appendTo(selector)	Append all of the matched elements to another, specified, set of elements.
before(content)	Insert content before each of the matched elements.
clone(bool)	Clone matched DOM Elements, and all their event handlers, and select the clones.
clone( )	Clone matched DOM Elements and select the clones.
empty( )	Remove all child nodes from the set of matched elements.
html(val)	Set the html contents of every matched element.
html( )	Get the html contents (innerHTML) of the first matched element.
insertAfter(selector)	Insert all of the matched elements after another, specified, set of elements.
insertBefore(selector)	Insert all of the matched elements before another, specified, set of elements.
prepend(content)	Prepend content to the inside of every matched element.
prependTo(selector)	Prepend all of the matched elements to another, specified, set of elements.
remove(expr)	Removes all matched elements from the DOM.
replaceAll(selector)	Replaces the elements matched by the specified selector with the matched elements.
replaceWith(content)	Replaces all matched elements with the specified HTML or DOM elements.
text(val)	Set the text contents of all matched elements.
text( )	Get the combined text contents of all matched elements.
wrap(elem)	Wrap each matched element with the specified element.
wrap(html)	Wrap each matched element with the specified HTML content.
wrapAll(elem)	Wrap all the elements in the matched set into a single wrapper element.

wrapAll(html)	Wrap all the elements in the matched set into a single wrapper element.
wrapInner(elem)	Wrap the inner child contents of each matched element (including text nodes) with a DOM element.
wrapInner(html)	Wrap the inner child contents of each matched element (including text nodes) with an HTML structure.

jQuery - Events Handling

Event Type	Description
blur	Occurs when the element loses focus
change	Occurs when the element changes
click	Occurs when a mouse click
dblclick	Occurs when a mouse double-click
error	Occurs when there is an error in loading or unloading etc.
focus	Occurs when the element gets focus
keydown	Occurs when key is pressed
keypress	Occurs when key is pressed and released
keyup	Occurs when key is released
load	Occurs when document is loaded
mousedown	Occurs when mouse button is pressed
mouseenter	Occurs when mouse enters in an element region
mouseleave	Occurs when mouse leaves an element region
mousemove	Occurs when mouse pointer moves
mouseout	Occurs when mouse pointer moves out of an element
mouseover	Occurs when mouse pointer moves over an element
mouseup	Occurs when mouse button is released
resize	Occurs when window is resized
scroll	Occurs when window is scrolled
select	Occurs when a text is selected
submit	Occurs when form is submitted
unload	Occurs when documents is unloaded

jQuery – AJAX

Methods and Description
jQuery.ajax(options) Load a remote page using an HTTP request.
jQuery.ajaxSetup(options) Setup global settings for AJAX requests.
jQuery.get(url, [data], [callback], [type]) Load a remote page using an HTTP GET request.
jQuery.getJSON(url, [data], [callback]) Load JSON data using an HTTP GET request.
jQuery.getScript(url, [callback]) Loads and executes a JavaScript file using an HTTP GET request.
jQuery.post(url, [data], [callback], [type]) Load a remote page using an HTTP POST request.
load(url, [data], [callback]) Load HTML from a remote file and inject it into the DOM.
serialize( ) Serializes a set of input elements into a string of data.
serializeArray( ) Serializes all forms and form elements like the .serialize() method but returns a JSON data structure for you to work with.

Based on different events/stages following methods are available:

Methods and Description
ajaxComplete(callback) Attach a function to be executed whenever an AJAX request completes.
ajaxStart(callback) Attach a function to be executed whenever an AJAX request begins and there is none already active.
ajaxError(callback) Attach a function to be executed whenever an AJAX request fails.
ajaxSend(callback) Attach a function to be executed before an AJAX request is sent.
ajaxStop(callback) Attach a function to be executed whenever all AJAX requests have ended.
ajaxSuccess(callback) Attach a function to be executed whenever an AJAX request completes successfully.

jQuery – Effects

Methods and Description
animate(params, [duration, easing, callback]) A function for making custom animations.
animate(params, options) A function for making custom animations.
fadeIn(speed, [callback]) Fade in all matched elements by adjusting their opacity and firing an optional callback after completion.
fadeOut(speed, [callback]) Fade out all matched elements by adjusting their opacity to 0, then setting display to "none" and firing an optional callback after completion.
fadeToggle(speed, [callback]) This function toggles between fadeIn and fadeOut
fadeTo(speed, opacity, callback) Fade the opacity of all matched elements to a specified opacity and firing an optional callback after completion.
hide( ) Hides each of the set of matched elements if they are shown.
hide(speed, [callback]) Hide all matched elements using a graceful animation and firing an optional callback after completion.
show( ) Displays each of the set of matched elements if they are hidden.
show(speed, [callback]) Show all matched elements using a graceful animation and firing an optional callback after completion.
slideDown(speed, [callback]) Reveal all matched elements by adjusting their height and firing an optional callback after completion.
slideToggle(speed, [callback]) Toggle the visibility of all matched elements by adjusting their height and firing an optional callback after completion.
slideUp(speed, [callback]) Hide all matched elements by adjusting their height and firing an optional callback after completion.
stop([clearQueue, gotoEnd]) Stops all the currently running animations on all the specified elements.
toggle( ) Toggle displaying each of the set of matched elements.
toggle(speed, [callback]) Toggle displaying each of the set of matched elements using a graceful animation and firing an optional callback after completion.

toggle(switch) Toggle displaying each of the set of matched elements based upon the switch (true shows all elements, false hides all elements).
jQuery.fx.off Globally disable all animations.

UI Library Based Effects:

To use these effects you would have to download jQuery UI Library **jquery-ui-1.7.2.custom.min.js** or latest version of this UI library from jQuery UI Library.

Methods and Description
Blind Blinds the element away or shows it by blinding it in.
Bounce Bounces the element vertically or horizontally n-times.
Clip Clips the element on or off, vertically or horizontally.
Drop Drops the element away or shows it by dropping it in.
Explode Explodes the element into multiple pieces.
Fold Folds the element like a piece of paper.
Highlight Highlights the background with a defined color.
Puff Scale and fade out animations create the puff effect.
Pulsate Pulsates the opacity of the element multiple times.
Scale Shrink or grow an element by a percentage factor.
Shake Shakes the element vertically or horizontally n-times.
Size Resize an element to a specified width and height.
Slide Slides the element out of the viewport.
Transfer Transfers the outline of an element to another.

Chapter 39 – AngularJS

Conceptual Overview

AngularJS is not a library. It's a JavaScript framework that embraces extending HTML into a more expressive and readable format.

Concept	Description
Template	HTML with additional markup
Directives	extend HTML with custom attributes and elements
Model	the data shown to the user in the view and with which the user interacts
Scope	context where the model is stored so that controllers, directives and expressions can access it
Expressions	access variables and functions from the scope
Compiler	parses the template and instantiates directives and expressions
Filter	formats the value of an expression for display to the user
View	what the user sees (the DOM)
Data Binding	sync data between the model and the view
Controller	the business logic behind views
Dependency Injection	Creates and wires objects and functions
Injector	dependency injection container
Module	a container for the different parts of an app including controllers, services, filters, directives which configures the Injector
Service	reusable business logic independent of views

Chapter 40 – AJAX, JSON and DOJO

AJAX (Asynchronous JavaScript and XML)

AJAX allows web pages to be updated asynchronously by exchanging small amounts of the data with the server behind the scenes, without reloading the whole page.

- AJAX is a technique for creating fast and dynamic web pages
- AJAX applications are browser and platform-independent
- With AJAX, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page
- Data is usually retrieved using the XMLHttpRequest object
- Despite the name, the use of XML is not needed, and the requests need not be asynchronous

AJAX is based on Internet standards, and uses a combination of:

-
- XMLHttpRequest object(to exchange the data asynchronously with a server)
- JavaScript/DOM (to display/interact with the data)
- CSS (to style the data)
- XML (often used as the format for transferring data)

Create an XMLHttpRequest Object:

```
If(window.XMLHttpRequest) {
   XMLHttp = new XMLHttpRequest();
}
else // IE 5 or 6
XMLHttp = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Send a Request to a Server:

```
open(method,URL, async)
send(string for POST)
method - GET or POST
```

GET:

1. Simpler and Faster than POST
2. Used frequently

POST:
1. A cached file is not an option(update file or database)
2. Sending a large amount of data(has no limitations)
3. Sending user input(unknown characters)
4. Robust and Secure than GET

```
XMLHttp.open("GET", "demo.JSP", true);
XMLHttp.send();
```

```
XMLHttp.open("POST", "demo.JSP", true);
XMLHttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
XMLHttp.send(prop=val&prop=Val);
```

ASYNC=true

```
XMLHttp.onreadystatechange=function (){
if(XMLHttp.readyState == 4 && XMLHttp.status == 200) {
 document.getElementById("div").innerHTML=XMLHttp.responseText;
}
}
XMLHttp.open("GET", "demo.JSP", true);
XMLHttp.send();
```

ASYNC=false

```
XMLHttp.open("GET", "demo.JSP", true);
XMLHttp.send();
document.getElementById("div").innerHTML=XMLHttp.responseText;
```

Server Response:

1. responseText
2. responseXML

Ready State:

onreadystatechange event
- Stores a function(or the name of a function) to be called automatically each time the readyState property changes

readyState property
- Holds the status of the XMLHttpRequest. Changes from 0 to 4.
0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready

status property
- 200: "OK"
  404: Page not Found

Using a Callback Function

A callback function is a function passed as a parameter to another function.

Using responseXML property:

```
XmlDoc = XMLHttp.responseXML.documentElement;
Tags = XmlDoc.getElementByTagName("name");

For(int I=0; I<tags.length; I++){
 txt= Tags[I].firstChild.nodeValue; or. txt=Tags[I].childNodes[0].nodeValue;
}
document.getElementById("div").innerHTML=txt;
```

JSON (JavaScript Object Notation)

JSON is a Java library that helps convert Java objects into a string representation. This string, when eval()ed in JavaScript, produces an array that contains all of the information that the Java object contained. JSON's object notation grammar is suitable for encoding many nested object structures. Since this grammar is much smaller than its XML counterpart, and given the convenience of the eval() function, it is an ideal choice for fast and efficient data transport between browser and server.

JSON is designed to be used in conjunction with JavaScript code making HTTP requests. Since server-side code can be written in a variety of languages, JSON is available in many different languages such as C#, Python, PHP, and of course, Java.

DOJO

Dojo is a set of powerful JavaScript libraries that provide a simple API to a plethora of features. One of these features is the ability to make HTTP requests and receive their responses. This is the main functionality of Dojo that we will utilize. Aside from providing Ajax functionality, Dojo also provides packages for string manipulation, DOM manipulation, drag-and-drop support, and data structures such as lists, queues, and stacks.

Chapter 41 – UNIX Shell Script

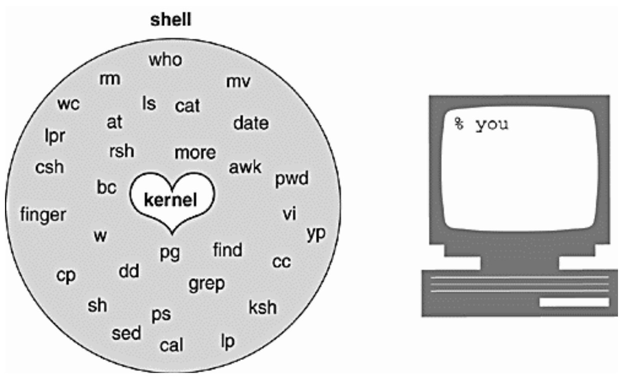
UNIX

UNIX stands for UNiplexed Information and Computing System. It was originally spelled "Unics."
It is a powerful, multi-user environment that has been implemented on a variety of platforms. Once the domain of servers and advanced users, it has become accessible to novices as well through the popularity of Linux, Solaris and Mac OS X. With the notable exception of Microsoft Windows, all current major operating systems have some kind of UNIX at their cores.

Shell

The shell is a special program used as an interface between the user and the heart of the UNIX operating system, a program called the kernel, as shown in Figure 1.1. The kernel is loaded into memory at boot-up time and manages the system until shutdown. It creates and controls processes, and manages memory, file systems, communications, and so forth. All other programs, including shell programs, reside out on the disk. The kernel loads those programs into memory, executes them, and cleans up the system when they terminate. The shell is a utility program that starts up when you log on. It allows users to interact with the kernel by interpreting commands that are typed either at the command line or in a script file.

Figure 1.1. The kernel, the shell, and you.



The Three Major UNIX Shells

The three prominent and supported shells on most UNIX systems are

1. the Bourne shell (AT&T shell),
2. the C shell (Berkeley shell), and
3. the Korn shell (superset of the Bourne shell).

The Bourne shell is the standard UNIX shell, and is used to administer the system. Most of the system administration scripts, such as the rc start and stop scripts and shutdown are Bourne shell scripts, and when in single user mode, this is the shell commonly used by the administrator when running as root. This shell was written at AT&T and is known for being concise, compact, and fast. The default Bourne shell prompt is the dollar sign ($).

The C shell was developed at Berkeley and added a number of features, such as command line history, aliasing, built-in arithmetic, filename completion, and job control. The C shell has been favored over the Bourne shell by users running the shell interactively, but administrators prefer the Bourne shell for scripting, because Bourne shell scripts are simpler and faster than the same scripts written in C shell. The default C shell prompt is the percent sign (%).

The Korn shell is a superset of the Bourne shell written by David Korn at AT&T. A number of features were added to this shell above and beyond the enhancements of the C shell. Korn shell features include an editable history, aliases, functions, regular expression wildcards, built-in arithmetic, job control, co-processing, and special debugging features. The Bourne shell is almost completely upward-compatible with the Korn shell, so older Bourne shell programs will run fine in this shell. The default Korn shell prompt is the dollar sign ($).

Responsibilities of the Shell

The shell is ultimately responsible for making sure that any commands typed at the prompt get properly executed. Included in those responsibilities are:

1. Reading input and parsing the command line.
2. Evaluating special characters.
3. Setting up pipes (|), redirection (>), and background processing (&).
4. Handling signals.
5. Setting up programs for execution.

Types of Commands

When the shell is ready to execute the command, it evaluates command types in the following order:

1. Aliases
2. Keywords
3. Functions (bash)
4. Built-in commands
5. Executable programs

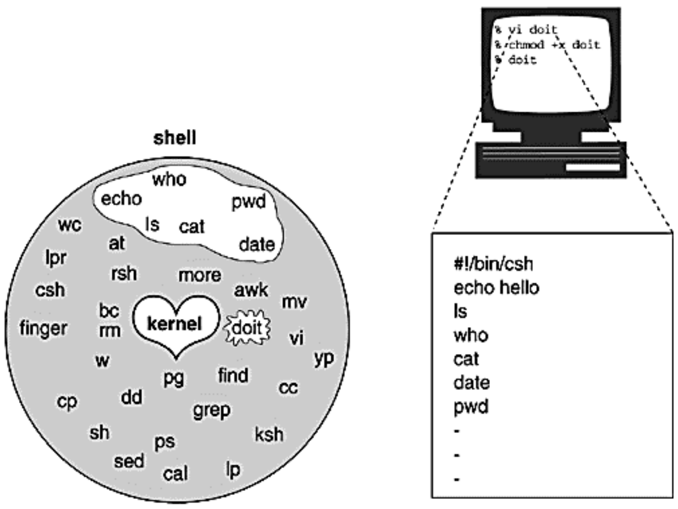
Parsing the Command Line

1. History substitution is performed (if applicable).
2. Command line is broken up into tokens, or words.
3. History is updated (if applicable).
4. Quotes are processed.
5. Alias substitution and functions are defined (if applicable).
6. Redirection, background, and pipes are set up.
7. Variable substitution ($user, $name, etc.) is performed.
8. Command substitution (echo for today is 'date') is performed.
9. Filename substitution, called globbing (cat abc.??, rm *.c, etc.) is performed.
10. Program execution.

Shell Script

A shell script is a script written for the shell, or command line interpreter, of an operating system. The shell is often considered a simple domain-specific programming language. The typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Creating a generic shell script



```
% vi doit
% chmod +x doit
% doit
```

```
#!/bin/csh
echo hello
ls
who
cat
date
pwd
-
-
-
```

Bourne Shell Script Quick Reference

I. Introduction to Shell Scripts

A. The shell is the program that you run when you log in
B. It is a command interpreter
C. There are three standard shells - C, Korn and Bourne
D. Shell prompts users, accepts command, parses, then interprets command
E. Most common form of input is command line input
```
cat file1 file2 file3
```
F. Most commands are of the format
```
command [- option list] [argument list]
```
G. Redirection and such
1. < redirect input from standard input
2. > redirect output from standard output
3. >> redirect output and append
4. | "pipes" output from one command to another
```
ls -l | more
```
5. `tee` "pipes" output to file and standard out
```
ls -l | tee rpt2 | more
```
H. Entering commands
1. Multiple commands can be entered on the same line if separated by ;
2. Command can span multiple lines if \R is typed at the end of each line except the last (R stands for carriage return, i.e. ENTER). This is escape sequence.

I. Wild card characters can be used to specify file names in commands
1. * 0 or more characters
2. ? one character of any kind
3. [, ,] list of characters to match single character

J. Simplest scripts combine commands on single line like
```
ls -l | tee rpt2 | more
```
K. Slightly more complex script will combine commands in a file
1. Use any text editor to create file, say `my_sc`
2. Type commands into file, one per line (unless you use ; to seperate)
3. Save file
4. Make file readable and executable (more later on this)
```
chmod a+rx my_sc
```
5. run script by entering path to file
```
./my_sc
```
We will make this a little easier later

L. See examples 1 and 2

II. Variables

A. The statment name=value creates and assigns value to variable
```
SUM=12
```
B. Traditional to use all upper case characters for names
C. Access content of variable by preceding name with $
```
echo $SUM
```
D. Arguments go from right to left
E. Results of commands can be assigned to variables
```
SYS=`hostname`
```
F. Strings are anything delimited by ""
G. Variables used in strings are evaluated
H. See example 3
I. System/standard variables
1. Command line arguments
Accessed by `$1` through `$9` for the first 9 command line arguments. Can access more by using the `shift` command. This makes $1 .. $9 reference command line arguments 2-10. It can be repeated to access a long list of arguments.
2. `$#` number of arguments passed on the command line
3. `$-` Options currently in effect (supplied to sh or to set
4. `$*` all the command line arguments as one long double quoted string
5. `$@` all the command line arguments as a series of double quoted strings
6. `$?` exit status of previous command
7. `$$` PID ot this shell's process
8. `$!` PID of most recently started background job
9. `$0` First word, that is, name of command/script

III. Conditional Variable Substitution

A. `${var:-string}` Use var if set, otherwise use string
B. `${var:=string}` Use var if set, otherwise use string and assign string to var
C. `${var:?string}` Use var if set, otherwise print string and exit
D. `${var:+string}` Use string if var if set, otherwise use nothing

IV. Conditional

A. The *condition* part can be expressed two ways. Either as
```
test condition
```
or
```
[ condition ]
```
where the spaces are significant.

B. There are several conditions that can be tested for
1. `-s file` file greater than 0 length
2. `-r file` file is readable

3. `-w file` file is writable
4. `-x file` file is executable
5. `-f file` file exists and is a regular file
6. `-d file` file is a directory
7. `-c file` file is a character special file
8. `-b file` file is a block special file
9. `-p file` file is a named pipe
10. `-u file` file has SUID set
11. `-g file` file has SGID set
12. `-k file` file has sticky bit set
13. `-z string` length of string is 0
14. `-n string` length of string is greater than 0
15. `string1 = string2` string1 is equal to string2
16. `string1 != string2` string1 is different from string2
17. `string` string is not null
18. `int1 -eq int2` integer1 equals integer2
19. `int1 -ne int2` integer1 does not equal integer2
20. `int1 -gt int2` integer1 greater than integer2
21. `int1 -ge int2` integer1 greater than or equal to integer2
22. `int1 -lt int2` integer1 less than integer2
23. `int1 -le int2` integer1 less than or equal to integer2
24. `! condition` negates (inverts) condition
25. `cond1 -a cond2` true if condition1 and condition2 are both true
26. `cond1 -o cond2` true if either condition1 or condition2 are true
27. `\( \)` used to group complex conditions

V. Flow Control

A. The if statement

```
if condition
then
commands
else
commands
fi
```

B. Both the while and until type of loop structures are supported

```
while condition
do
commands
done

until condition
do
commands
done
```

C. The case statement is also supported

```
case string in
pattern1)
commands
;;

pattern2)
commands
;;

esac
```

The pattern can either be an integer or a single quoted string

The * is used as a catch-all default pattern

D. The for command

```
for var [in list]
do
commands
done
```

where either a list (group of double quoted strings) is specified, or $@ is used

VI. Other Commands

A. Output
1. Use the `echo` command to display data
2. `echo "This is some data"` will output the string
3. `echo "This is data for the file = $FILE"` will output the string and expand the variable first. The output from an `echo` command is automatically terminated with a newline.

B. Input
1. The `read` command reads a line from standard input
2. Input is parsed by whitespace, and assigned to each respective variable passed to the `read` command
3. If more input is present than variables, the last variable gets the remainder
4. If for instance the command was `read a b c` and you typed "Do you Grok it" in response, the variables would contain $a="Do", $b="you" $c="Grok it"

C. Set the value of variables $1 thru $n
1. If you do `` set `command` ``, then the results for the command will be assigned to each of the variables $1, $2, etc. parsed by whitespace

D. Evaluating expressions
1. The `expr` command is used to evaluate expressions

2. Useful for integer arithmetic in shell scripts `i=`expr $i +1``

E. Executing arguments as shell commands

1. The `eval` command executes its arguments as a shell command

VII. Shell functions

A. General format is

```
B.   function_name ()
C.   {
D.   commands
E.   }
```

VIII. Miscellaneous

A. \n at end of line continues on to next line

B. Metacharacters

1. * any number of characters
2. ? any one character
3. [,] list of alternate characters for one character position

C. Substitution

1. delimit with `` (back quote marks, generally top left corner of keyboard)
2. executes what is in `` and substitutes result in string

D. Escapes

1. \ single character
2. ' groups of characters
3. " groups of characters, but some special characters processed ($\`)

E. Shell options

1. Restricted shell sh -r
   a. can't cd, modify PATH, specify full path names or redirect output
   b. should not allow write permissions to directory
2. Changing shell options
   a. Use set option +/- to turn option on/off
   b. e interactive shell
   c. f filename substitution
   d. n run, no execution of commands
   e. u unset variables as errors during substitution
   f. x prints commands and arguments during execution

Examples

Single Line Script

```
#! bin/sh
# Script lists all files in current
directory in decending order by size

ls -l | sort -r -n  +4 -5
```

Multiline Script

```
#!/usr/bin/ksh
# Lists 10 largest files in current
directory by size

ls -l > /tmp/f1
sort -r -n +4 -5 /tmp/f1 > /tmp/f2
rm /tmp/f1
head /tmp/f2 > /tmp/f3
rm /tmp/f2
more /tmp/f3
rm /tmp/f3
```

```
#!/usr/bin/ksh
# Uses variables to store data from
commands

SYS=`hostname`
ME=`whoami`
W="on the system"
echo "I am $ME $W $SYS"
```

This is a quick reference guide to the meaning of some of the less easily guessed commands and codes.

Command	**Description**	**Example**
&	Run the previous command in the background	ls &
&&	Logical AND	if ["$foo" -ge "0"] && ["$foo" -le "9"]
||	Logical OR	if ["$foo" -lt "0"] \|\| ["$foo" -gt "9"] (not in Bourne shell)
^	Start of line	grep "^foo"
$	End of line	grep "foo$"
=	String equality (cf. -eq)	if ["$foo" = "bar"]
!	Logical NOT	if ["$foo" != "bar"]
$$	PID of current shell	echo "my PID = $$"
$!	PID of last background command	ls & echo "PID of ls = $!"
$?	exit status of last command	ls ; echo "ls returned code $?"
$0	Name of current command (as called)	echo "I am $0"
$1	Name of current command's first parameter	echo "My first argument is $1"
$9	Name of current command's ninth parameter	echo "My ninth argument is $9"
$@	All of current command's parameters (preserving whitespace and quoting)	echo "My arguments are $@"
$*	All of current command's parameters (not preserving whitespace and quoting)	echo "My arguments are $*"
-eq	Numeric Equality	if ["$foo" -eq "9"]
-ne	Numeric Inquality	if ["$foo" -ne "9"]
-lt	Less Than	if ["$foo" -lt "9"]
-le	Less Than or Equal	if ["$foo" -le "9"]
-gt	Greater Than	if ["$foo" -gt "9"]
-ge	Greater Than or Equal	if ["$foo" -ge "9"]
-z	String is zero length	if [-z "$foo"]
-n	String is not zero length	if [-n "$foo"]
-nt	Newer Than	if ["$file1" -nt "$file2"]
-d	Is a Directory	if [-d /bin]
-f	Is a File	if [-f /bin/ls]
-r	Is a readable file	if [-r /bin/ls]
-w	Is a writable file	if [-w /bin/ls]
-x	Is an executable file	if [-x /bin/ls]
parenthesis: (...)	Function definition	function myfunc() { echo hello }

If Necessity is the Mother of Invention then Curiosity is the Father and we're their Kids!