Notes

* **a.compareTo(b):**  
  **Comparable interface** : Compares values and **returns an int** which tells if the values compare less than, equal, or greater than.  
  If your class objects have a natural order, implement the Comparable<T> interface and define this method. All Java classes that have a **natural ordering** implement Comparable<T> - Example: **String,**[**wrapper classes**](http://www.javatpoint.com/wrapper-class-in-java)**, BigInteger**
* ***compare(a, b):***  
  ***Comparator interface*** : Compares values of two objects. This is implemented as part of the Comparator<T> interface, and the typical use is to define one or more small utility classes that implement this, to pass to methods such as sort() or for use by sorting data structures such as ***TreeMap*** and ***TreeSet***. You might want to create a Comparator object for the following:
  + ***Multiple comparisons.*** To provide several different ways to sort something. For example, you might want to sort a Person class by name, ID, age, height, ... You would define a Comparator for each of these to pass to the sort() method.
  + ***System class*** To provide comparison methods for classes that you have no control over. For example, you could define a Comparator for Strings that compared them by length.
  + ***Strategy pattern*** To implement a Strategy pattern, which is a situation where you want to represent an algorithm as an object that you can pass as a parameter, save in a data structure, etc.

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

**Built-in Packages**  
These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:  
1) **java.lang:**Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.  
2) **java.io:**Contains classed for supporting input / output operations.  
3) **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.  
4) **java.applet:**Contains classes for creating Applets.  
5) **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).  
6) **java.net:**Contain classes for supporting networking operations.

Only major difference between **Enumeration** and ***iterator*** is Iterator has a ***remove() method*** while Enumeration doesn't. Enumeration acts as **Read-only interface**, because it has the methods only to **traverse and fetch the objects**, where as by using ***Iterator*** we can ***manipulate the objects like adding and removing the objects*** from collection e.g. Arraylist.

**Enumeration :** hasMoreElement(), nextElement(), N/A

***Iterator* :** hasNext(), next(), remove()

**boolean hasNext():** this method returns true if this Iterator has more element to iterate.

**Object next()**: return the next element in the collection until the hasNext() method return true. Its always recommended to call hasNext() method before calling next() method to avoid [java.util.NoSuchElementException: Hashtable Iterator](http://javarevisited.blogspot.com/2012/02/how-to-solve-javautilnosuchelementexcep.html)

**remove():** method remove the last element return by the iterator this method only calls once per call to next().

***Iterator*** is more secure it doesn’t allow another thread to modify the collection when some another thread is iterating the collection and throws concurrentModificationException

To remove objects from Collection than don't use for-each loop instead use Iterator's remove() method to avoid any ConcurrentModificationException.

**Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element**

***fail-fast Iterators*** fail as soon as they realized that structure of Collection has been changed since iteration has begun. Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection. Ex: Iterators of Vector, ArrayList, HashSet

***fail-safe iterator*** doesn't throw any Exception if Collection is modified structurally

while one thread is Iterating over it because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator. Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException in Java.

All the collection classes in **java.util package are fail-fast** whereas collection classes in ***java.util.concurrent are fail-safe***.

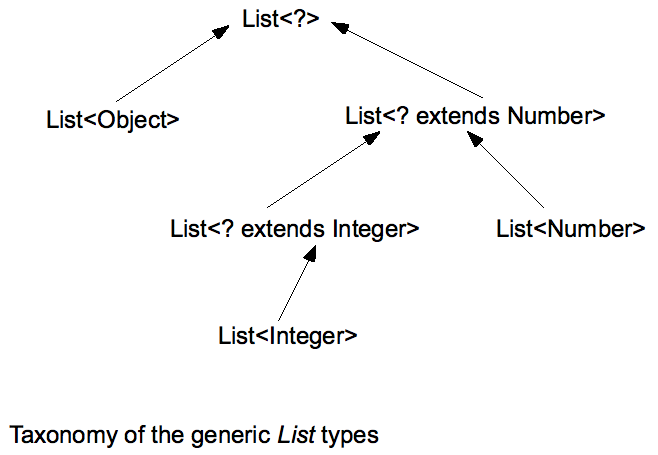
**ListIterator** : It is an Iterator which allows user to **traverse** Collection like [ArrayList and](http://www.blogger.com/goog_1292333036)[HashSet](http://javarevisited.blogspot.com/2012/01/convert-arraylist-to-set-java-example.html) **in both direction by using method previous() and next ().** You can obtain ListIterator from all List implementation including [ArrayList and LinkedList](http://javarevisited.blogspot.com/2012/02/difference-between-linkedlist-vs.html). ListIterator doesn’t keep current index and its current position is determined by call to next() or previous() based on direction of traversing.

|  |  |  |
| --- | --- | --- |
| **Iterator** | **ListIterator** | **Enumerator** |
| hasNext() | hasNext() | hasMoreElement() |
| next() | next() | nextElement() |
| remove() | remove() | N/A |
|  | previous() |  |

**Generics** in Java: Generic in Java is added to provide compile time type-safety of code and removing risk of ClassCastException at [runtime](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html). For example, HashSet of String will only contain String object and if you try to put Integer or any other object, the compiler will complain.

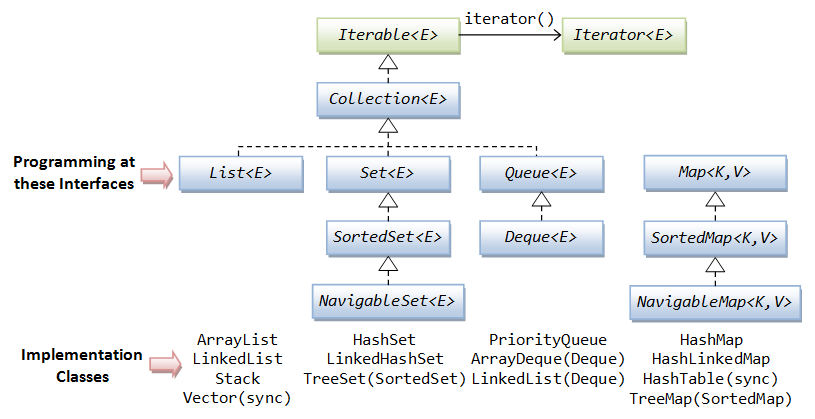
Advantages of generics: Type safety, no ClassCastException

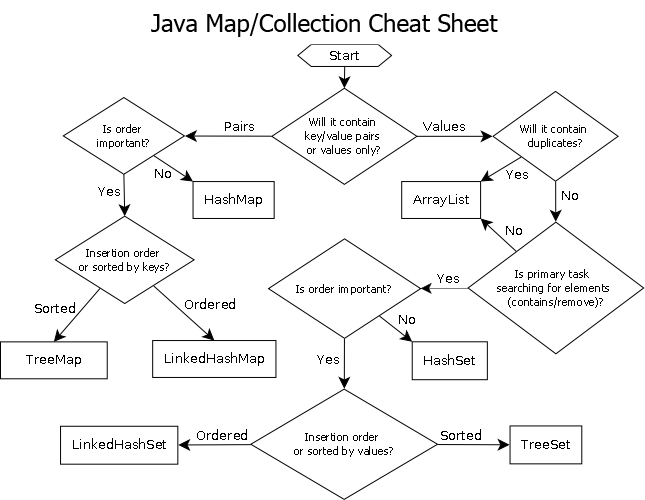
Ex: [HashMap](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html)<String, Set<Integer>> contacts = new HashMap<String,Set<Integer>>()

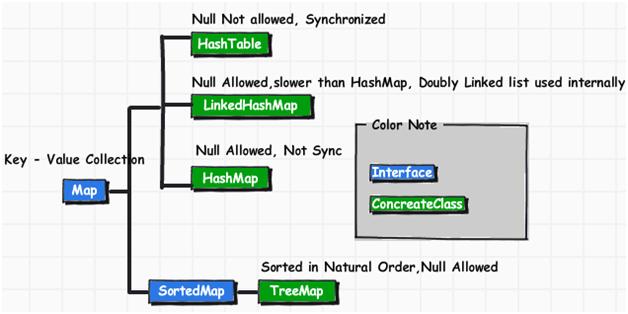


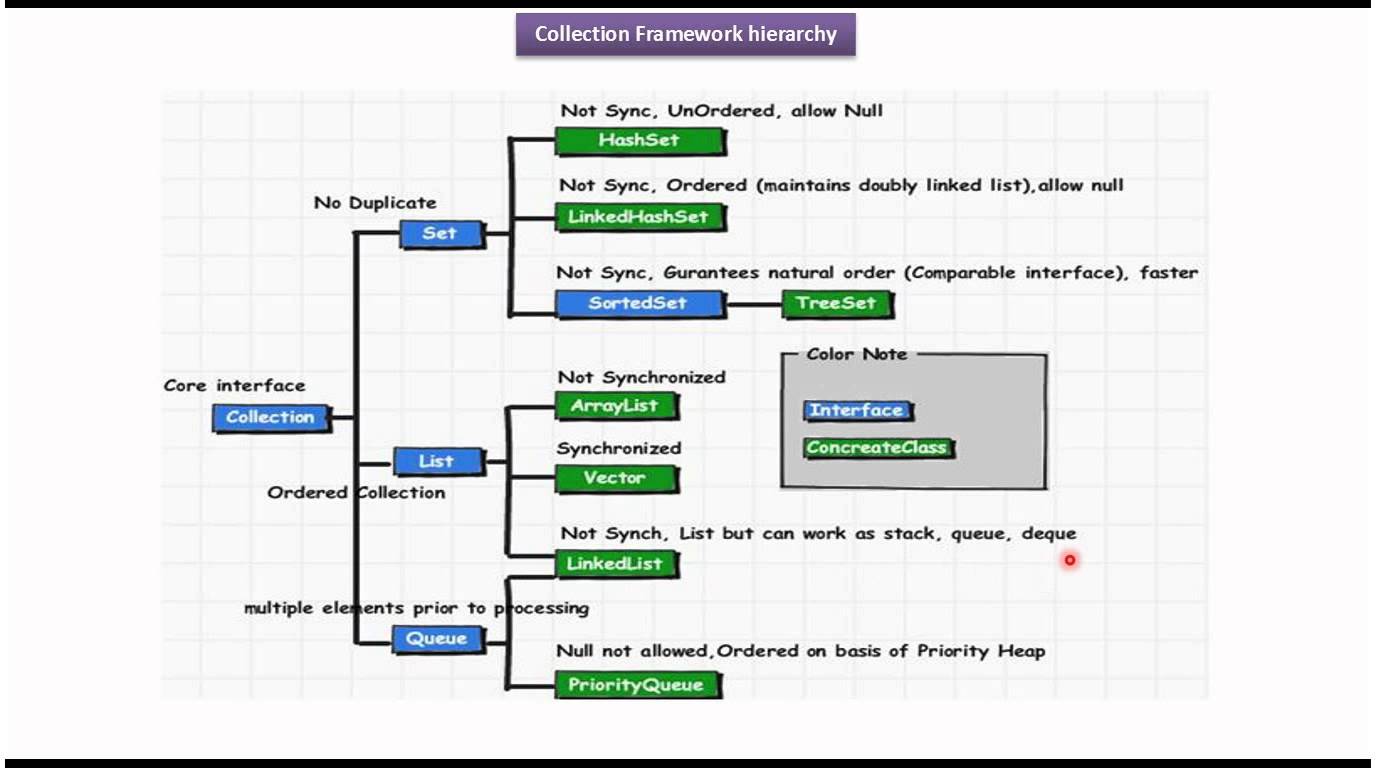
**for each loop:** The for-each loop is used to traverse array or collection in java. It works on elements basis not index. It returns element one by one in the defined variable. Syntax: for(Type var:array){ }

**Collections**









Principal features of non-primary implementations:

* HashMap has slightly better performance than LinkedHashMap, but its iteration order is undefined
* HashSet has slightly better performance than LinkedHashSet, but its iteration order is undefined
* TreeSet is ordered and sorted, but slower
* TreeMap is ordered and sorted, but slower
* LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration

**To avoid ConcurrentModificationException** while iterating over a collection------

Sometimes we want to add or remove elements from the list if we find some specific element, in that case we should use concurrent collection class – **CopyOnWriteArrayList.** This is **a thread-safe variant of java.util.ArrayList** in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

**ConcurrentHashMap**is the class that is similar to HashMap but **works fine** when you try to **modify your map at runtime**.

**Concurrent collection classes: CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet**

Chosing the right type of collection based on the need, for example **if size is fixed**, we might want to use **Array over ArrayList**. If we have to ***iterate over the Map in order of insertion***, we need to use ***TreeMap***. If we **don’t want duplicates**, we should use **Set**.

There are two ways we could synchronized [HashMap](http://crunchify.com/java-hashmap-containskeyobject-key-and-containsvalueobject-value-check-if-key-exists-in-map/" \t "_blank)

* Java Collections synchronizedMap() method
* Use ConcurrentHashMap

**ConcurrentHashMap**

* You should use ConcurrentHashMap when you need very high concurrency in your project.
* It is thread safe without synchronizing the whole map.
* Reads can happen very fast while write is done with a lock.
* There is no locking at the object level.
* The locking is at a much finer granularity at a hashmap bucket level.
* ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
* ConcurrentHashMap uses multitude of locks.

**SynchronizedHashMap**

* Synchronization at Object level.
* Every read/write operation needs to acquire lock.
* Locking the entire collection is a performance overhead.
* This essentially gives access to only one thread to the entire map & blocks all the other threads.
* It may cause contention.
* SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

**Main method is static** because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

**this:** It is a reference variable that refers to the current object.

**Cloning:** object cloning is a process of generating the exact copy of object with the different name. Always override the clone() method of Object class in the respective class which needs to be cloned.

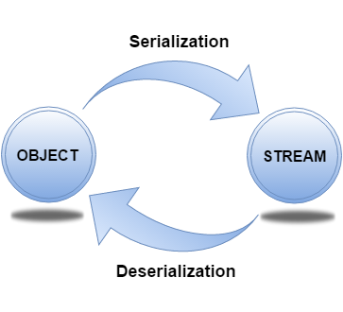
@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

**Marker interface**: Don’t have any method or data member. Ex: **Cloneable, Serializable, Remote**



**Serialization in java** is a mechanism of *writing the state of an object into a byte stream*.

**String** class and all the **wrapper classes** implements *java.io.Serializable* interface by default.

**ObjectOutputStream** class is used to write primitive data types and Java objects to an OutputStream.

public final void writeObject(Object obj) throws IOException {} writes the specified object to the ObjectOutputStream.

**Deserialization** is the process of reconstructing the object from the serialized state

**ObjectInputStream** deserializes objects and primitive data written using an ObjectOutputStream.

public final Object readObject() throws IOException, ClassNotFoundException{} reads an object from the input stream.

* If a class implements serializable then all its sub classes will also be serializable
* If a class has a reference of another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, NotSerializableException is thrown at runtime.
* If there is any static data member in a class, it will not be serialized because static is the part of class not object.
* In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed.
* If you **don't want to serialize any data member** of a class, you can mark it as **transient**.

**Java Reflection** is a process of examining or modifying the run time behavior of a class at run time.

The **==** operator tests whether two variables have the **same references (aka pointer to a memory address). (Value and ref)**

Whereas the **equals()** method tests whether two variables refer to objects that have the **same state (values). (Only value)**

**Hashing** is a way to assign a unique code for any variable/object after applying any function/algorithm on its properties.

A true Hashing function must follow this rule: Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.

**Multiple inheritance** **can be used in Java 8 using INTERFACE only. Default method** to be used.

**First** **solution** is to create an own method that overrides the default implementation.

public class car implements vehicle, fourWheeler {

default void print() {

System.out.println("I am a four wheeler car vehicle!");

}

}

**Second** **solution** is to call the default method of the specified interface using super.

public class car implements vehicle, fourWheeler {

default void print() {

vehicle.super.print();

}

}

**Collection API improvements:** We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

* Iterator default method forEachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
* Collection default method removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.
* Collection spliterator() method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
* Map replaceAll(), compute(), merge() methods.
* Performance Improvement for HashMap class with Key Collisions

**Concurrency API improvements:** Some important concurrent API enhancements are:

* ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
* CompletableFuture that may be explicitly completed (setting its value and status).
* Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

**Java IO improvements:** Some IO improvements known to me are:

* Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.
* Files.lines(Path path) that reads all lines from a file as a Stream.
* Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
* BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

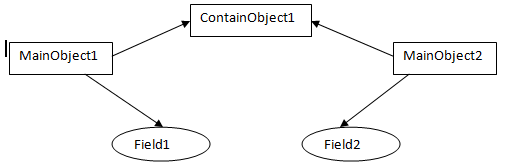
**Miscellaneous Core API improvements:** Some misc API improvements that might come handy are:

* ThreadLocal static method withInitial(Supplier supplier) to create instance easily.
* Comparator interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
* min(), max() and sum() methods in Integer, Long and Double wrapper classes.
* logicalAnd(), logicalOr() and logicalXor() methods in Boolean class.
* ZipFile.stream() method to get an ordered Stream over the ZIP file entries. Entries appear in the Stream in the order they appear in the central directory of the ZIP file.
* Several utility methods in Math class.
* jjs command is added to invoke Nashorn Engine.
* jdeps command is added to analyze class files
* JDBC-ODBC Bridge has been removed.
* PermGen memory space has been removed

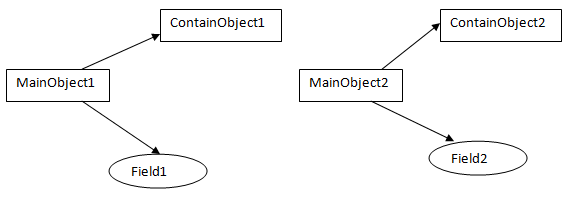
**Marker interface:**

Marker interface in java is an interface which does not have any method. **Marker interface is used to inform the JVM that the classes implementing them will have some special behavior**. In java we have following four major marker interfaces:

* Searilizable interface
  + During object serialization, the default Java serialization mechanism writes the metadata about the object, which includes the class name, field names and types, and superclass.
  + All this information is stored as part of the serialized object.
  + When you deserialize the object, this information is read to reconsitute the object.
  + But to perform the deserialization, the object needs to be identified first and this will be done by **serialVersionUID**.
  + So everytime an object is serialized the java serialization mechanism automatically computes a hash value using ObjectStreamClass’s computeSerialVersionUID() method by passing the class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA), which returns a hash value, the serialVersionUID.
  + Now when the serilaized object is retrieved, the JVM first evaluates the serialVersionUID of the serialized class and compares the serialVersionUID value with the one of the object.
  + If the sserialVersionUID values match then the object is said to be compatible with the class and hence it is de-serialized.
  + If not InvalidClassException exception is thrown.
  + **Serialization: metadata written(classname, field name, type, superclass) 🡪 serialVersionUID (hash value) returned using ObjectStreamClass’s computeSerialVersionUID() method by passing class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA).**
  + **Deserialization: serialVersionUID of class is checked with serialVersionUID of one object 🡪 If matches: Deserialized, else: InvalidClassException exception**
  + In multi JVM environment, generated serialVersionUID can have different values. Instead of relying on the JVM to generate the serialVersionUID, you **explicitly mention (generate) the serialVersionUID** in your class. The syntax is:   
    **private final static long serialVersionUID = <integer value>**
  + Yes, its a static, private variable in the class. Once you define the serialVersionUID in your class explicitly, you dont need to update it until and unless you make the incompatible changes.
  + **Compatible** changes: Adding fields, Adding classes, Removing classes, Adding writeObject/readObject methods, Removing writeObject/readObject methods, Adding java.io.Serializable, Changing the access to a field, Changing a field from static to nonstatic or transient to nontransient
  + **Incompatible** changes: Deleting fields, Moving classes up or down the hierarchy, Changing a nonstatic field to static or a nontransient field to transient.
    - Changing the declared type of a primitive field.
    - Changing the writeObject or readObject method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not.
    - Changing a class from Serializable to Externalizable or visa-versa is an incompatible change since the stream will contain data that is incompatible with the implementation in the available class.
    - Removing either Serializable or Externalizable is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
  + **Externalization** is nothing but serialization but by implementing Externalizable interface to persist and restore the object.
    - To externalize your object, you need to implement Externalizable interface that extends Serializable interface
    - Unlike Serializable interface, Externalizable interface is not a marker interface and it provides two methods - **writeExternal** and **readExternal**.
* **Cloneable** interface: The purpose of Cloneable interface is to tell JVM that this class is eligible for cloning.
  + **Shallow copy**: It is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.



* + **Deep Copy**: A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.



**Shallow copy = Original object + fields + Reference to referenced objects.**

**Deep copy = Original object + fields + Copy of Reference objects.**

* + clone() is method of Object class.
  + When you invoke clone() method in your object, it should either:
    - 1. return an Object reference to a copy of the object upon which it is invoked, or
    - 2. throw CloneNotSupportedException
* Remote interface
* ThreadSafe interface

Any @Component or @Configuration can be marked with @Profile to limit when it is loaded

**Set multiple environments in spring boot**

* By default, [Spring Boot](http://docs.spring.io/spring-boot/docs/1.4.2.RELEASE/reference/htmlsingle/) look for your externalized configuration file ( application.proroperties ) in four predetermined locations :
  + in classpath root,
  + in the package /config in classpath,
  + in the current directory
  + in /config subdirectory of the current directory.
* 1st Way : using command arguments
  + Spring Boot provides the argument
    - **spring.config.name** to set configuration files names seperated with a comma.
    - **spring.config.location** in which you must set the locations where Spring Boot will find your externalized configuration files.
    - java -jar myproject.jar --spring.config.name=application,conf

--spring.config.location=classpath:/external/properties/,classpath:/com/roufid/tutorial/configuration/

* 2nd Way: use environment variables
  + To tell Spring Boot to pick up what you want where you want. set the name of your externalized configuration files in the **SPRING\_CONFIG\_NAME** environment variable separated with a comma and the locations in **SPRING\_CONFIG\_LOCATION**.
    - set SPRING\_CONFIG\_NAME=conf

set SPRING\_CONFIG\_LOCATION=classpath:/external/properties/,classpath:/com/roufid/tutorial/configuration/

java -jar myproject.jar

* 3rd Way: Programmatically

ConfigurableApplicationContext applicationContext = new SpringApplicationBuilder(Application.class)

.properties("spring.config.name:application,conf", "spring.config.location:classpath:/external/properties/,classpath:/com/roufid/tutorial/configuration/")

.build().run(args);

**Active profile in springboot** can be set by

The Spring Environment has an API for this, but normally you would set a System property (spring.profiles.active) or an OS environment variable (**SPRING\_PROFILES\_ACTIVE**). E.g. launch your application with a -D argument (remember to put it before the main class or jar archive):

**$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar**

In Spring Boot you can also set the active profile in application.properties, e.g.

**spring.profiles.active=production**

[**FailureAnalyzer**](https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/api/org/springframework/boot/diagnostics/FailureAnalyzer.html) is a great way to **intercept an exception on startup** **and turn it into a human-readable message**, wrapped into a [FailureAnalysis](https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/api/org/springframework/boot/diagnostics/FailureAnalysis.html" \t "_top). **AbstractFailureAnalyzer** is a convenient extension of FailureAnalyzer that **checks the presence of a specified exception type in the exception to handle**

**Change configuration depending on the environment**

* A YAML file is actually a sequence of documents separated by --- lines, and each document is parsed separately to a flattened map.
* If a YAML document contains a spring.profiles key, then the profiles value (comma-separated list of profiles) is fed into the SpringEnvironment.acceptsProfiles() and if any of those profiles is active that document is included in the final merge (otherwise not).

Example:

server:

port: 9000

---

spring:

profiles: development

server:

port: 9001

---

spring:

profiles: production

server:

port: 0

* In this example the default port is 9000, but if the Spring profile ‘development’ is active then the port is 9001, and if ‘production’ is active then it is 0.
* The YAML documents are merged in the order they are encountered (so later values override earlier ones).
* To do the same thing with properties files you can use application-${profile}.properties to specify profile-specific values.

**Logging**:

* Spring Boot uses [**Commons Logging**](https://commons.apache.org/logging) **for all internal logging**, but leaves the underlying log implementation open. Default configurations are provided for [Java Util Logging](http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html),[Log4J2](https://logging.apache.org/log4j/2.x/" \t "_top) and [Logback](http://logback.qos.ch/" \t "_top). In each case loggers are pre-configured to use console output with optional file output also available.
* **By default, If you use the ‘Starters’, Logback will be used for logging**. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J or SLF4J will all work correctly.

**Singleton Pattern:** Singleton-scoped beans

**Factory Pattern:** Bean Factory classes

**Prototype Pattern:** Prototype-scoped beans

**Adapter Pattern:** Spring Web and Spring MVC

**Proxy Pattern:** Spring Aspect Oriented Programming support

**Template Method Pattern:** *JdbcTemplate*, *HibernateTemplate,* etc.

**Front Controller:** Spring MVC *DispatcherServlet*

**Data Access Object:** Spring DAO support

**Model View Controller:**Spring MVC

There are three ways to handle exceptions in Spring MVC:

1. **Using @ExceptionHandler at controller level** – this approach has a major feature – the @ExceptionHandler annotated method is only active for that particular controller, not globally for the entire application
2. **Using HandlerExceptionResolver** – this will resolve any exception thrown by the application
3. **Using @ControllerAdvice** – Spring 3.2 brings support for a global @ExceptionHandler with the @ControllerAdvice annotation, which enables a mechanism that breaks away from the older MVC model and makes use of ResponseEntity along with the type safety and flexibility of @ExceptionHandler