Java Core

Rev 1.2 (21/07/2019)

Abstract classes
An abstract class contains the keyword “abstract” in its declaration and cannot be instantiated, if a class contains at least one abstract method the class must be declared as abstract.

- It may or may not contain abstract methods.
- The subclasses must provide implementation for all abstract methods in superclass.

Interfaces
Collection of abstract methods that have to be implemented, contract or api definition.

- Cannot be instantiated or contain constructor methods.
- Attributes contained in the class must be declared final and static and cannot hold instance fields.
- As from java 8, Interfaces can have default and static methods, private or public.
- Interfaces can be extended.

Interfaces vs abstract classes
- Interfaces are implemented, abstract classes inherited.
- As from java 8, both can contain code in methods.
- A class can can implement multiple interfaces but inherit just one class.

Polymorphism/method overriding
A subclass can not override a method that is marked as final or private, likewise constructors cannot be overriden, Overriding happens when a subclass has the same name, number/type of parameters, and the same return type as an instance method of the superclass.

Rules for method overriding:
- method signature (return type and argument list) must be the same.
- access level (private, public, default) cannot be more restrictive than the one of the parent class.
- Uncheck exceptions cannot be added or the class declaration in the method to be broader, although it can be more restrictive.

Pass by value
Method arguments are passed by value, the variables are duplicated to the function's scope, from the point of view of the caller these are treated as they were final.

```
function (final Object a){
        a = b; // a is the same for the caller
}
```

However, since an Object is a reference to an object in memory caller and callee will share the same object, therefore changes in function for that object will affect the caller.

```
function (final Object a){
        a.value  = 100; //affects the same value on the caller
}
```

Final keyword

- Classes declared as final cannot be inherit.
- Methods declared as final cannot be overridden.
- Variables declared as final can be only instantiated once, does not affect method invocation, (e.g. *final List<String> a = new ArrayList(); list.add("It's Ok");*)

Constants, must be declared final and static
public/private static final int MAX_SECONDS = 25;

Immutable
Immutable means that once the constructor for an object has completed execution that instance can't be modified, these means that references to the object can be passed around, without worrying about any function changing its contents.

To create a immutable class in java, you have to do following steps.

- Declare the class as final so it can't be extended.
- Don't provide setter methods for variables.
- Make all attributes private and final where necessary.
- Initialize all the fields via a constructor performing deep copy.
- Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

equals/hashcode
If two objects are equal according to *equals()* method, then *hashCode()* of the two objects must produce the same integer result. If you only override equals() and not hashCode() your class violates this contract

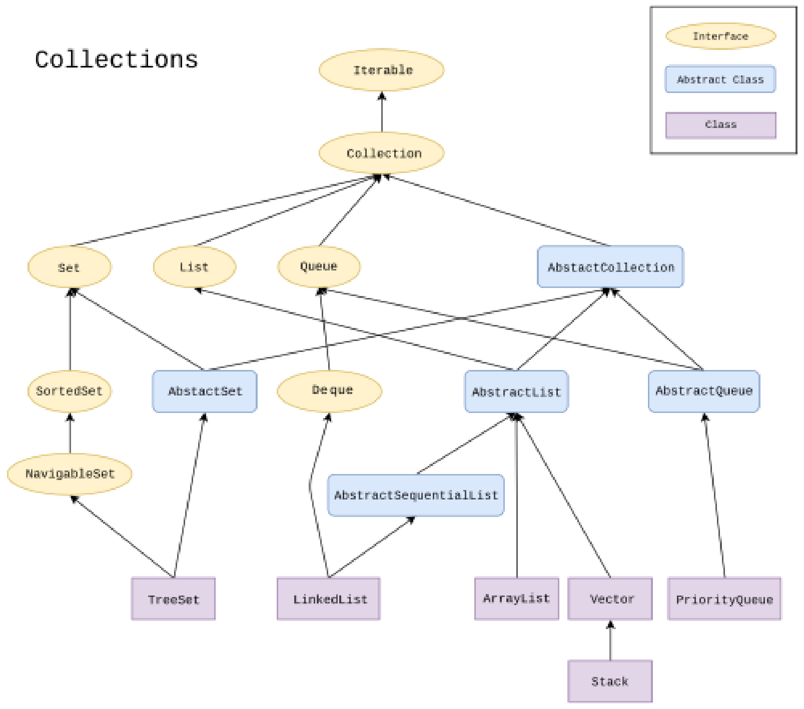
Comparable/Comparator

Interface *Comparable<T>* implement by T and contains method ***(int) compareTo(T)*** to get the *natural order* of the objects. Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Collections.sort(List<T>), T implements Comparable<T>, orders by ASC

Interface Comparator<T> allows to implement an external comparator classes for T, must implement ***(int) compare(T).***
There can be different classes that implement Comparator<T> for one specific attribute (e.g. CompareObjectByBate, CompareObjectByName, ..)

Collections.sort(List<T>, Comparator<T>)



List<T>

- Can contain duplicates
- Preserves insertion order of elements
- Can contain null values

Set<T>

- Unique values
- Unordered list
- Can contain one null value (no duplicates)
- LinkHashSet implementation keeps insert order

Queue<T>

- Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out), but also implements element interation.
- Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations
- Deque implements FIFO and FILO strategies, implementations shouldn't not allow null

Vector vs ArrayList

- Vectors are synchronized, ArrayLists are not.
- Data Growth Methods

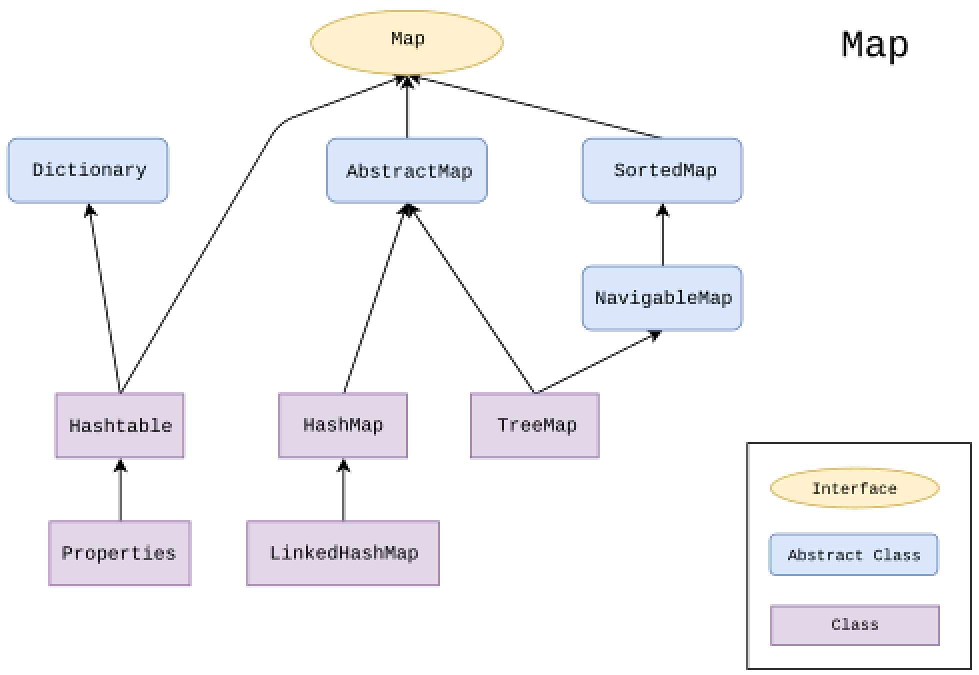
Vector is a legay implementation, should use Collections.synchronizedList(List<T> list) to get an synchronized list, Vector syncronizes the method and not the all list interaction.

Collections.synchronizedList(List<T> list)

Returns a synchronized (thread-safe) list backed by the specified list. In order to guarantee serial access, it is critical that all access to the backing list is accomplished through the returned list.
It is imperative that the user manually synchronize on the returned list when iterating over it:

```
List list = Collections.synchronizedList(new ArrayList());
    ...
synchronized(list) {
    Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.
The returned list will be serializable if the specified list is serializable.



Interface Map<String,T>

- No interface allows duplicate keys
- Some implementation allow null

HashMap<-,T> - is implemented as a hash table, and there is no ordering on keys or values.
TreeMap<-,T> - is implemented based on red-black tree structure, and it is ordered by the key. T must implement comparable<T>.
LinkedHashMap<-,T> Hash table and linked list implementation of the Map interface, with predictable iteration order, does not implement Iterable.

	synchronized	null values/keys
HashMap	N	Y
ConcurrentHashMap	Y	N
Hashtable	Y	N

Hashtable is a legay implementation, If a thread-safe highly-concurrent implementation is desired, then it is recommended to use ConcurrentHashMap in place of Hashtable.

Initial capacity/load factor

The load factor represents at what level the capacity of any storage object like List,Map,Tree etc should be increased.

Every object had initial capacity and when that capacity got filled till certain factor then it's capacity should increased or doubled.

Default initial capacity of the HashMap takes is 16 and load factor is 0.75f (i.e 75% of current map size). The load factor represents at what level the HashMap capacity should be doubled. For example product of capacity and load factor as 16 * 0.75 = 12 . Every time 12 more elements added to map.

HashMap/Hashtable

There are several differences between HashMap and Hashtable in Java:

Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.

Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.

Neither one maintains insertation order.

One of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.