CYBAGE
Delivering Value. Scientifically.

Collection Framework

Author & Presenter -Asfiya Khan
(Senior Technical Trainer)

Agenda

- Concept of Generics
- Wrapper Classes
- Need of Collection
- Collection Hierarchy
- List
- Set
- Iterating Collection
- Sorting In Collection
- Map
- Stream API – Java 8 Feature

Concept Of Generics

- Generics in Java is similar to templates in C++.
- The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces.
- we use <> to specify parameter types in generic class creation.
- Syntax :

  BaseType <Type> obj = new BaseType <Type>()

  Here <Type> can be replaced with only Wrapper Classes.

  In Parameter type we can not use primitives like 'int','char' or 'double'

Concept Of Generics - Need

- **Code Reuse** - We can write a method/class/interface once and use for any type we want.
- **Type Safety** - Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time)
- **Explicit Type Casting is not needed**

- **Implementing generic algorithms**

Wrapper Classes

- Java provide 8 primitive data types
- Sometimes there is a need to treat these primitives as objects
- Therefore Java provides **class counterparts** for those primitive data types which are known as Object Wrappers or Wrapper classes.
- Wrapper classes provide home for methods & variables related to the type.
- There are certain classes which can deal only with Objects & not primitive types (here wrapper classes are useful) : e.g. Collections
- Example: Integer i =new Integer(4); int x = i.intValue();
  int y = Integer.parseInt("4");

Auto Boxing and Unboxing

In JDK 1.4.x:

- As any Java programmer knows, you can't put an int (or other primitive value) into a collection.
- Collections can only hold object references.
- you have to *box* primitive values into the appropriate wrapper class (which is Integer in the case of int).
- When you take the object out of the collection, you get the Integer that you put in.
- if you need an int, you must *unbox* the Integer using the intValue method
- All of this boxing and unboxing is a pain, and clutters up your code

Auto Boxing and Unboxing

Before:

```
int i = 12;
ArrayList l = new ArayList();
l.add(new Integer(i));        ..... Explicit Boxing


Integr a=     l.get(index);
int b= a.intValue();        .....        Explicit Unboxing
```

Auto Boxing and Unboxing

Now:

- The auto boxing and unboxing feature automates the process, eliminating the pain and the clutter.

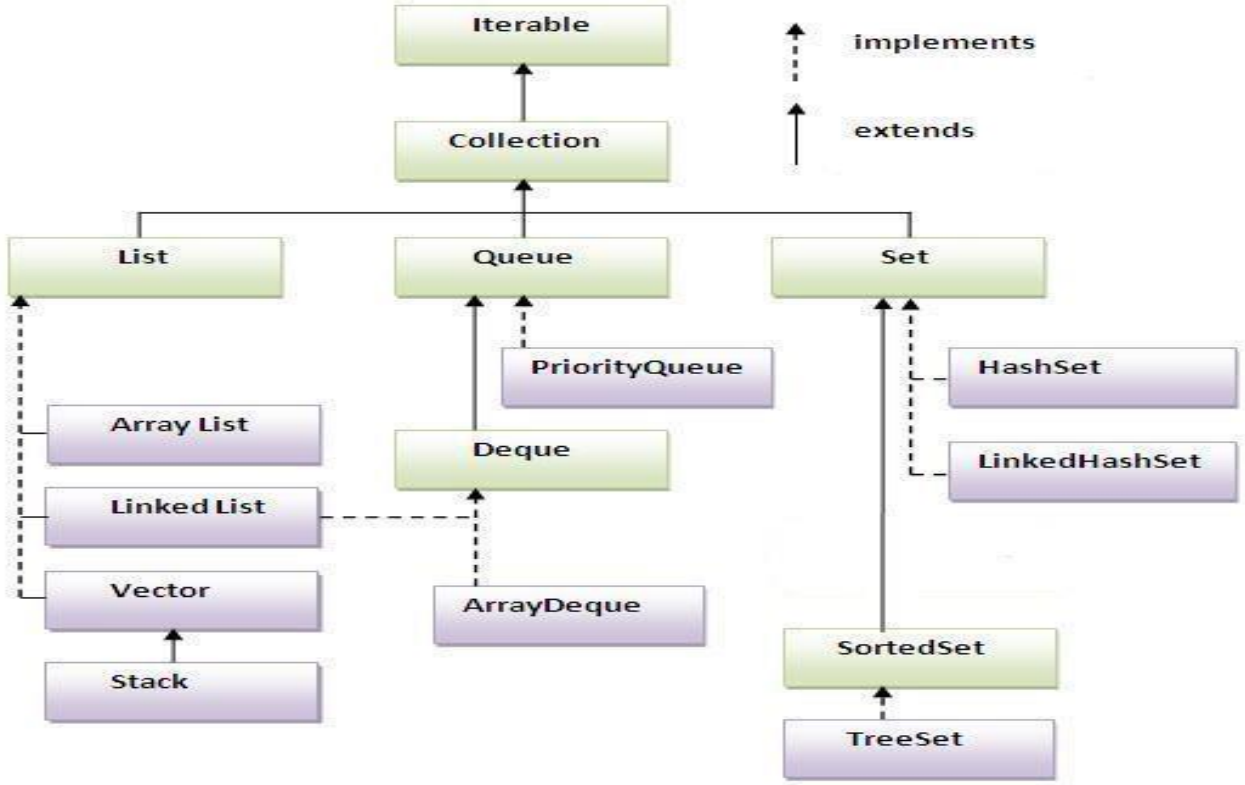
Example:

```
int i = 12;

ArrayList l = new ArrayList();

l.add(i);                  ..... Auto Boxing
int a =   l.get(index);    .....   Auto Unboxing
```

Need of Collection

- Java's collection classes provides a higher level interface than arrays.
- Arrays have a fixed size. Collections have a flexible size.
- Efficiently implementing a complicated data structures (e.g., hash tables) on top of raw arrays is a demanding task. The standard HashMap gives you that for free.
- There are different implementation you can choose from for the same set of services: ArrayList vs. LinkedList, HashMap vs. TreeMap.

Collection Hierarchy



Collection Interface

Collection is an interface in the java.util package, and as its name suggests, it is used to define a collection of objects.

Methods in Collection Interface:

```
boolean  add (Object o)
boolean addAll (Collection c)
void     clear()
boolean    contains ( Object o)
boolean    isEmpty()
Iterator iterator()
boolean remove (Object o)
int size()
```

List

- Its an ordered collection.
- Unlike sets, lists typically allow duplicate elements
- They typically allow multiple null elements if they allow null elements at all
- Example: ArrayList , LinkedList

List

```
import java.util.*;
public class ListDemo
{
    public static void main (String[] args)
    {
       List<String> friuts = new ArrayList<String>();
        friuts.add("Mango");
         friuts.add("Apple");
         friuts.add("StrawBerry");
         friuts.add("Orange");
         friuts.add("Kiwi");

        List<String> range = new ArrayList<String>();
        range = friuts.subList(2, 4);
          System.out.println(range);
    }
}
```

Set

Set : Its a Collection of ***unique*** elements

(i.e. a set contains no duplicate elements)
The elements are not ordered

Set can contain at most one null element
Example: HashSet

Sorted Set: It's a sub interface of Set. It further guarantees that its iterator will traverse the set in ascending element order, (sorted according to the *natural ordering* of its elements)

Example: TreeSet

Set

```
import java.util.*;
public class SetExample {
public static void main(String[] args) {
        Set set = new HashSet();
        set.add("one");
        set.add("second");
        set.add("3rd");
        set.add(new Integer(4));
        set.add(new Float(5.0F));
        set.add("second"); set.add(new Integer(4));
        System.out.println(set);
       }
 }
```

What will be the Output ?

Iterating Collection – Iterator Interface

- Allows the user to visit the elements of a collection one by one
- It has three methods:

```
boolean hasNext()
Object  next()
void    remove()
```

Code Example:

```
List list = new ArrayList();
    // add some elements
Iterator elements = list.iterator();
while ( elements.hasNext() ) {
    System.out.println(elements.next());
}
```

Sorting In Collection

- **sort()** method is present in java.util.Collections class which is used to sort any (Predefined Type) List in ascending order.
- Example –

  Let us suppose that our list contains

  {"Hello", "Friends", "Dear", "Is", "Superb"}

  After using Collection.sort(), we obtain a sorted list as

  {"Dear", "Friends", "Hello", "Is", "Superb"}

- For applying sort method on User Defined Types Collection we require to implement **Comparable or Comparator** interface.

Comparable Vs Comparator

- **Comparable lets a class implement its own comparison**
  - It's in the same class (it is often an advantage)
  - There can be only one implementation (so you can't use that if you want two different sorting cases)

- **Comparator is an external comparison**
  - It is typically in a unique instance (either in the same class or in another place)
  - You name each implementation with the way you want to sort things.
  - You can provide comparators for classes that you do not control.

Comparable Vs Comparator

- To implement Comparable interface, class must implement a single method compareTo()

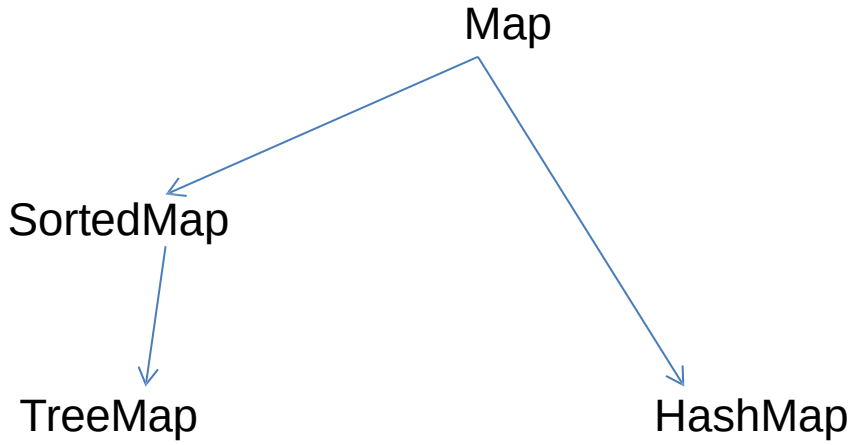
  public int compareTo(Object o){}

- To implement Comparator interface, class must implement a single method compare()

  public int compare (Object obj1 , Object obj2){}

Map

- An object that maps keys to values.
- A map cannot contain duplicate keys
- Each key can map to at most one value.
- The Map interface provides three *collection views*, which allow a map's contents to be viewed as:
  - a set of keys
  - a collection of values
  - a set of key-value mappings



Examples : HashMap,TreeMap

Map Demo

```
Class MapDemo{
        p.s.v.main(String args[]){

        HashMap hm=new HashMap();
        hm.put("Let us c",300);
        hm.put("Let us c++",400);
        hm.put("Thinking in Java",350);
   Iterator itr=hm.entrySet().iterator();
  while(itr.hasNext())
   {
      System.out.println(itr.next());
   }
}
} Output Generated: Let us c=300 Thinking in Java=350 Let us c++=400
```

Stream API – Java 8 Feature

- The Stream API is used to process collections of objects.
- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined.
- **Following are the different operations On Streams-**

Intermediate Operations:

map
filter
sorted

Terminal Operations:

collect
forEach
reduce

Stream API – Java 8 Feature

map :

```
List number = Arrays.asList(2,3,4,5);

List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

filter :

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

sorted:

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());
```

Stream API – Java 8 Feature

collect:

```
List number = Arrays.asList(2,3,4,5,3);

Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

forEach:

```
List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

reduce:

```
List number = Arrays.asList(2,3,4,5);

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

CYBAGE
Delivering Value. Scientifically.

Thank You!

www.cybage.com