Collections & Generics

Authored by : Sangeeta Joshi ***Presented by: Sangeeta Joshi***

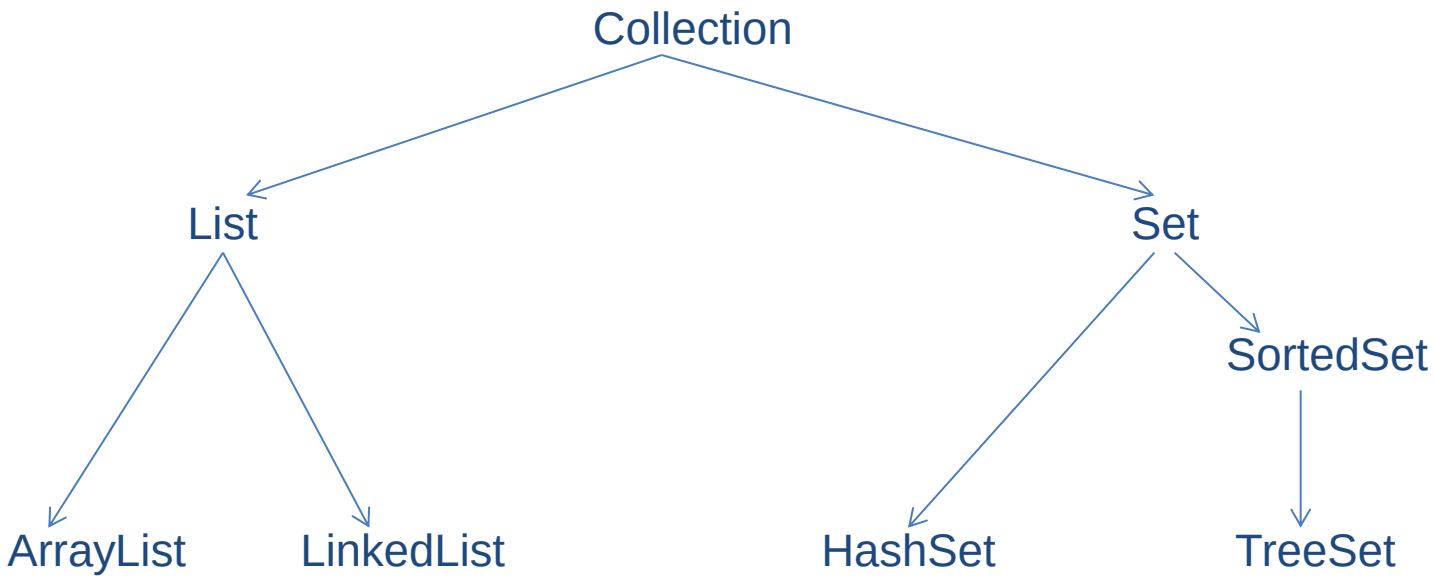
Agenda

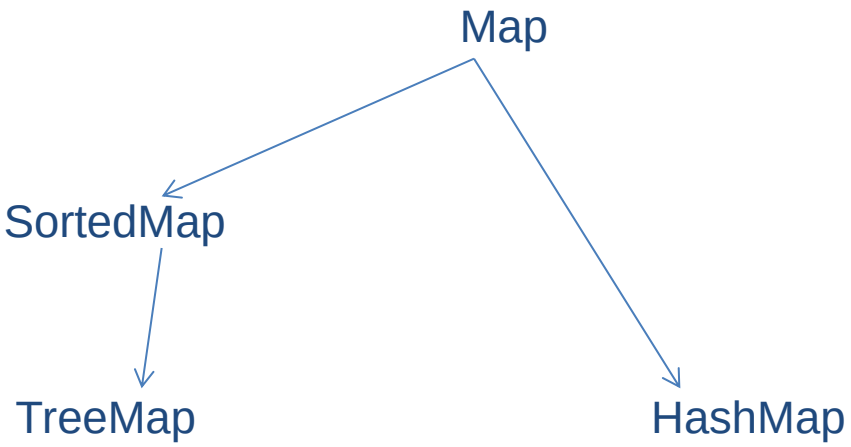
- Collection
- Set
- List
- Map
- Iterator

Collection : It is an object that represents ***a group of objects***

Collections framework : a unified architecture for representing and manipulating collections

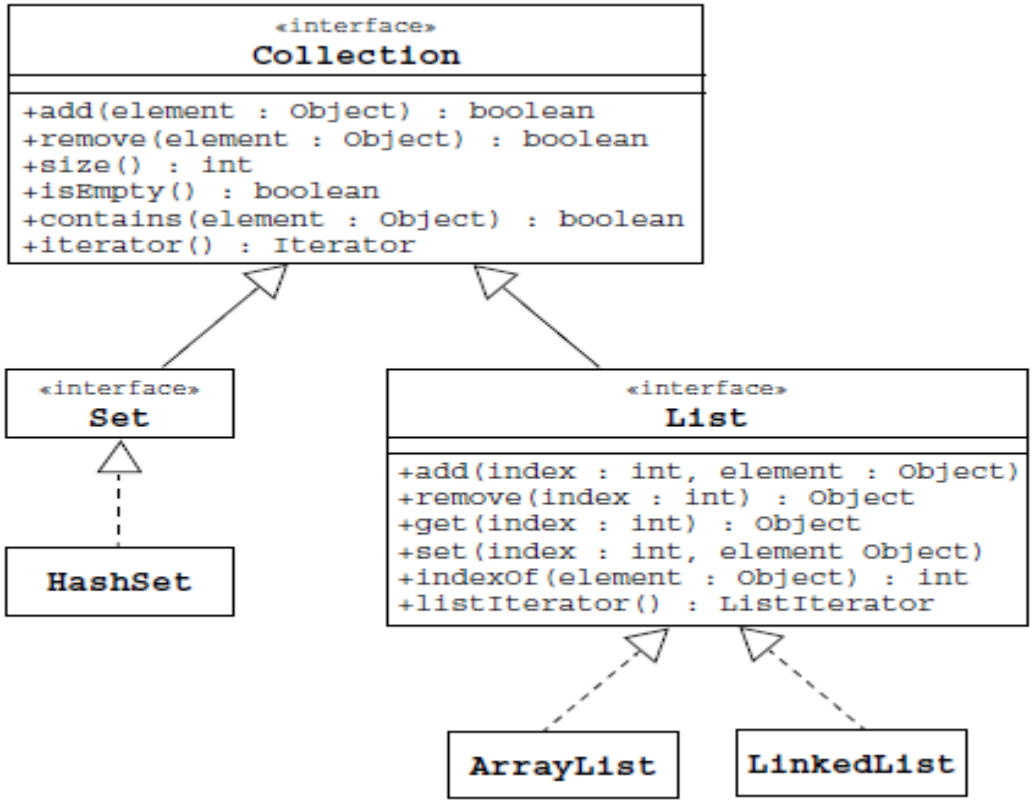
Most basic interfaces : Collection , Map , Iterator





Collections

The Collections API



Collections

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()

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"); // duplicate, not added
 set.add(new Integer(4)); // duplicate, not added
 System.out.println(set); } }
```

The output generated from this program is:

[one, second, 5.0, 3rd, 4]

List

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 ListExample {
public static void main(String[] args) {
List list = new ArrayList();
list.add("one");
list.add("second");
list.add("3rd");
list.add(new Integer(4));
list.add(new Float(5.0F));
list.add("second"); // duplicate, is added
list.add(new Integer(4)); // duplicate, is added
System.out.println(list); } }
```

The output generated from this program is:

[one, second, 3rd, 4, 5.0, second, 4]

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

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());
}
```

Generics

The Problem: (Pre-J2SE 5.0) Code is not Type Safe

Suppose you want to maintain String entries in a Vector. By mistake, you add an Integer element. Compiler does not detect this. This is not type safe code.

```
Vector v = new Vector();
v.add(new String("valid string"); // intended
v.add(new Integer(4));            // unintended
  // ClassCastException occurs during runtime
String s = (String)v.get(0);
```

Generics

Definition of the Generic List interface

```
interface List<E>
   {
   void add(E x);
   Iterator<E> iterator();
    }
```

Invocation (or usage) of List interface with concrete type parameter, String

```
List<String> ls = new ArrayList<String>(10);
```

Generics

- Problem: Collection element types
  - Compiler is unable to verify types
  - Assignment must have type casting
  - ClassCastException can occur during runtime
- Solution: Generics
  - Tell compiler the ‘type’ of the collection
  - Let the compiler fill in the cast
  - Example: Compiler will check if you are adding Integer type entry to a String type collection (compile time detection of type mismatch)

Using Generic Classes: 1

- Instantiate a generic class to create type specific object
- In J2SE 5.0, all collection classes are rewritten to be generic classes

```
Vector<String> vs = new Vector<String>();
vs.add(new Integer(5)); // Compile error!
vs.add(new String("hello"));
String s = vs.get(0); // No casting needed
```

Generics

- Generics provides abstraction over Types

  Classes, interfaces and methods can be parameterized by types (in the same way a Java type is parameterized by an instance of it)

- Generics makes type safe code possible

  If it compiles without any errors or warnings, then it must not raise any unexpected ClassCastException during runtime

- Generics provides increased readability

Definition and Usage of Generic Class

- Definitions:

  LinkedList<E> has a type parameter E that represents the type of the elements stored in the list

- Usage:

  Replace type parameter <E> with concrete type argument, like <Integer> or <MyType>

  LinkedList<Integer> can store only Integer or sub-type of Integer as elements

```
LinkedList<Integer> li =new LinkedList<Integer>();
li.add(new Integer(0));
Integer i = li.iterator().next();
```

Using Generic Classes: 2

- Generic class can have multiple type parameters
- Type argument can be a custom type

```
HashMap<String, Mammal> map =
        new HashMap<String, Mammal>();
        map.put("wombat", new Mammal("wombat"));
        Mammal w = map.get("wombat");
```

Type-safe Code Again

- The compiler guarantees that either:

  the code it generates will be type-correct at run time,
  or
  it will output a warning (using Raw type) at compile time

- If your code compiles without warnings and has no casts, then you will never get a ClassCastException.
  This is ***"type safe"*** code

Assignments

Any Questions?