Core Java

Collections

Lesson Objective

- The Collection API
- Collection Interface
- Generics
- Enhanced For loop
- Auto-boxing with Collections
- Implementing classes
- The Legacy Classes and Interfaces
- Arrays and Collections

Collections Framework

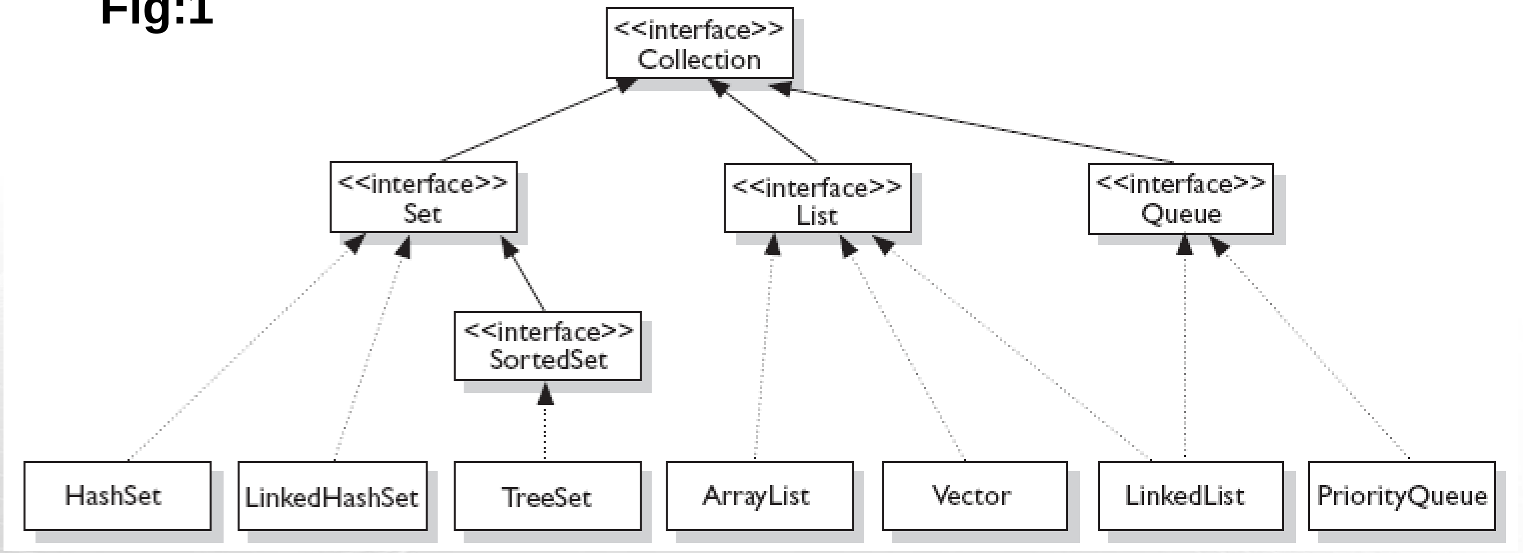
- A Collection is a group of objects.
- Collections framework provides a a set of standard utility classes to manage collections.
- Collection is used to store, retrieve objects, and to transmit them from one method to another.
- Collections Framework consists of three parts:
  - Core Interfaces
  - Concrete Implementation
  - Algorithms such as searching and sorting

Advantages of Collections

- **Reduces programming effort** by providing useful data structures and algorithms so you do not have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

Interfaces and Implementations

Fig:1



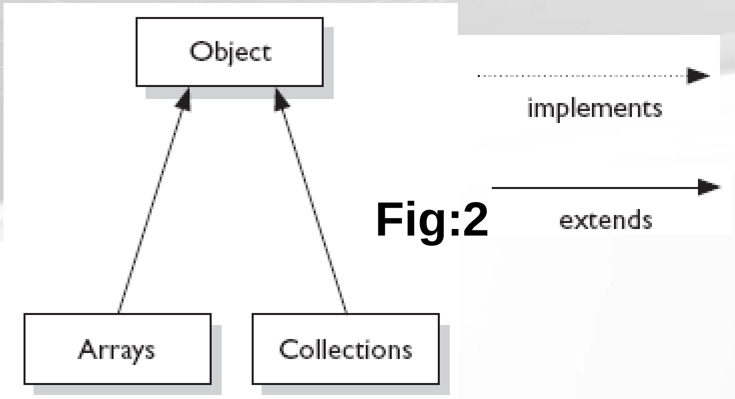
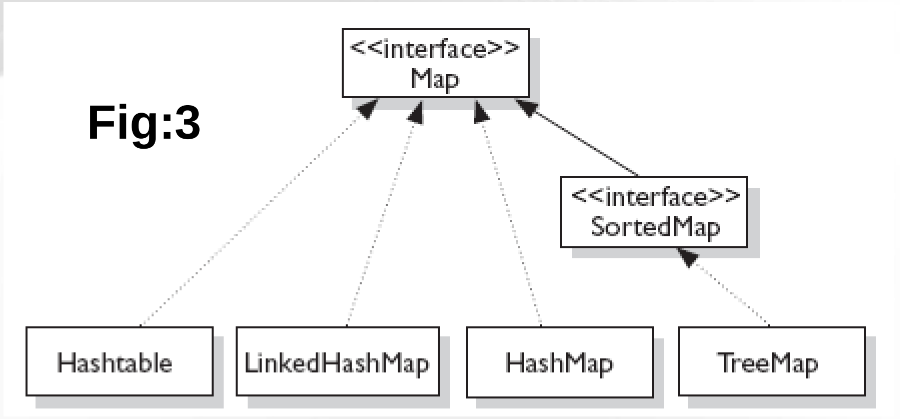


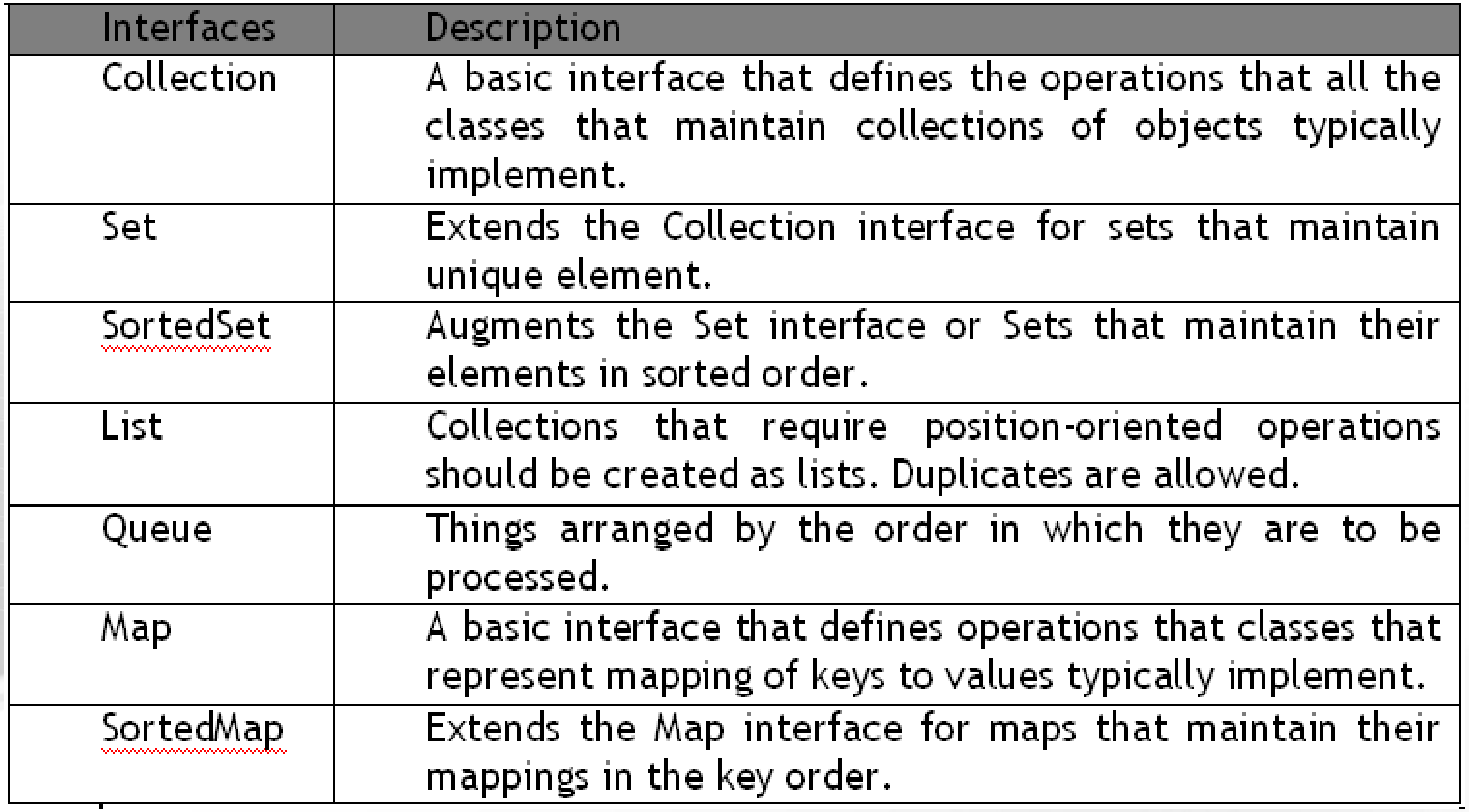
Fig:2

Fig:3



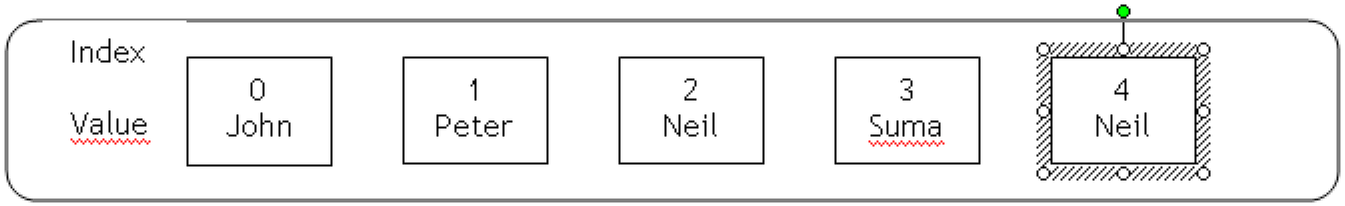
Collection Interfaces

- Let us discuss some of the collection interfaces:

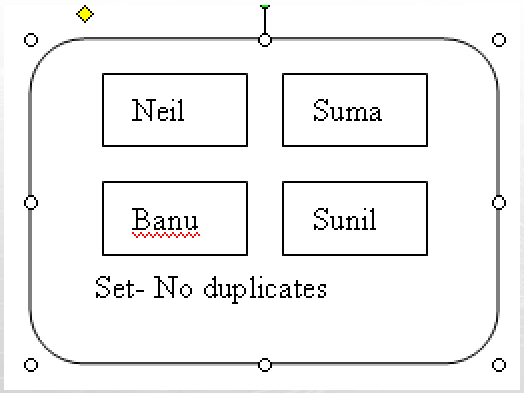


Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

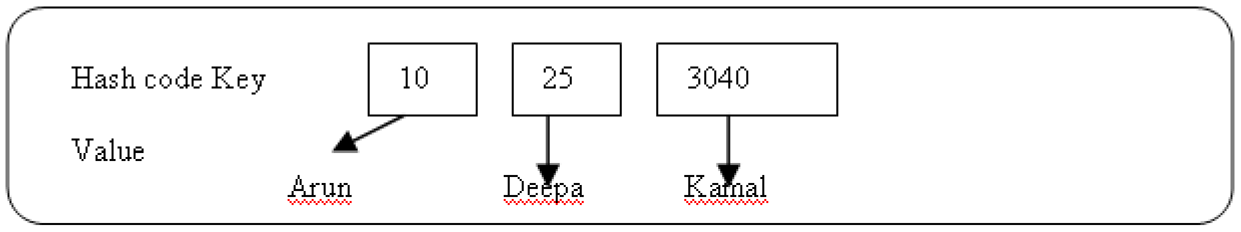
Illustration of List, Set, and Map



List - Duplicates allowed

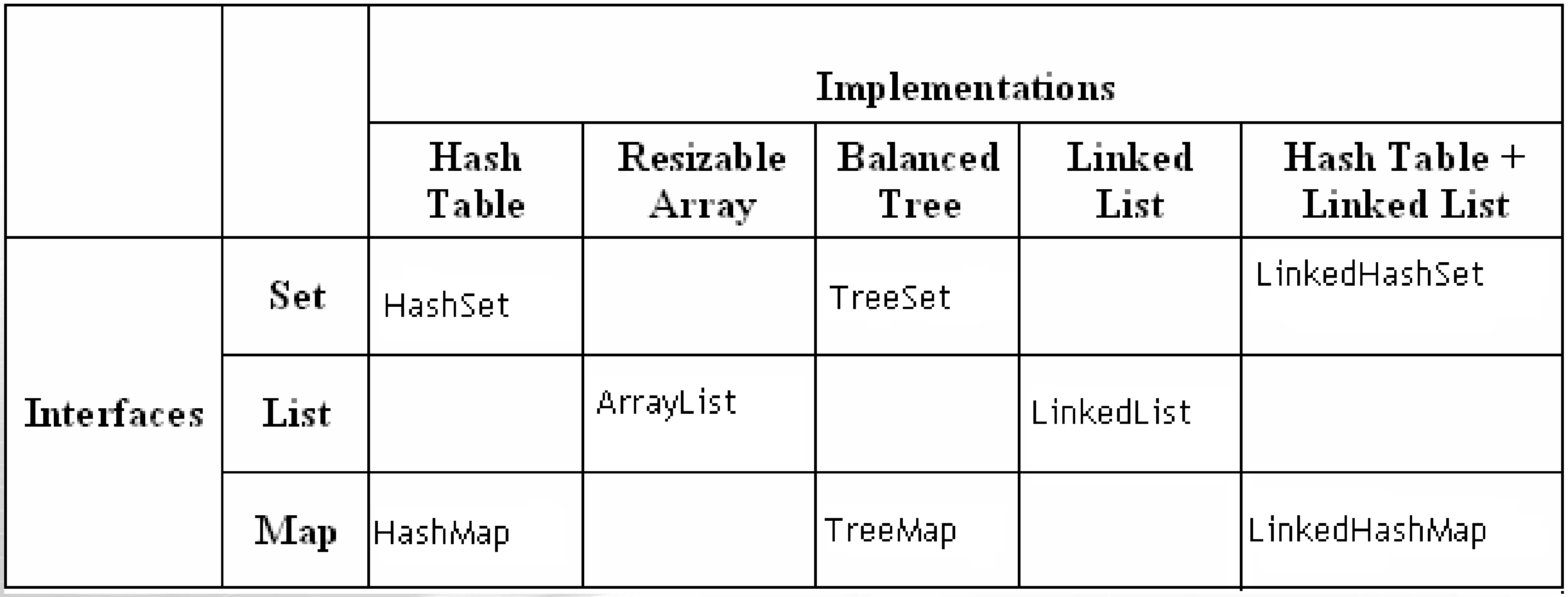


Set- No duplicates



Hash Map - Key generated from RollNo

Collection Implementations



		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Collection Implementations

- **HashSet:** A HashSet is an unsorted, unordered Set.

  It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation is, the better access performance you will get.

  Use this class when you want a collection with no duplicates and you do not care about order when you iterate through it.

- **LinkedHashSet:** A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

  Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

Collection Implementations

- **TreeSet**: A TreeSet stores objects in a sorted sequence. It stores its elements in a tree and they are automatically arranged in a sorted order.

- **ArrayList**: Think of this as a **growable** array. It gives you fast iteration and fast random access. It is an ordered collection (by index).

  However, it is not sorted. ArrayList now implements the new RandomAccess interface — a marker interface (meaning it has no methods) that says, "this list supports fast (generally constant time) random access."

  Choose this over a **LinkedList** when you need fast iteration but are not as likely to be doing a lot of insertion and deletion.

- **LinkedList**: A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.

Collection Implementations

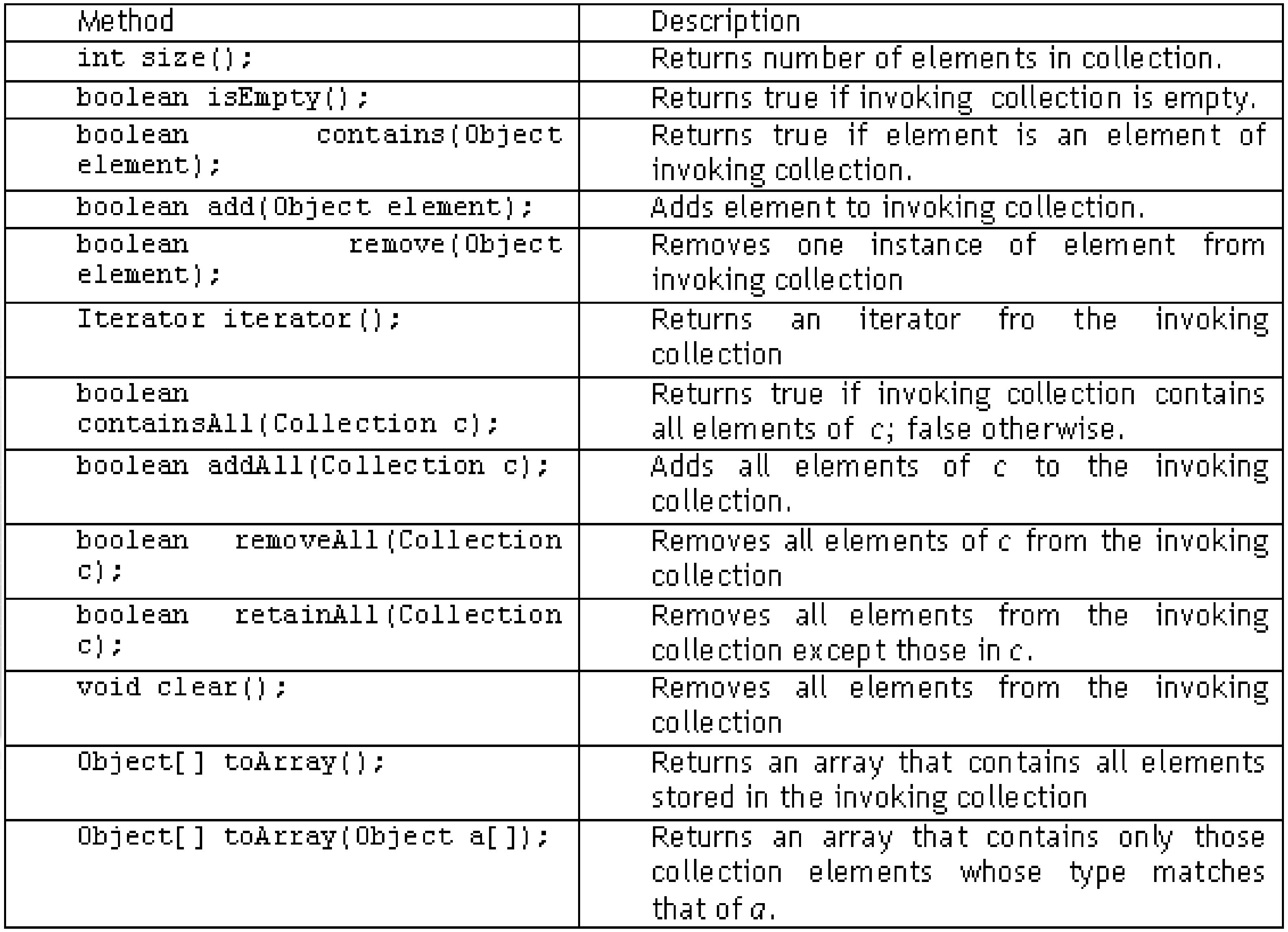
- **HashMap:** The HashMap gives you an unsorted, unordered Map. When you need a Map and you do not care about the order (when you iterate through it), then HashMap is the way to go.

  The other maps add a little more overhead. Where the keys land in the Map is based on the key's hashcode. So, like HashSet, the more efficient your hashCode() implementation, the better access performance you will get. HashMap allows one null key and multiple null values in a collection.

- **TreeMap:** TreeMap is a sorted Map.

- **LinkedHashMap:** Like its Set counterpart, LinkedHashSet, the LinkedHash-Map collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

Collection Interface methods



Method	Description
`int size();`	Returns number of elements in collection.
`boolean isEmpty();`	Returns true if invoking collection is empty.
`boolean contains(Object element);`	Returns true if element is an element of invoking collection.
`boolean add(Object element);`	Adds element to invoking collection.
`boolean remove(Object element);`	Removes one instance of element from invoking collection
`Iterator iterator();`	Returns an iterator fro the invoking collection
`boolean containsAll(Collection c);`	Returns true if invoking collection contains all elements of *c*; false otherwise.
`boolean addAll(Collection c);`	Adds all elements of *c* to the invoking collection.
`boolean removeAll(Collection c);`	Removes all elements of *c* from the invoking collection
`boolean retainAll(Collection c);`	Removes all elements from the invoking collection except those in *c*.
`void clear();`	Removes all elements from the invoking collection
`Object[] toArray();`	Returns an array that contains all elements stored in the invoking collection
`Object[] toArray(Object a[]);`	Returns an array that contains only those collection elements whose type matches that of *a*.

Demo :ArrayListDemo

Demo:

ArrayListSemo.java

ArrayListDemo.java

Iterators

- **Iterator** is an object that enables you to traverse through a collection.
- It can be used to remove elements from the collection selectively, if desired.

```
public interface Iterator<E>
 {
 boolean hasNext();     //returns true if there are more elements
 E next();              // returns next element
void remove();           // remove element from collection
 }
```

Demo :Iterators

Demo:

ItTest.java

ItTest.java

Comparator Interface

- The **java.util.Comparator** interface can be used to sort the elements of an Array or a list in the required way.

- It gives you the capability to sort a given collection in any number of different ways.

- Methods defined in Comparator Interface are as follows:
  - int compare(Object o1, Object o2)
    - *obj1* and *obj2* are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned.
  - boolean equals(Object obj)
    - *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**.

Demo :Comparator

Demo:

ComparatorExample.java

ComparatorExample.java

Generics

- **Generics** is a mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type.

  - Before Generics:

```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer x = (Integer) myIntegerList.iterator().next(); // 3
```

    - **Note:** Line no 3 if not properly typecasted will throw runtime exception

  - After Generics:

```
List<Integer> myIntegerList = new LinkedList<Integer>(); // 1
myIntegerList.add(new Integer(0)); //2
Integer x = myIntegerList.iterator().next(); // 3
```

Why Generics?

- Problem: Collection element types:
  - Compiler is unable to verify types.
  - Assignment must have type casting.
  - ClassCastException can occur during runtime.

- Solution: Generics
  - Tell the compiler 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 Generics

- You can instantiate a generic class to create type specific object.
  - In J2SE 5.0, all collection classes are rewritten to be generic classes.
  - Example:

```
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
```

Using Generics

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

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

Enhanced for loop

- Iterating over collections looks cluttered:

```
void printAll(Collection<emp> e) {
 for (Iterator<emp> i = e.iterator(); i.hasNext(); )
 System.out.println(i.next()); } }
```

- Using enhanced **for loop**, we can do the same thing as:

```
void printAll(Collection<emp> e) {
 for (emp t : e) )
 System.out.println(t); }}
```

  - When you see the colon (:) read it as “in.”
  - The loop above reads as “for each emp t in e.”

- The enhanced for loop can be used for both Arrays and Collections

Implementing Classes

- ArryaList
- HashSet
- TreeSet
- HashMap

ArrayList Class

- An ArrayList Class can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.
- It gives faster Iteration and fast random access.
- It uses Ordered Collection (by index), but not Sorted.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Demo : ArrayList Class

Demo:

ArrayListTest.java

ArrayListDemo.java

HashSet Class

- HashSet Class does not allow duplicates.
- A HashSet is an unsorted, unordered Set.
- It can be used when you want a collection with no duplicates and you do not care about the order when you iterate through it.

Demo : HashSet Class

Demo:

SetTest.java

SetTest.java

TreeSet Class

- TreeSet does not allow duplicates.
- It iterates in sorted order.
- Sorted Collection:
  - By default elements will be in ascending order.
- Not synchronized:
  - If more than one thread wants to access it at the same time, then it must be synchronized externally.

Demo : TreeSet Class

Demo:

TreesetDemo.java

TreesetDemo.java

HashMap Class

- Map is an object that stores ***key/value*** pairs.
- Given a key, you can find its value. Keys must be unique, however values may be duplicated
- HashMap uses the hashcode value of an object to determine how the object should be stored in the collection.
- Hashcode is used again to help locate the object in the collection.
- HashMap gives you an unsorted and unordered Map.
- It allows one null key and multiple null values in a collection.

Demo : HashMap Class

Demo:

HashMapDemo.java

HashMapDemo.java

Vector Class

- The **java.util.Vector** class implements a growable array of Objects.
- It is same as ArrayList. However, Vector methods are synchronized for thread safety.
- New java.util.Vector is implemented from List Interface.
- Creation of a Vector:
  - Vector v1 = new Vector(); // allows old or new methods
  - List v2 = new Vector(); // allows only the new (List) methods.

Demo : Vector Class

Demo:

VectorDemo.java

VectorDemo.java

Hashtable Class

- It is a part of **java.util package**.
- It implements a hashtable, which maps keys to values.
  - Any non-null object can be used as a key or as a value.
  - The Objects used as keys must implement the **hashcode** and the **equals method**.
- Synchronized class
- The Hashtable class only stores objects that override the **hashCode()** and **equals()** methods that are defined by Object.

Demo : HashTable Class

Demo:

HashTableDemo.java

HashTableDemo.java

Arrays Class

- The java.util.Arrays class is basically a set of static methods that are all useful for working with arrays.
- The Arrays class contains various methods for manipulating arrays such as sorting and searching
- In addition to that, it has got many utility methods for using with arrays such as a method for viewing arrays as lists and methods for printing the contents of an array, whatever be the dimension of the array.

Demo : Arrays Class

Demo:

ArraysMethodTester.java

ArraysMethodTester.java

Collections Class

- The java.util.Collections class is basically a set of static methods which operates on Collection.
- The Collections class contains various methods for manipulating collections such as reverse, sort, max, min, binarySearch etc.

Demo : Collections Class

Demo:

CollectionsDemo.java

CollectionsDemo.java

Common Best Practices on Collections

- Let us discuss some of the best practices on Collections:
  - Use for-each liberally.
  - Presize collection objects.
  - Note that Vector and HashTable is costly.
  - Note that LinkedList is the worst performer.
  - Choose the right Collection.
  - Note that adding objects at the beginning of the collections is considerably slower than adding at the end.
  - Encapsulate collections.
  - Use thread safe collections when needed.

Collection Best Practices.txt