Core Java 8 and Development Tools

Lesson 11: Collection

Lesson Objectives

- After completing this lesson, participants will be able to
  - Understand collection framework
  - Implement and use collection classes
  - Iterate collections
  - Create collection of user defined type

Collections Framework

- A Collection is a group of objects.
- Collections framework provides a set of standard utility classes to manage collections.
- Collections Framework consists of three parts:
  - Core Interfaces
  - Concrete Implementation
  - Algorithms such as searching and sorting

Advantages of Collections

- Collections provide the following advantages:
  - Reduces programming effort
  - Increases performance
  - Provides interoperability between unrelated APIs
  - Reduces the effort required to learn APIs
  - Reduces the effort required to design and implement APIs
  - Fosters Software reuse

Concept of Interfaces and Implementation

Fig:1

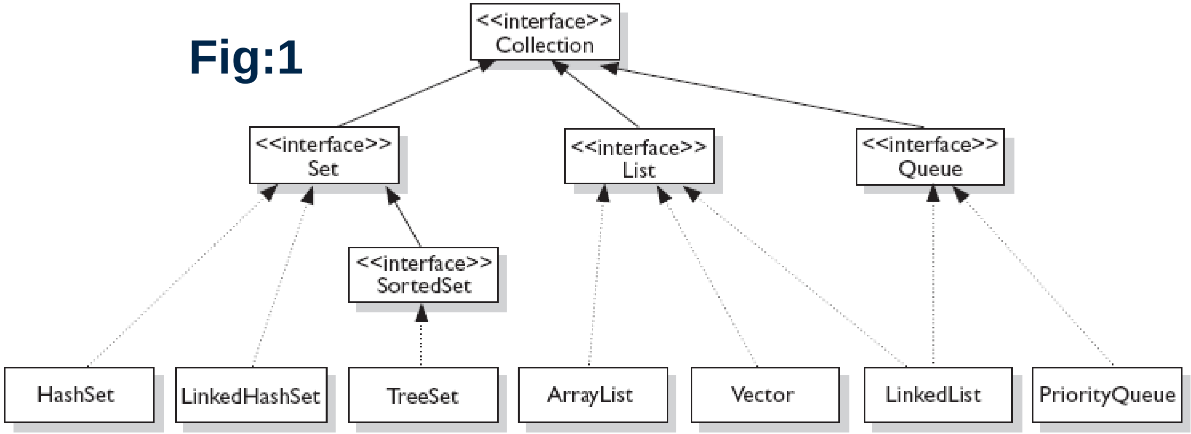


Fig:2

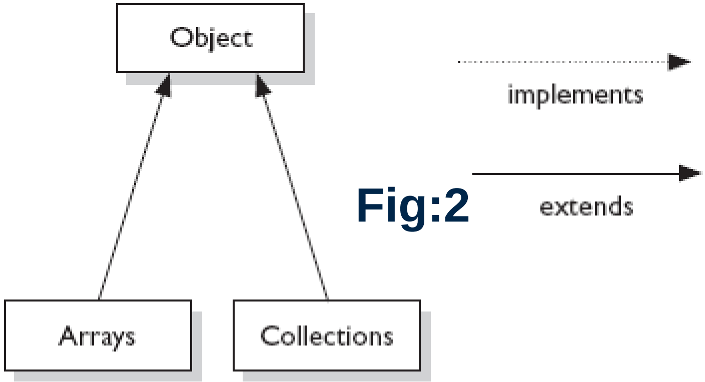
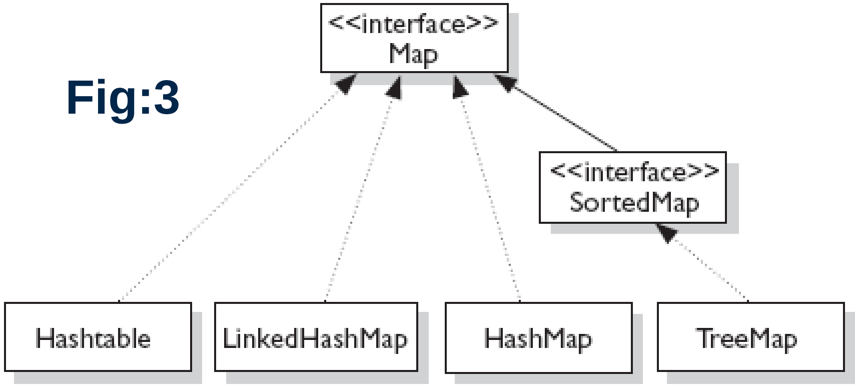
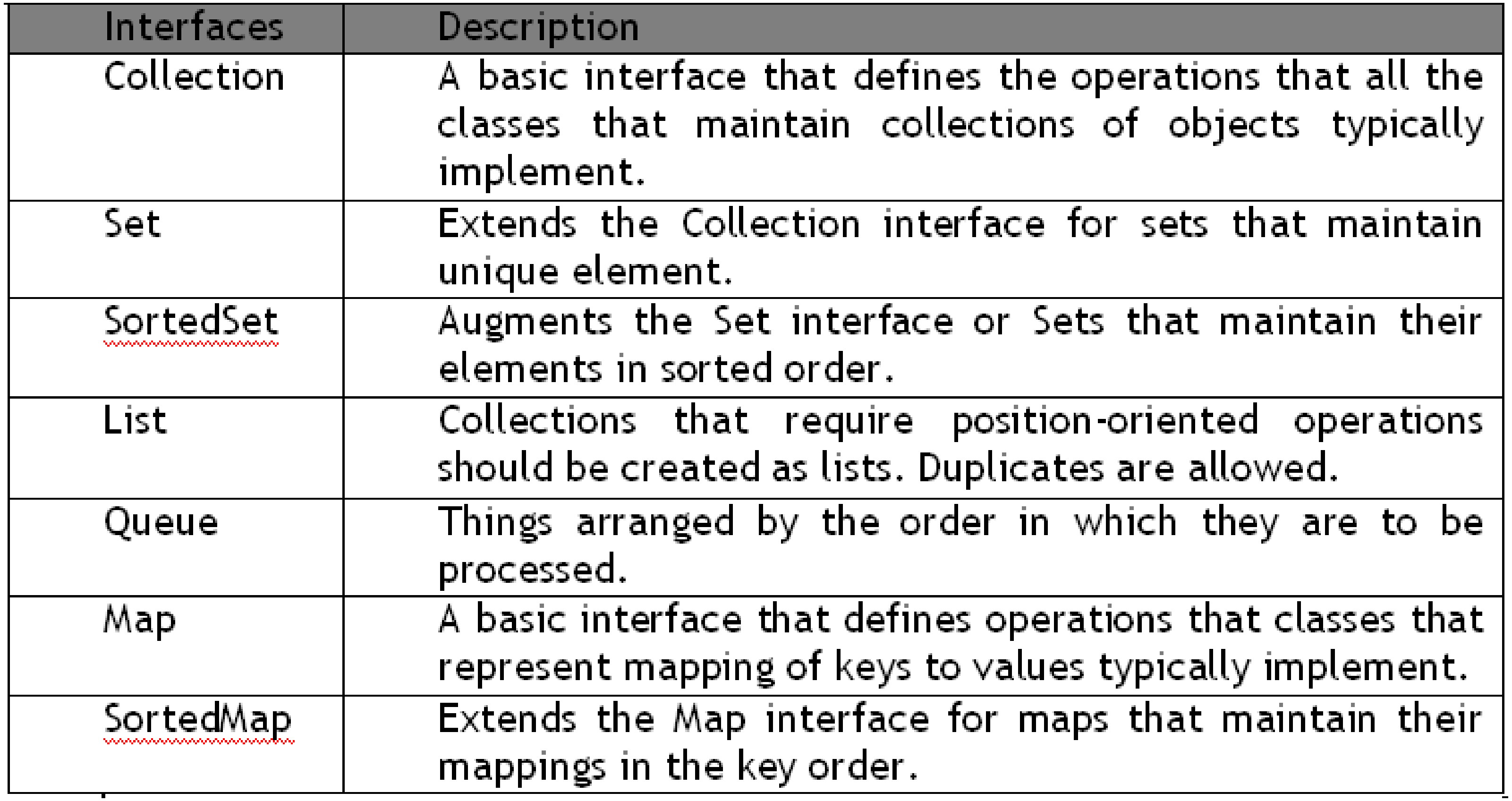


Fig:3



Collection Interfaces

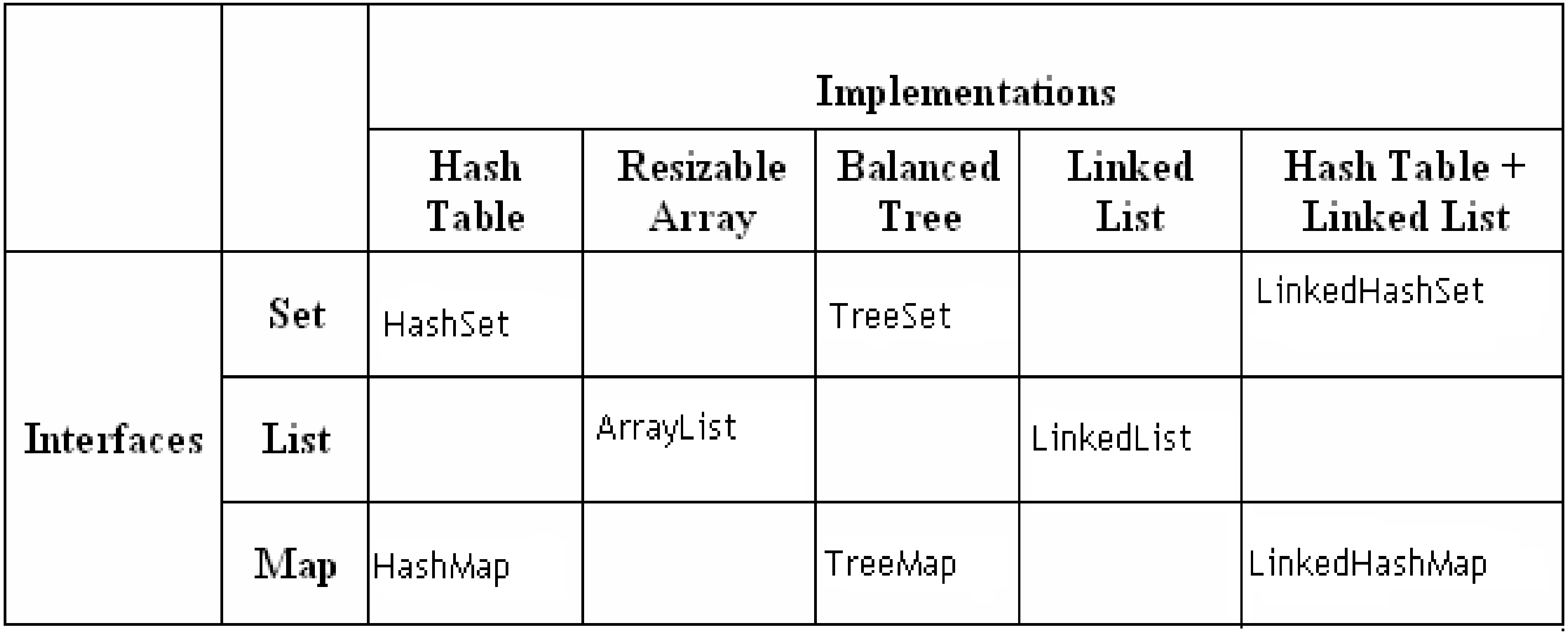
- Let us discuss some of the collection interfaces:



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.

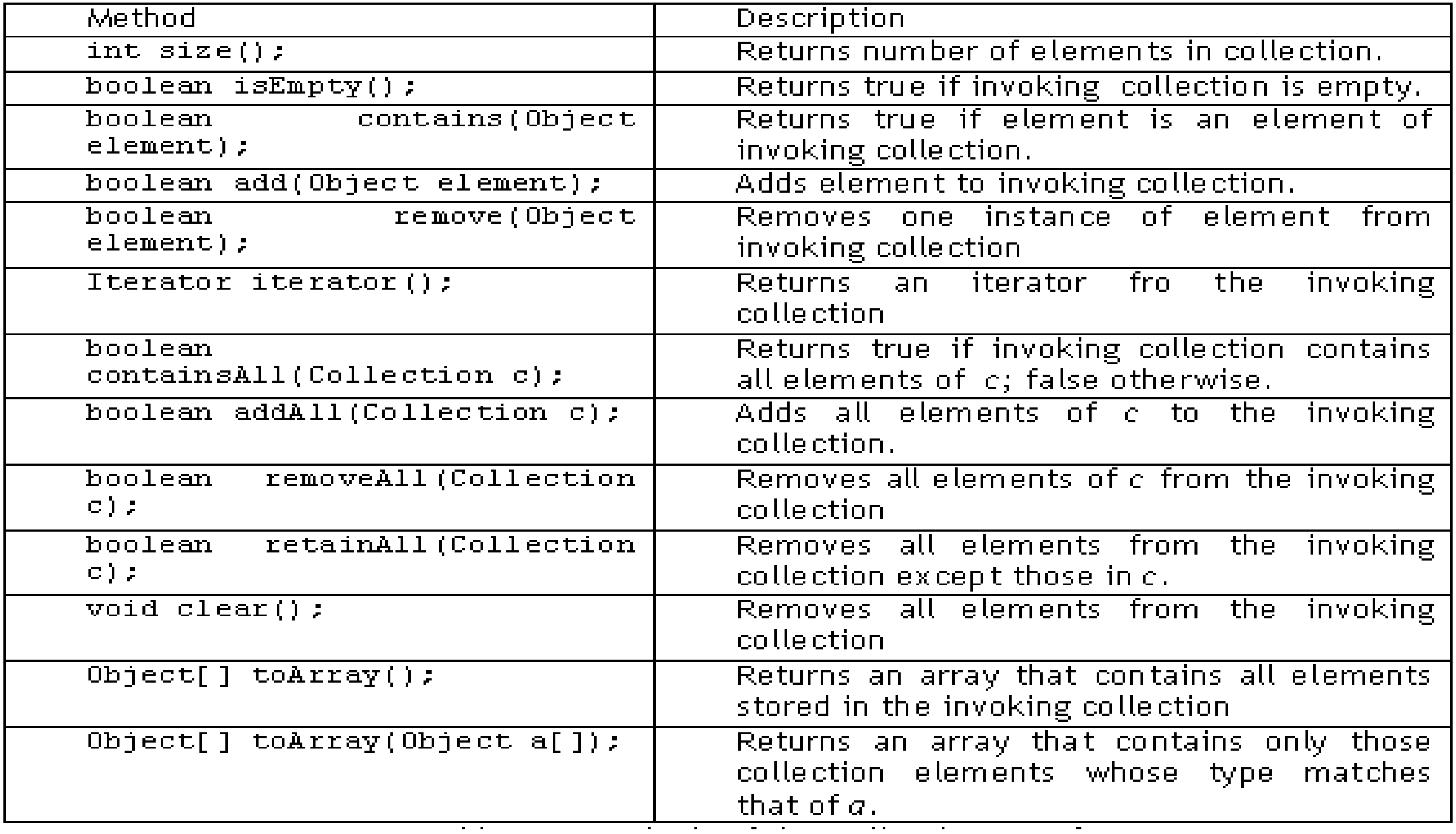
Collection Implementations

- 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 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*.

AutoBoxing with Collections

- Boxing conversion converts primitive values to objects of corresponding wrapper types.

```
int intVal = 11;
Integer iReference = new Integer(i); // prior to Java 5, explicit
Boxing
iReference = intVal;                    // In Java 5,Automatic Boxing
```

- Unboxing conversion converts objects of wrapper types to values of corresponding primitive types.

```
int intVal = iReference.intValue();   // prior to Java5, explicit
unboxing
intVal = iReference;                    // In Java 5, Automatic Unboxing
```

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: Array List Class

- Execute the ArrayListDemo.java program

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: Hash Set Class

- Execute the HashSetDemo.java program

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: Tree Set class

- Execute the Treeset.java program

HashMap Class

- 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

- Execute the HashMapDemo.java program

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.

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

Demo: Hash table Class

- Lesson-11 :-Execute the HashTableDemo.java program

Iterating through a collection

- 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();
 E next();
void remove();
 }
```

- Iterable is an superinterface of Collection interface, allows to iterate the elements using foreach method

```
Collection.forEach(Consumer<? super T> action)
```

Enhanced for loop

- Iterating over collections looks cluttered:

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

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

```
void printAll(Collection<Emp> employees) {
     for (Emp empObj : employees) )
        System.out.println( empObj ); }}
```

  - When you see the colon (:) read it as “in.”
  - The loop above reads as “for each emp ‘t’ in collection ‘e’.”

Demo :Concept of Iterators

- Execute: Lesson-11
  - MailList.java
  - ItTest.java program

Lab

- Lab 4: Collections

Best Practices

- 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.

Best Practices

- 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.

Summary

- The various Collection classes and Interfaces
- Generics
- Best practices in Collections

Review Questions

- Question 1: Consider the following code:

```
TreeSet map = new TreeSet();
    map.add("one");
    map.add("two");
    map.add("three");
    map.add("one");
    map.add("four");
    Iterator it = map.iterator();
    while (it.hasNext() )
        System.out.print( it.next() + " " );
```

- **Option 1:** Compilation fails
- **Option 2:** four three two one
- **Option 3:** one two three four
- **Option 4:** four one three two

Review Questions

- Question 2: Which of the following statements are true for the given code?

```
public static void before() {
   Set set = new TreeSet();
   set.add("2");
   set.add(3);
   set.add("1");
   Iterator it = set.iterator();
     while (it.hasNext())
           System.out.print(it.next() + " ");
}
```

- **Option 1:** The before() method will print 1 2
- **Option 2:** The before() method will print 1 2 3
- **Option 3:** The before() method will not compile.
- **Option 4:** The before() method will throw an exception at runtime.