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

This lesson discusses about collection framework in Java.

Lesson outline:

11.1: Collections Framework
11.2: Collection Interfaces
11.3: Implementing Classes
11.4: Iterating Collections
11.5: Comparable and Comparator
11.6: Best Practices

11.1: Collections Framework

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

Collections Framework:
A Collection (sometimes called a container) is an object that groups multiple elements into a single unit. Collection is used to store, retrieve objects, and to transmit them from one method to another.
The Collections API (also called the Collections framework) standardizes the way in which groups of objects are handled by your programs. It presents a set of standard utility classes to manage such collections. This framework is provided in the java.util package and comprises three main parts:

The core interfaces, which allow collections to be manipulated independent of their implementation. These interfaces define the common functionality exhibited by collections, and facilitate data exchange between collections.

A small set of implementations, which are concrete implementations of the core interfaces, providing data structures that a program can use. Eg LinkedLists, Arrays etc

An assortment of algorithms, which can be used to perform various operations on collections, such as sorting & searching.

The collection classes are the fundamental building blocks of the more complicated data structures used in the other Java packages in your own applications. There are several types of collections. They vary in storage mechanisms used, in the way they access data, and in the rules about what data may be stored.
Note: The Java Collection technology is similar to the Standard Template Library (STL) defined by C++.

11.1: Collections Framework

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

Collections Framework:

Advantages of Collections:

Collections provide the following advantages:

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. Since the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.

Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.

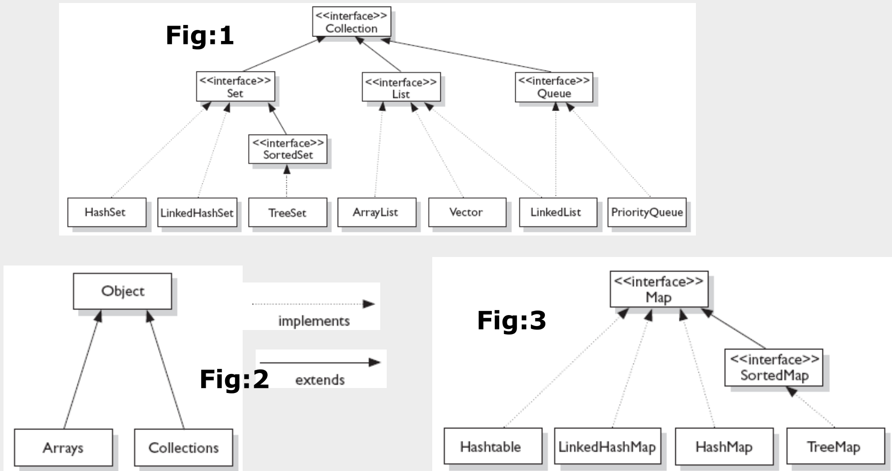
Reduces the effort required to learn APIs by eliminating the need to learn multiple ad hoc collection APIs.

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.

11.1: Collections Framework

Concept of Interfaces and Implementation



Interfaces and Implementation:
The core collection interfaces (shown in figure above) are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation.
Not all collections in the Collections Framework actually implement the Collection interface. Specifically, none of the Map-related classes and interfaces extend from Collection. So while SortedMap, Hashtable, HashMap, TreeMap, and LinkedHashMap are all thought of as collections, none are actually extended from Collection.
Note: Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including add(), remove(), contains(), size(), and iterator().

11.2: Collection Interfaces
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.

Interfaces and Implementation:
Collection Interfaces:
Following are the four major interfaces:

Set Interface: holds only unique values and rejects duplicates.

List Interface: represents an ordered list of objects, meaning the elements of a List can be accessed in a specific order, and by an index too. List can hold duplicates.

Queue Interface: represents an ordered list of objects just like a List. However, a queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue. Just like a queue in a supermarket!

Map Interface: represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. A Map cannot contain duplicate keys; each key can map to at most one value. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys.

SortedSet Interface: is a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering.

SortedMap Interface: is a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

11.2: Collection Interfaces
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 Implementations:
The Java Collections Framework provides several general-purpose implementations of the Set, List, and Map interfaces. The general purpose implementations are summarized in the table above.

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. Implements the Set interface.

LinkedHashSet: differs from HashSet by guaranteeing that the order of the elements during iteration is the same as the order they were inserted into the LinkedHashSet.

TreeSet: implements the SortedSet interface. Like LinkedHashSet, TreeSet also guarantees the order of the elements when iterated, but the order is the sorting order of the elements. This order is determined either by their natural order (if they implement Comparable), or by a specific Comparator implementation.

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.

11.2: Collection Interfaces

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*.

Collection Interface Methods:
The Collection Interface is the foundation on which the collection framework is built. It declares the core methods that all collections will have. Some of these methods are summarized in the table given in the above slide.
The bulk operations perform some operation on an entire Collection in a single shot. They are done through the following methods, namely: containsAll(), addAll(), removeAll(), retainAll(), clear().

11.2: Collection Interfaces

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
```

AutoBoxing with Collections:
J2SE 5 adds to the Java language autoboxing and auto-unboxing.
Primitive types and their corresponding wrapper classes can now be used interchangeably. For example: The following lines of code are legitimate in Java 5:

```
int intval1 = 0;
Integer intval2 = intval1;
int intval3 = new Integer(intval2);
```

This is often referred to as automatic boxing or unboxing.
If an int is passed where an Integer is expected, then the compiler will automatically insert a call to the Integer constructor. Conversely, if an Integer is provided where an int is required, then there will be an automatic call to the IntValue method.
Autoboxing is the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.
Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from type wrapper when its value is needed.

11.3: Implementing Classes

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();
```

ArrayList Class:
Let us check the power of ArrayList with an example:

```
List<String> myList = new ArrayList<String>();
```

In many ways, ArrayList<String> is similar to a String[] in that it declares a container that can hold only Strings. However, it is more powerful than a String[]. Let us look at some of the capabilities that an ArrayList has:

```
import java.util.*;
    public class ArrayListTest {
      public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        String str = "hi";
        list.add("string");
        list.add(str);
        list.add(str + str);
        System.out.println(list.size());
        System.out.println(list.contains(42));
        System.out.println(list.contains("hihi"));
        list.remove("hi");
        System.out.println(list.size());
      }
    }
```

```
output :
3
false
true
2
```

11.3: Implementing Classes

Demo: Array List Class

Execute the ArrayListDemo.java program

11.3: Implementing Classes

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.

HashSet Class:
Remember that Sets are used when you do not want any duplicates in your collection. If you attempt to add an element to a set that already exists in the set, then the duplicate element will not be added, and the add() method will return false. Remember, HashSets tend to be very fast because they use hashcodes.

```
import java.util.*;
    class SetTest {
      public static void main(String[] args) {
        boolean[] boolArr = new boolean[5];
        Set<Integer> set = new HashSet<Integer>();
        boolArr[0] = set.add(1);
        boolArr[1] = set.add(2);
        boolArr[2] = set.add(3);
        boolArr[3] = set.add(4);
        boolArr[4] = set.add(5);
        for (Integer index : set)
            System.out.print(index + " ");
       }
    }
```

O/P: 2 4 1 3 5

Note: The order of the objects printed are not predictable

11.3: Implementing Classes

Demo: Hash Set Class

Execute the HashSetDemo.java program


```
import java.util.*;
class HashSetDemo {
  public static void main(String args[]) {
  // create a hash set
      HashSet hs = new HashSet();
      // add elements to the hash set
          hs.add("B");
          hs.add("A");
          hs.add("D");
          hs.add("E");
          hs.add("C");
          hs.add("F");
         System.out.println(hs);
     }
}
```

Output :
[D, A, F, C, B, E]

11.3: Implementing Classes

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.

TreeSet:

TreeSet implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements, or by the comparator provided at set creation time, depending on which constructor is used.

```
class TreeSetDemo {
 public static void main(String args[]) {
   TreeSet<String> treeSet = new TreeSet<String>();
   treeSet.add("One");
   treeSet.add("Two");
   treeSet.add("Three");
   treeSet.add("Four");
   treeSet.add("Five");
        System.out.println("Contents of treeset");
   Iterator iterator = treeSet.iterator(); // obtaining iterator object
   while (iterator.hasNext()) { // to iterate thru collection.
       Object object = iterator.next();
        System.out.print(object + "\t");
    }
  }}
```

O/P: Five Four One Three Two

11.3: Implementing Classes

Demo: Tree Set class

Execute the Treeset.java program

You can also refer to the

11.3: Implementing Classes

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.

HashMap Class:
Map is an object that stores key/value pairs. Given a key, you can find its value. Keys must be unique, values may be duplicated. The HashMap class implements the map interface. The HashMap class uses a hash table to implement Map interface.
The following example maps names to account balances.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String,Double> hm = new HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set set = hm.entrySet();     // Get a set of the entries
Iterator i = set.iterator();     // Get an iterator
    while(i.hasNext()) {          // Display elements
        Map.Entry me = (Map.Entry)i.next();
        System.out.println(me.getKey() + ": "+ me.getValue());
    }
// Deposit 1000 into John Doe's account
double balance =  ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " + hm.get("John Doe")); } }
```

11.3: Implementing Classes
Demo: HashMap Class

Execute the HashMapDemo.java program

The example is provided on previous page.
The output of the program is as follows:
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe's new balance: 4434.34
The above program first populates the HashMap object. Then the contents of the map are displayed using a set-view, obtained by calling entrySet(). The keys and values are displayed by calling getKey() and getValue() methods of the Map.Entry interface.

Note: TreeMap instead of HashMap will have given a sorted output.

11.3: Implementing Classes

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.

Vector Class:
Vectors (the java.util.Vector class) are commonly used instead of arrays. This is because they expand automatically when new data is added to them. The Java 2 Collections API introduced a similar ArrayList data structure.
ArrayLists are unsynchronized and therefore faster than Vectors. However, they are less secure in a multithreaded environment. The Vector class was changed in Java 2 to add the additional methods supported by ArrayList. The description below is for the (new) Vector class.
Vectors can hold only Objects and not primitive types (for example: int). If you want to put a primitive type in a Vector, put it inside an object (for example: to save an integer value use the Integer class or define your own class). If you use the Integer wrapper, you will not be able to change the integer value, so it is sometimes useful to define your own class.
Constructor summary:
Vector()
Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
Vector(Collection<? extends E> c)
Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
Vector(int initialCapacity)
Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
Vector(int initialCapacity, int capacityIncrement)
Constructs an empty vector with the specified initial capacity and capacity increment.

11.3: Implementing Classes

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

Hashtable Class:
The Hashtable was a part of the original java.util package.
Hashtable is synchronized, and stores a key/value pair using the hashing technique. While using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed. Subsequently, the resulting hash code is used as the index at which the value is stored within the table. The Hashtable class only stores objects that override the hashCode() and equals() methods that are defined by Object.

11.3: Implementing Classes

Demo: Hash table Class

Lesson-11 :-Execute the HashTableDemo.java program


```
import java.util.*;
class HashTableDemo {
  public static void main(String args[]) {
      Hashtable<String,Double> balance = new
                                    Hashtable<String,Double>();
      Enumeration names;
      String str;
      double bal;
      balance.put("Arun", new Double(3434.34));
      balance.put("Radha", new Double(123.22));
      balance.put("Ram", new Double(99.22));
      // Show all balances in hash table.
     names = balance.keys();
      while(names.hasMoreElements()) {
          str = (String) names.nextElement();
          System.out.println(str + ": " +
          balance.get(str));
      }
    // Deposit 1,000 into Zara's account
        bal = ((Double)balance.get("Ram")).doubleValue();
        balance.put("Ram", new Double(bal+1000));
        System.out.println("Ram's new balance: " +
        balance.get("Ram"));
  }
}
```

11.4: Iterating Collection

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)
```

Iterators:
Java provides two interfaces that define the methods by which you can access each element of a collection: Enumeration and Iterator.

Enumeration is a legacy interface and is considered obsolete for new code. It is now superseded by the iterator interface.

The iterator() method of every collection returns an iterator to a collection. It is similar to an Enumeration, but differs in two respects:

Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

Method names have been improved.

There is no safe way to remove elements from a collection while traversing it with an Enumeration.
In the example in the above slide, the following parameters are used:

boolean hasNext() : returns true if there are more elementsObject next() : It returns next element. It throws NoSuchElementException if there is no next element.

void remove() : It removes current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next()

Note 1: The hasNext() method is identical in function to Enumeration.hasMoreElements(), and the next() method is identical in function to Enumeration.nextElement().
Note 2: Iterator.remove() is the only safe way to modify a collection during iteration. The behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

11.4: Iterating Collection

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'."

Enhanced for loop:
The enhanced for loop can be used for both Arrays and Collections:

```
class Enhancedforloop {
  static void printArray(int intArr[]) {
       for (int arrayindex : intArr )
           System.out.println(arrayindex);
  }
  static void printCollection(ArrayList arrList) {
      for (Object object : arrList)
          System.out.println(object);
  }

public static void main(String arg[]) {
    int intArr[] = { 1, 2, 3, 4, 5 };
    printArray(intArr);
    ArrayList arraylist = new ArrayList();
    arraylist.add(10);
    arraylist.add(30);
    arraylist.add(20);
    printCollection(arraylist);
  }
}
```

11.4: Iterating Collection

Demo :Concept of Iterators

Execute: Lesson-11

- MailList.java
- ItTest.java program

Lab

Lab 4: Collections

11.5: Common Best Practices on 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.

Common Best Practices on Collections:
Use for-each liberally : When there is a choice, the for-each loop should be preferred over the for loop, since it increases legibility.
Presize collection objects.

This is necessary because whenever the collection size has reached the maximum, internally whole array is copied to a new array with new increased size. This takes considerable time.

Try to presize any collection object to be as big as it will need to be. It is better for the object to be slightly bigger than necessary than to be smaller. This recommendation really applies to collections that implement size increases in such a way that objects are discarded.

For example: Vector grows by creating a new larger internal array object, copying all the elements from and discarding the old array. Most collection implementations work similarly, so presizing a collection to its largest potential size reduces the number of objects discarded.

Vector and HashTable is costly.

Usage of vector is very costly especially in code which heavily uses Vector to store lots of elements. Avoid using that if the elements in it are of same type. This is because elements are stored as Object so while accessing them one has to cast them into relevant classes which is very costly. Use ArrayList instead.

HashTable has the same reason as in the case of Vector. Moreover, the problem is compounded because of the use of Key and Value. Use HashMap class instead.

Never use linked List while accessing the objects : Sequentially access the elements.

11.5: Common Best Practices on Collections

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.

Common Best Practices on Collections:

Choosing the right Collection:

We can select the appropriate collection based on the different implementation of the collection interfaces. As we know, there are different collection classes available such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, and so on.

Principal features of non-primary implementations:

HashMap has slightly better performance than LinkedHashMap. However, its iteration order is undefined.

HashSet has slightly better performance than LinkedHashSet. However its iteration order is undefined.

TreeSet is ordered and sorted, but slow.

TreeMap is ordered and sorted, but slow.

LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration.

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.