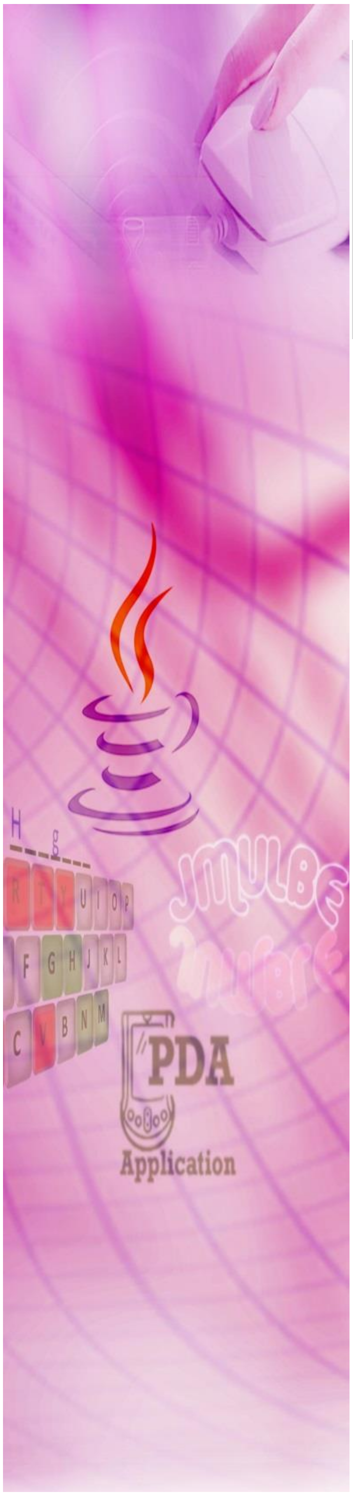

Session 4

Instructor Inputs

Session Overview

This session covers the Implementing Type-safety section of Chapter 3 and the following sections of Chapter 4 of the book, Programming in Java – SG:

- Using the Set Interface
- Using the List Interface
- Using the Map Interface

This session will help the students to learn how to implement type-safety. In addition, the students will become familiar with the various interfaces and classes to create a collection of objects.

Implementing Type-safety

Start the discussion by explaining to the students that the length of the code can be minimized by using the type inference feature of generics. Further, explain type inference in detail with the code given in the SG. Next, discuss how the diamond operator, <>, can be used to reduce the overhead task of specifying the type of object while declaring the generic object. In addition, while explaining this topic, you can discuss the code given under the Using a Generic Type Inference subtopic in the Additional Examples topic.

Thereafter, start the discussion on wildcards by explaining the need of passing an object of the class that is inherited from another class as a type parameter in the generic class or method with the help of the code given in the SG. Further, discuss the types of wildcard and explain the code of the `<? extends>` generic type. In addition, while explaining this topic, you can discuss the code given under the Using Wildcards subtopic in the Additional Examples topic.

Additional Examples

Using a Generic Type Inference

The following code demonstrates the example of type inference with the diamond operator:

```
class Triangle<T>
{
    private T side;

    public Triangle(T s)
    {
        side = s;
    }

    public T getSide()
    {
        return side;
    }
}

 class ProcessShape<K, V>
{
    private K keys;
    private V values;
```

```
        public ProcessShape(K k, V v)
        {
            keys = k;
            values = v;
        }

        public K getKey()
        {
            return keys;
        }

        public V getValues()
        {
            return values;
        }
    }

    public class UseProcessShape
    {
        public static void main(String[] args)
        {

            ProcessShape<String, Triangle<Integer>> obj = new
    ProcessShape<>("Key1", new Triangle<Integer>(10));
            System.out.println(obj.getKey() + " " + obj.getValues().getSide());
        }
    }
```

In the preceding code, there are two generic classes, `Triangle` and `ProcessShape`. The `Triangle` class is defined with one type parameter. On the other hand, `ProcessShape` is defined with two type parameters. In addition, the `UseProcessShape` class creates an object of the `ProcessShape` class by specifying the `String` and `Triangle<Integer>` arguments of the type parameter and by invoking the constructor with the diamond operator.

Using Wildcards

The following code demonstrates the use of the unbounded wildcard:

```
    class First
    {
        public String l1;

        public First(String a)
        {
            l1 = a;
        }

        public String getFirst()
        {
            return l1;
        }
    }
    class Second extends First
    {
        public String l2;
        public Second(String a)
        {
            super(a);
        }


    }
    class Third
    {
        public String l3;

        public Third(String a)
        {
            l3 = a;
        }

    }

    class Process<T>
    {
        public T level;

        public Process(T l)
        {
            level = l;
        }

    }

    public class WCDemo
    {
       public static void main(String[] args)
    {
        Process<?> pobj1 = new Process<>(new First("Race 1"));      //Base Class
    of Second Class
        Process<?> pobj2 = new Process<>(new Second("Race 2"));     //Sub Class
    of First Class
        Process<?> pobj3 = new Process<>(new Third("Race 2"));      //Standalone
    Class
        }
    }
```

In the preceding code, five classes, `First`, `Second`, `Third`, `Process`, and `WCDemo`, are created. The `Second` class inherits the `First` class. In addition, in the `WCDemo` class, three objects of the `Process` class are created. However, while creating the object, the unbounded type, `<?>`, wildcard is specified that can accept any object.

Activity 3.1: Working with Generics

Handling Tips

Discuss the problem statement with the students.

To perform the activity, 3.1, you need to use the **MessageProcessor.txt** and **MainClass.txt** files, which are provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 03\Activity 3.1\Input Files**

The solution files, **EmailMessage.java**, **MainClass.java**, **MessageProcessor.java**, and **SMSMessage.java**, for this activity are provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 03\Activity 3.1\Solution**

Using the Set Interface

Handling Tips

Start the discussion by explaining the scenario of the chat application given in the SG. Thereafter, inform the students that a collection of objects can be created by using the `Collection` interface. Further, tell the students that to ensure the uniqueness of the objects in the collection, the `Set` interface can be used. Thereafter, explain the class hierarchy of the `java.util` package that implements the `Set` interface along with the figure given in the SG. Next, explain the `Iterator` interface that enables the traversing of the objects in the set collection.

Further, explain the `HashSet` class that implements the `Set` interface and enables you to create the unsorted collection of objects. Next, discuss how an object of the `HashSet` class can be created by using the various constructors. Thereafter, explain the various methods of the `HashSet` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the HashSet Class subtopic in the Additional Inputs topic.

Next, explain the `TreeSet` class that implements the `Set` interface and enables you to create the sorted collection of objects. Thereafter, discuss how the object of the `TreeSet` class can be created by using the various constructors. Thereafter, explain the various methods of the `TreeSet` class along with the sample code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the TreeSet Class subtopic in the Additional Inputs topic.

Additional Inputs

Working with the HashSet Class

The following table lists some additional methods of the `HashSet` class with their description.

Method	***Description***
`boolean contains(Object o)`	*Is used to check whether the specified object is present in `HashSet` or not. It returns `true` if the object is present. Otherwise, it returns `false`.*
`boolean isEmpty()`	*Is used to check whether `HashSet` is empty or not. It returns `true` if there is no element. Otherwise, it returns `false`.*

The Methods of the HashSet Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.HashSet;

public class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet<String> obj = new HashSet<String>();
        String sobj1 = new String("Element 1");
        String sobj2 = new String("Element 2");
        String sobj3 = new String("Element 3");
        String sobj4 = new String("Element 4");

        obj.add(sobj1);
        obj.add(sobj2);
        obj.add(sobj3);
        obj.add(sobj4);
        obj.add(sobj2);

        System.out.println("\nHashSet after adding the objects: " + obj);

        System.out.println(obj.contains(sobj4));
        System.out.println(obj.isEmpty());

   }
}
```

Once the preceding code is executed, the following output is displayed:

```
HashSet after adding the objects: [Element 4, Element 3, Element 2, Element
1]
true
false
```

In the preceding code, the `contains()` method returns `true` because `sobj4` is present in the collection. In addition, the `isEmpty()` method returns `false` because the collection is not empty and it contains objects.

Working with the TreeSet Class

The following table lists some additional methods of the `TreeSet` class with their description.

Method	***Description***
`E first()`	*Is used to get the first object from `TreeSet`.*
`E last()`	*Is used to get the last object from `TreeSet`.*

The Methods of the TreeSet Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.TreeSet;

public class TreeSetDemo
{
    public static void main(String[] args)
    {
        TreeSet<Integer> obj = new TreeSet<Integer>();
        Integer iobj1 = new Integer(114);
        Integer iobj2 = new Integer(111);
        Integer iobj3 = new Integer(113);
        Integer iobj4 = new Integer(112);

        obj.add(iobj1);
        obj.add(iobj2);
        obj.add(iobj3);
        obj.add(iobj4);
        obj.add(iobj2);

        System.out.println("\nTreeSet after adding the objects: " + obj);

        System.out.println("\nThe first object of TreeSet is: " +
obj.first());
        System.out.println("\nThe last object of TreeSet is: " + obj.last());

   }
}
```

Once the preceding code is executed, the following output is displayed:

```
TreeSet after adding the objects: [111, 112, 113, 114]

The first object of TreeSet is: 111

The last object of TreeSet is: 114
```

In the preceding code, objects have been added to the collection in the sorted order. Therefore, the `first()` method returns `111` because `iobj2` is the first object, which has the value, `111`, after sorting. In addition, the `last()` method returns `114` because `iobj1` is the last object, which has the value, `114`, after sorting.

Using the List Interface

Start the discussion by explaining the `List` interface by explaining the difference between the `List` interface and the `Set` interface. Further, tell the students that the `List` interface allows you to add duplicate objects, which will have a specific order. Thereafter, explain the class hierarchy of the `java.util` package that implements the `List` interface along with the figure given in the SG. Next, explain the `ListIterator` interface that enables the traversing of the objects in the list collection.

Further, explain the `ArrayList` class that implements the `List` interface and enables you to create a resizable array. Next, discuss how the object of the `ArrayList` class can be created by using the various constructors. Thereafter, explain the various methods of the `ArrayList` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the ArrayList Class subtopic in the Additional Inputs topic.

Next, explain the `LinkedList` class that implements the `List` interface and enables you to create the doubly-linked list. In addition, while explaining the `LinkedList` class, you can discuss the difference between `ArrayList` and `LinkedList`, as given under the Working with the LinkedList Class subtopic in the Additional Inputs topic. Thereafter, discuss how the object of the `LinkedList` class can be created by using the various constructors. Further, explain the various methods of the `LinkedList` class along with the code given in the SG.

Thereafter, explain the `Vector` class that implements the `List` interface and enables you to create the list collection similar to `ArrayList` and `LinkedList`. Further, explain how `Vector` is different from `ArrayList` and `LinkedList`. Next, discuss how the object of the `Vector` class can be created by using the various constructors. Further, explain the various methods of the `Vector` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the Vector Class subtopic in the Additional Inputs topic.

Additional Inputs

Working with the ArrayList Class

The following table lists some additional methods of the `ArrayList` class with their description.

Method	***Description***
`int indexOf(Object o)`	*Is used to get the index of the first occurrence of the specified object in `ArrayList`.*
`int lastIndexOf(Object o)`	*Is used to get the index of the last occurrence of the specified object in `ArrayList`.*

The Methods of the ArrayList Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.ArrayList;

public class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> obj = new ArrayList<String>();
        String sobj1 = new String("Element 1");
        String sobj2 = new String("Element 2");
        String sobj3 = new String("Element 3");
        String sobj4 = new String("Element 4");

        obj.add(sobj1);
        obj.add(sobj2);
        obj.add(sobj3);
        obj.add(sobj1);
        obj.add(sobj4);

        System.out.println("\nArrayList after adding the objects: " + obj);

        System.out.println("\nThe index of the first occurrence of sobj1: " +
obj.indexOf(sobj1));
        System.out.println("\nThe index of the last occurrence of sobj1: " +
obj.lastIndexOf(sobj1));
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
ArrayList after adding the objects: [Element 1, Element 2, Element 3, Element
1, Element 4]

The index of the first occurrence of sobj1: 0

The index of the last occurrence of sobj1: 3
```

In the preceding code, the `indexOf()` method returns `0` because the first occurrence of `sobj1` is at the index, `0`. In addition, the `lastIndexOf()` method returns `3` because the last occurrence of `sobj1` is at the index, `3`.

Working with the LinkedList Class

`LinkedList` allows constant-time insertions or removals, but only sequential access of objects. You can traverse the list forwards or backwards, but getting an object in the middle takes time proportional to the size of the list. On the other hand, `ArrayList` allows random access so that you can get any object in constant time. However, adding or removing to/from anywhere except at/from the end requires shifting all the elements over, either to make an opening or to fill the gap.

Working with the Vector Class

The following table lists some additional methods of the `Vector` class with their description.

Method	***Description***
`E firstElement()`	*Is used to get the first object from `Vector`.*
`Void insertElement(E obj, int index)`	*Is used to insert the specified object at the specified index in `Vector`.*
`E lastElement()`	*Is used to get the last object from `Vector`.*

The Methods of the Vector Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.Vector;

public class VectorDemo
{
    public static void main(String[] args)
    {
        Vector<Double> obj = new Vector<Double>();
        Double dobj1 = new Double(77.5);
        Double dobj2 = new Double(68.1);
        Double dobj3 = new Double(52.8);
        Double dobj4 = new Double(40.2);

        obj.add(dobj1);
        obj.add(dobj3);
        obj.add(dobj2);
        obj.add(dobj1);
        obj.add(dobj4);

        System.out.println("\nVector after adding the objects: " + obj);

        System.out.println("\nThe first object from Vector: " +
obj.firstElement());

        obj.insertElementAt(new Double(33.45), 2);

        System.out.println("\nVector after inserting new object at index 2: "
+ obj);

        System.out.println("\nThe last object from Vector: " +
obj.lastElement());

    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Vector after adding the objects: [77.5, 52.8, 68.1, 77.5, 40.2]

The first object from Vector: 77.5

Vector after inserting new object at index 2: [77.5, 52.8, 33.45, 68.1, 77.5,
40.2]

The last object from Vector: 40.2
```

In the preceding code, the `firstElement()` and `lastElement()` methods return `77.5` as the first object and `40.2` as the last object, respectively. In addition, the `insertElement()` method inserts `33.45` at the index, 2.

Using the Map Interface

Start the discussion by explaining the concept of key-value pair. Then, inform the student that to create a collection of key-value pair objects, you can use the `Map` interface. Further, emphasize on the point that `Map` allows duplicate value objects, but the key object must be unique. Thereafter, explain the class hierarchy of the `java.util` package that implements the `Map` interface along with the figure given in the SG. In addition, inform the students that the `Hashtable` class inherits the `Dictionary` class. In addition, the `Dictionary` class implements the `Map` interface.

Further, explain the `HashMap` class that implements the `Map` interface and enables you to create a collection of objects in an unordered form. Next, discuss how the object of the `HashMap` class can be created by using the various constructors. Thereafter, explain the various method of the `HashMap` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the HashMap Class subtopic in the Additional Inputs topic.

Next, explain the `TreeMap` class that implements the `Map` interface and enables you to create a collection of objects in the sorted order with unique keys. In addition, discuss how an object of the `TreeMap` class can be created by using the various constructors. Thereafter, explain the various methods of the `TreeMap` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the TreeMap Class subtopic in the Additional Inputs topic.

Thereafter, explain the `Hashtable` class that implements the `Map` interface and enables you to create an unordered collection of objects that cannot contain the `null` object. Further, explain how `Hashtable` is different from `HashMap` and `TreeMap`. Next, discuss how an object of the `Hashtable` class can be created by using the various constructors. Thereafter, explain the various methods of the `Hashtable` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods given under the Working with the Hashtable Class subtopic in the Additional Inputs topic.

Additional Inputs

Working with the HashMap Class

The following table lists some additional methods of the `HashMap` class with their description.

Method	***Description***
`Set<K> keySet()`	*Is used to get the `Set` view of the key objects contained in `HashMap`.*
`Collection<V> values()`	*Is used to get a collection of values of `HashMap`.*

The Methods of the HashMap Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.HashMap;
public class HashMapDemo
    {
        public static void main(String[] args)
        {
        HashMap<String, Integer> obj = new HashMap<String, Integer>();
        Integer iobj1 = new Integer(11);
        Integer iobj2 = new Integer(44);
        Integer iobj3 = new Integer(22);
        Integer iobj4 = new Integer(33);
        obj.put("L1", iobj1);
        obj.put("L2", iobj2);
        obj.put("L3", iobj3);
        obj.put("L4", iobj4);
        obj.put("L1", iobj3);
        System.out.println("HashMap after adding the objects: " + obj);

        System.out.println("\nHashMap contains the following key objects: ");
        for(int i=0;i<obj.size();i++)
        {
          System.out.println(obj.keySet().toArray()[i]);
        }

        System.out.println("\nHashMap contains the following value objects:
");
        for(int i=0;i<obj.size();i++)
        {
          System.out.println(obj.values().toArray()[i]);
        }
        }
    }
```

Once the preceding code is executed, the following output is displayed:

```
HashMap after adding the objects: {L1=22, L2=44, L3=22, L4=33}

HashMap contains the following key objects:
L1
L2
L3
L4

HashMap contains the following value objects:
22
44
22
33
```

In the preceding code, the `keyset()` method returns the set of key objects, and the `values()` method returns the collection of value objects. In addition, the `toArray()` method is used to get the array of objects of a set or collection.

Working with the TreeMap Class

The following table lists some additional methods of the `TreeMap` class with their description.

Method	***Description***
`Map.Entry<K,V> firstEntry()`	*Is used to get the first key-value mapping in `TreeMap`.*
`Map.Entry<K,V> lastEntry()`	*Is used to get the last key-value mapping in `TreeMap`.*

The Methods of the TreeMap Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.TreeMap;

public class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap<Integer, String> obj = new TreeMap<Integer, String>();
        String sobj1 = new String("Value 2");
        String sobj2 = new String("Value 1");
        String sobj3 = new String("Value 3");
        String sobj4 = new String("Value 4");

        obj.put(101, sobj1);
        obj.put(102, sobj2);
        obj.put(103, sobj3);
        obj.put(104, sobj4);
        obj.put(105, sobj3);
        System.out.println("TreeMap after adding the objects: " + obj);
        System.out.println("\nFirst key-value mapping is: " +
obj.firstEntry());
        System.out.println("\nLast key-value mapping is: " +
obj.lastEntry());

    }
}
```

Once the preceding code is executed, the following output is displayed:

```
TreeMap after adding the objects: {101=Value 2, 102=Value 1, 103=Value 3,
104=Value 4, 105=Value 3}

First key-value mapping is: 101=Value 2

Last key-value mapping is: 105=Value 3
```

In the preceding code, the `firstEntry()` method returns `101=Value 2` as the first key-value mapping, and the `lastEntry()` method returns `105=Value 3` as the last key-value mapping.

Working with the Hashtable Class

The following table lists some additional methods of the `Hashtable` class with their description.

Method	***Description***
`boolean containsKey(Object key)`	*Is used to check whether the specified key is present in `Hashtable`.*
`Boolean containsValue(Object value)`	*Is used to check whether the specified value is present in `Hashtable`.*

The Methods of the Hashtable Class

Consider the following code that demonstrates the use of the preceding methods:

```
import java.util.Hashtable;

public class HashtableDemo
{
    public static void main(String[] args)
    {
        Hashtable<Integer, Double> obj = new Hashtable<Integer, Double>();
        Double dobj1 = new Double(55.6);
        Double dobj2 = new Double(34.6);
        Double dobj3 = new Double(98.6);
        Double dobj4 = new Double(12.5);

        obj.put(1, dobj1);
        obj.put(2, dobj2);
        obj.put(3, dobj3);
        obj.put(4, dobj4);
        obj.put(1, dobj3);
        System.out.println("Hashtable after adding the objects: " + obj);

 System.out.println("\nDoes Hashtable contain the key as 5: "
+obj.containsKey(5));

     System.out.println("\nDoes Hashtable contain the value as 34.6: "
+obj.containsValue(34.6));
   }
}
```

Once the preceding code is executed, the following output is displayed:

```
Hashtable after adding the objects: {4=12.5, 3=98.6, 2=34.6, 1=98.6}

Does Hashtable contain the key as 5: false

Does Hashtable contain the value as 34.6: true
```

In the preceding code, the `containsKey()` method returns `false` because the key, `5`, is not present in `Hashtable`. In addition, the `containsValue()` method returns `true` because the value, `34.6`, is present in `Hashtable`.

FAQs

- *Why does the `Map` interface not extend the `Collection` interface?*

  Ans: The `Map` interface does not extend the `Collection` interface because the `Map` interface represents mappings and not a collection of objects.

- *Can the `for` loop be used instead of `Iterator` or `ListIterator` to traverse the objects of a collection?*

  Ans: Yes, the *`for`* loop can be used to traverse the objects of a collection.

- *Can objects be removed during iteration?*

  Ans: Yes, objects can be removed during iteration by using the `remove()` method.

- *What is the difference between size and capacity?*

  Ans: Size is the number of objects stored in a particular collection. On the other hand, capacity is the maximum number of objects a collection can store at a given point of time.