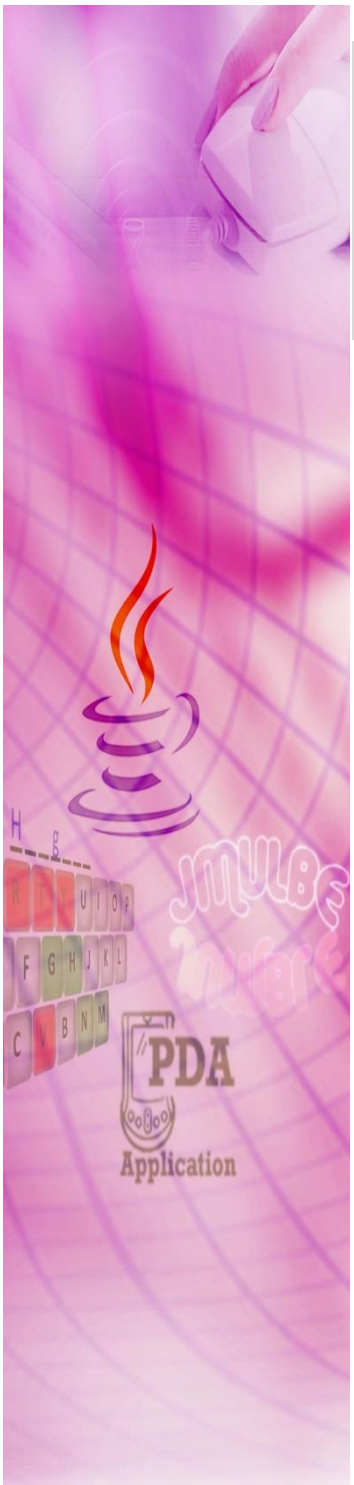

Instructor Inputs

Session 5

Session Overview

This session covers the Using the Deque Interface and Implementing Sorting sections of Chapter 4, the Using Threads in Java section, and the following topics of the Creating Threads section of Chapter 5 of the book, Programming in Java – SG:

- Creating a Thread by Extending the Thread Class
- Creating a Thread by Implementing the Runnable Interface

This session will help the students to become familiar with the `Deque` interface to create a collection of objects. In addition, the students will learn how to sort the objects of a collection. Furthermore, the students will explore the concept of threads in Java. They will learn creating and using threads in Java.

Using the Deque Interface

Start the discussion by explaining a collection of objects using the `Deque` interface. Tell the students that the `Deque` interface enables the insertion and deletion of an object in a collection from both the ends. Further, emphasize on the point that the `Deque` interface extends the `Queue` interface, which works on the First In First Out (FIFO) basis. Thereafter, explain the class hierarchy of the `java.util` package, which implements the `Deque` interface, along with the figure given in the SG.

Further, explain that the `ArrayDeque` class implements the `Deque` interface and allows you to create a resizable array, which enables you to insert or delete objects to/ from both the ends. Next, discuss how an object of the `ArrayDeque` class can be created by using its various constructors. Thereafter, explain the various methods of the `ArrayDeque` class along with the code given in the SG. In addition, while explaining the methods, you can discuss some additional methods, as given in the table under the Additional Inputs topic.

Additional Inputs

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

Method	***Description***
`E peekFirst()`	*Is used to retrieve the first object from `ArrayDeque`. It returns `null` if `Deque` is empty.*
`E peekLast()`	*Is used to retrieve the last object from `ArrayDeque`. It returns `null` if `Deque` is empty.*

The Methods of the ArrayDeque Class

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

```
import java.util.ArrayDeque;

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

```
    {
        ArrayDeque<Double> obj = new ArrayDeque<Double>();
        Double dobj1 = new Double(7.5);
        Double dobj2 = new Double(8.5);
        Double dobj3 = new Double(9.5);
        Double dobj4 = new Double(10.5);

        System.out.println("The first object of ArrayDeque is: " +
obj.peekFirst());

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

        obj.addFirst(dobj2);
        obj.addLast(dobj3);

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

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

    }
}
```

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

```
The first object of ArrayDeque is: null

ArrayDeque after adding the objects: [8.5, 7.5, 10.5, 9.5]

The last object of ArrayDeque is: 9.5
```

In the preceding code, the `peekFirst()` method returns the `null` value because there is no object in `ArrayDeque`. In addition, the `peekLast()` method returns `9.5` because it is the last object in `ArrayDeque`.

Implementing Sorting

Start the discussion by explaining the scenario of the Address Book application in the context of a collection of objects for the contact details. Further, discuss the need of sorting the contact details according to the contact names. Then, emphasize on the point that Java provides the `Comparable` and `Comparator` interfaces.

Further, explain the `Comparable` interface, which is present in the `java.lang` package. Discuss the `compareTo()` method that is used for comparing the references of the objects. Next, discuss the example of the `Student` class that overrides the `compareTo()` method to define the sorting logic. In addition, while demonstrating the code, you can discuss the additional information given in the Additional Inputs topic.

Thereafter, explain the `Comparator` interface, which is present in the `java.util` package and provides the `compare()` method to compare the references of two objects. Then, discuss the difference between the `Comparable` interface and the `Comparator` interface. Next, discuss the example of the `Student` class with the two different sorting logics by using the `compare()` method. In addition, while demonstrating the code, you can discuss the additional example given in the Additional Examples topic.

Additional Inputs

Consider the example of the `Employee` class that stores the employee ID and employee name, as shown in the following code:

```
class Employee
{
    int empID;
    String empName;

    public Employee(int eID, String eName)
    {
        empID = eID;
        empName = eName;
    }

    public int getID()
    {
        return empID;
    }

    public String getName()
    {
        return empName;
    }

    public String toString()
    {
        return "Employee ID: " + empID + "\nEmployee Name: " + empName;
    }

}
```

Further, to create an employee list that will contain only the employee objects, you can create the `EmployeeList` class, as shown in the following code:

```
class EmployeeList
{
    public static void main(String[] args)
    {
        TreeSet<Employee> obj = new TreeSet<Employee>();
        Employee e1 = new Employee(104, "Ricky Donald");
        Employee e2 = new Employee(101, "Peter Parker");
        Employee e3 = new Employee(103, "Mark Jones");
        Employee e4 = new Employee(102, "Henry Wills");

        obj.add(e1);
        obj.add(e2);
        obj.add(e3);
        obj.add(e4);

        System.out.println("\n\nEmployee List is as follow:");

        Iterator i = obj.iterator();
        while(i.hasNext())
```

```
        {
            System.out.println("\n" + i.next());
        }

    }
}
```

Once the preceding code is executed, the following exception is generated:

```
Exception in thread "main" java.lang.ClassCastException: Employee cannot be
cast to java.lang.Comparable
```

The preceding exception is thrown because the `TreeSet` class creates a collection of objects of the user-defined classes only when the classes implement the `Comparable` interface. However, the `Employee` class does not implement the `Comparable` interface. In addition, the `Employee` class does not override the `compareTo()` method. To ensure that the exception is not raised, you need to modify the `Employee` class, as shown in the following code:

```
class Employee implements Comparable<Employee>
{
    int empID;
    String empName;

    public Employee(int eID, String eName)
    {
        empID = eID;
        empName = eName;
    }

    public int getID()
    {
        return empID;
    }

    public String getName()
    {
        return empName;
    }

    public String toString()
    {
        return "Employee ID: " + empID + "\nEmployee Name: " + empName;
    }

    public int compareTo(Employee obj)
    {

        if (this.empID > obj.getID())
            return 1;
        else if (this.empID < obj.getID())
            return -1;
        else
            return 0;
    }
}
```

In the preceding code, the `Employee` class extends the `Comparable` interface and overrides the `compareTo()` method. The `compareTo()` method compares the employee IDs and sorts the employee objects in the ascending order on the basis of the employee IDs.

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

```
Employee List is as follow:

Employee ID: 101
Employee Name: Peter Parker

Employee ID: 102
Employee Name: Henry Wills

Employee ID: 103
Employee Name: Mark Jones

Employee ID: 104
```

Additional Examples

Consider a scenario where you want to create a class that will store the project ID and project name. For this, you can use the following code:

```
class Project
{
    String projID;
    String projName;

    public Project (String pID, String pName)
    {
        projID = pID;
        projName = pName;
    }

    public String getID()
    {
        return projID;
    }

    public String getName()
    {
        return projName;
    }

    public String toString()
    {
        return "Project ID: " + projID + "\nProject Name: " + projName;
    }
}
```

In the preceding code, the `Project` class defines the member variables, `projID` and `projName`. In addition, the setter and getter methods are defined.

Thereafter, you want to sort the list of projects according to project name. For this, you have decided to use the `Comparator` interface, as shown in the following code:

```
import java.util.Comparator;

class SortProject implements Comparator<Project>
{
    public int compare(Project a, Project b)
    {
         return a.getName().compareTo(b.getName());

    }
}
```

In the preceding code, `SortProject` class implements the `Comparator` interface. In addition, the `compare()` method is overridden to compare the instance of `Project` object.

In addition, you want to create a class to create a list of all the projects. For this, you have decided to create a collection, as shown in the following code:

```
import java.util.Iterator;
import java.util.TreeSet;

class ProjectList
{
    public static void main(String[] args)
    {
        TreeSet<Project> obj = new TreeSet< Project >(new SortProject());
        Project e1 = new Project("T103", "Turbine Flex");
        Project e2 = new Project("T101", "Beta Core");
        Project e3 = new Project("T104", "German Blade");
        Project e4 = new Project("T102", "Textile Dome");

        obj.add(e1);
        obj.add(e2);
        obj.add(e3);
        obj.add(e4);

        System.out.println("\n\nProject List is as follow:");

        Iterator i = obj.iterator();
        while(i.hasNext())
        {
            System.out.println("\n" + i.next());
        }

   }
}
```

In the preceding code, an object of `TreeSet` class is created. The `add()` method of `TreeSet` class is used to add the objects of the `Project` class. In addition, the `Iterator` interface is used to traverse through the `TreeSet` object.

Activity 4.1: Working with Collections

Handling Tips

Discuss the problem statement with the students.

To perform the activity, 4.1, you need to use the **Contact.txt**, **AddContact.txt**, and **Menu.txt** files, which are provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 04\Activity 4.1\Input Files**

The solution files, **Contact.java**, **AddContact.java**, and **Menu.java**, for this activity are provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 04\Activity 4.1\Solution**

Using Threads in Java

Handling Tips

Ask the students if they have played a computer game. Elicit the responses and tell them that in a computer game, multiple functions, such as displaying the score, playing the background music, and moving the characters, occur simultaneously. Tell them that in order to implement the same functionality in Java, they can use threads. Provide them the definition of threads. Explain this with the example of a text editor and browser, as given in the SG.

Next, explain a single-threaded application and the requirement of multithreading. Briefly, explain the advantages and disadvantages of multithreading along with the information given under the The Basic Concept of Multithreading subtopic in the Additional Inputs topic. Focus on the race condition, deadlock condition, and lock starvation. Ensure that the students have gained a clear understanding on the preceding topics. After the students have understood the concept of threads and multithreaded programming, explain how to implement threads in Java by using the `Thread` class. Explain the information about the daemon threads given under the The Thread Class subtopic in the Additional Inputs topic. You also need to explain why the main thread is essential in Java and how it works.

You need to explain the life cycle of a thread to the students. In addition, you must explain the following stages of the life cycle of a thread with the help of the figure given in the SG:

- New
- Runnable
- Not Runnable
- Terminated or Dead

While explaining the preceding stages, you must also explain the various methods required during those stages. In addition, you can also discuss the additional information given under the The Life Cycle of a Thread subtopic in the Additional Inputs topic.

At the end of this section, you may ask the students the following questions to check whether the students have understood the concepts learned so far:

- In which situation a thread enters the blocked state?
- Which methods force the running thread to enter a waiting or blocked state?

Additional Inputs

The Basic Concept of Multithreading

In the multithreaded application, when a CPU switches from one thread to another, it performs a context switch. This involves saving the state of an old process or thread and loading the state of a new one. During the execution of a large or complex application, context switching can lead to a significant overhead of the execution of an application. Therefore, the applications must be designed so that context switching is reduced and the performance of the application is improved. The context switching overhead can be reduced by minimizing the number of active threads being used for an application.

The Thread Class

In Java, some threads are classified as daemon threads. Daemon threads are created automatically by JVM and are useful to run as a background service. For example, the garbage collector represents a daemon thread. The main thread is a non daemon thread. The status of a thread can be changed to a daemon thread by using the `setDeamon()` method of the `Thread` class.

The Life Cycle of a Thread

The various thread states are defined in the `Thread.State` enum. Some of the constants defined in this enum are listed in the following table.

Enum Constant	***Description***
`NEW`	*Thread state for a thread that has not started.*
`RUNNABLE`	*Thread state for a runnable thread.*
`TERMINATED`	*Thread state for a terminated thread.*
`WAITING`	*Thread state for a waiting thread.*
`TIMED_WAITING`	*Thread state for a thread with a waiting time.*

The Constants of the Thread.State Enum

You can use the following methods to work with the constants of the `Thread.State` enum:

- `static Thread.State valueOf(String name)`: Returns the enum constant of this type with the specified name.
- `static Thread.State[]values()`: Returns an array containing the constants of this enum type in the order in which they are declared.

Creating Threads

Handling Tips

Explain the scenario given in the SG to the students, and then tell the students that threads can be created by:

- Extending the `Thread` class.
- Implementing the `Runnable` interface.

While explaining the creation of threads, you must explain the importance of the `run()` and `start()` methods to the students. Demonstrate the two ways to create threads with the help of the examples given in the SG and also explain the difference between them. While explaining the creation of a thread by using the `Thread` class, you can demonstrate the code given under the Create a Thread by Extending the Thread Class subtopic in the Additional Examples topic.

Additional Examples

Create a Thread by Extending the Thread Class

The following code demonstrates how to create a thread by using the `Thread` class:

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 1; i < 11; i++) {
            System.out.println(i + " ");
            try {
                sleep(1000);
            } catch (InterruptedException e) {
            }
        }
     }
    public static void main(String args[]) {
        SimpleThread th = new SimpleThread("Thread-A");
        th.start();

    }
}
```

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

```
1
2
3
4
5
6
7
8
9
10
```

In the preceding code, a thread, `Thread-A`, is created. In addition, the run() method is overridden to print the numbers from 1 to 10.

FAQs

- *What happens if we call the `start()` method on the same thread more than once?*

  Ans: If you call the `start()` method more than once, it will throw an exception at runtime.

- *Can we stop a thread execution?*

  Ans: You can try to stop the execution of a thread by using the `stop()` method on a thread. The `stop()` method is deprecated and should not be used as it is potentially unsafe.

- *Can we create a group of threads?*

  Ans: Yes, in Java, you can create a group of threads by using the `ThreadGroup` class.

- *If we do not override the `run()` method after extending the `Thread` class in a class, will it generate an error?*

  Ans: No, the program will not generate an error.

- *How can we determine the number of threads in a program?*

  Ans: The number of threads can be determined by using the `activeCount()` method of the `Thread` class. You can use the following code snippet for accomplishing the preceding task:

  ```
  System.out.println(Thread.activeCount());
  ```

- *Can we apply multiple ways of sorting logic with a single collection of objects?*

  Ans: Yes, you can apply multiple ways of sorting logic with a single collection of objects. For this, you have to explicitly specify each sorting logic by using the `sort()` method of the `Collections` class.