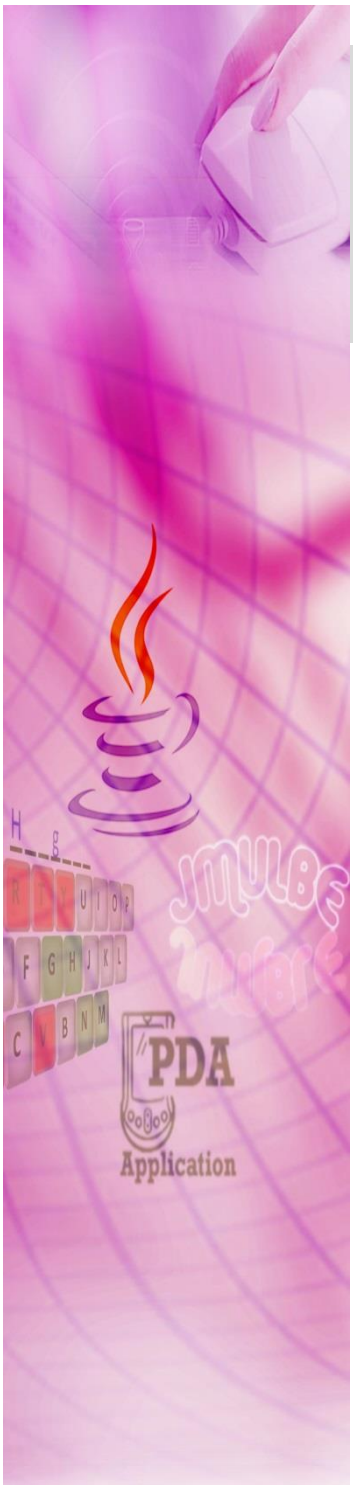

Session 8

Instructor Inputs

Session Overview

This session covers Chapter 6 of the book, Introduction to Java – SG. This session will help the students to become familiar with the concept of exception handling in Java by using the `try`, `catch`, `finally`, and `try`-with resources blocks and also by using the `throw` and `throws` statements. In addition, the students will learn to throw an exception and implement user-defined exceptions and assertions.

Handling Exceptions

Handling Tips

Ask the students about the types of errors they usually encounter while programming. Then, tell the students that in a real-time application, the abnormal termination of a program can have severe consequences. For example, if a distributed application encounters an error or terminates abnormally, then the work of a large number of users can get affected and also the precious time will be wasted.

Now, tell the students that to prevent these issues, Java provides a mechanism known as exception handling. Then, discuss the scenario with the students and relate exception handling to the Classic Jumble Word game. Explain the advantages of exceptions given under the Exploring Exceptions heading in the Additional Inputs topic. Briefly explain to the students the built-in exceptions and exception classes. Next, explain the concept of the checked and unchecked exceptions and focus on the fact that it is mandatory for a programmer to handle checked exceptions. If a method throws a checked exception, then it can be either handled in that method or in the method that had called it.

Further, tell them about the keywords that are used to implement exception handling. Explain each of the keywords with its individual usage. After explaining the `try` and `catch` blocks, explain the concept of nested `try` and `catch` block with the help of the example given in the Additional Inputs topic.

Next, tell the students that sometimes, it is necessary to execute certain statements, irrespective of the fact whether an exception is raised or not. In that case, the `finally` block can be used to execute those required statements. Then, tell the students that they can throw an exception explicitly by using the `throw` keyword. Further, tell the students that the `throw` statement causes termination of the normal flow of the control of the Java code and stops the execution of subsequent statements after the throw statement. Next, introduce the `throws` keyword and tell the students that the `throws` clause is used by a method to specify the types of exceptions thrown by the method. The `throw` keyword throws a new exception, whereas the `throws` keyword is used to declare that a method may raise an exception. Then, tell the students that in addition to the built-in exceptions, they can create customized exceptions, as per the application requirements. Explain the steps to create user-defined exceptions and ensure that the students have gained a clear understanding of the topic. Before concluding the topic, explain the example given in the Additional Examples topic.

Additional Inputs

Exploring Exceptions

Exceptions have the following benefits:

- By using exceptions, you can separate the programming logic of your application from the details that need to be implemented when errors are identified in the application. This makes the application more readable.

- By using exceptions, only those methods that can be affected by the exception are required to handle the exceptions. Java exceptions provide the ability to propagate error reporting in the method stack. The JVM searches backwards through the method call stack to identify the methods that need to handle a specific exception.
- Java exceptions are categorized and grouped in a class hierarchy, which makes it easier to implement them. For example, to catch a specific I/O exception, you can specify an exception handler with the `IOException` argument, as shown in the following code snippet:

```
catch (IOException e)
{
…
}
```

Implementing Exception Handling

The nested try-catch block is used to handle exceptions in Java applications. You can enclose a try-catch block in an existing try-catch block. The enclosed try-catch block is called the inner try-catch block and the enclosing block is called the outer try-catch block. If the inner `try` block does not contain the matching catch statement to handle an exception, then the `catch` statement in the outer block is checked for the exception handler. The following code shows how to implement a nested try-catch block:

```
public class NestedTryCatch
{

    public static void main(String args[])
    {
    int num[]= {1,2,3,4,5};
    try {
          try
            {
                int number= args.length;
                for(int i=0;i<num.length; i++)
                {
                       int result = num[i]/i;

                }
            }
          catch(ArithmeticException e)
                {
                 System.out.println("Number can not be divided by zero");
                }
       num[6]=7;

        }

    catch(ArrayIndexOutOfBoundsException e)
    {
         System.out.println("Incorrect array index");
    }

    }

}
```


Additional Examples

The following code shows how to implement user-defined exceptions:

- Creating an exception class named `CalculateException`.

```
class CalculateException extends Exception
{

    CalculateException()
    {
           System.out.println("Invalid input, Recheck the values");

    }
}
```

- Implementing user-defined exception in `SimpleInterest` class.

```
import java.util.Scanner;
class SimpleInterest
{
    static void calculate(int p,int r, int t) throws CalculateException
    {
           if((p<=0) || (r<=0)|| (t<=0))
                 {
                  throw new CalculateException();
                 }
                 else
                 {
                  System.out.println("The Simple Interest is: ");
                     System.out.println((p*r*t)/100);
                 }
    }
    public static void main(String args[])
    {

              Scanner input = new Scanner(System.in);
              System.out.println("Enter Principal Amount: ");
              int principal= input.nextInt();
               System.out.println("Enter Rate: ");
              int rate= input.nextInt();
               System.out.println("Enter Time: ");
              int time= input.nextInt();

              try
           {
                 calculate(principal,rate,time);
           }
           catch (CalculateException Obja)
           {
                 System.out.println("Exception caught: " + Obja);
           }
    }
}
```

Activity 6.1: Exception Handling

Handling Tips

Discuss the problem statement with the students.

To perform the activity, 6.1, you need to use the **Activity4.1.txt** file, which is provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 06\Activity 6.1\Input Files**

The solution files, **Hangman.java**, **MenuInputException.java**, and **WrongInputException.java**, for this activity are located at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 06\Activity 6.1\Solution**

Using the assert Keyword

Handling Tips

Discuss the scenario given in the SG with the students. Tell them that an assertion enables a user to test the assumptions about a program. For example, in a program that calculates the speed of a vehicle, you can always assume that the speed will be less than the speed of light. Explain another example that in a program that performs the division of integers, you can assert that the division will not be performed by 0.

Then, explain the benefits of assertions to the students. Next, tell the students about the types of assertions supported by Java with the help of the information given in the Additional Inputs topic. Tell them that to implement assertions in a Java program, assertions must be enabled by using the `-ea` VM option. Further, tell them that the `assert` keyword is used to implement assertions in Java. Explain how to implement assertions by using a suitable code snippet and conclude the session. Further, you can explain how to use assertions with a `switch` construct by using the example given in the Additional Examples topic.

Additional Inputs

Implementing Assertions

Java supports the following types of assertions:

- **Preconditions**: This assertion must be true before invoking a method. A precondition ensures that a method is invoked only when certain conditions hold true. For example, before extracting a value from a stack, you need to ensure that the stack is not empty.
- **Postconditions**: This assertion is applied at the end of a method or code and. It must be true at the end when it is checked. Postconditions work in a contract with preconditions, such that if the precondition is true at the beginning of a method, then after the method has executed, the method must guarantee that the postcondition is true. For example, after inserting a value in the stack, verify if it has been added on the top of the stack.
- **Invariants**: This assertion must always be true wherever specified in a code or method. For example, the stack itself should exist.

Additional Examples

Implementing Assertions

The following code demonstrates the use of assertions with a `switch` construct:

```
public class Result
{

    public static void main(String[] args)
{
char operator = '+';       // assumed either '+', '-', '*', '/' only
      int operand1 = 5, operand2 = 6, result = 0;
      switch (operator) {
          case '+': result = operand1 + operand2;
          break;
          case '-': result = operand1 - operand2;
          break;
          case '*': result = operand1 * operand2;
          break;
          case '/': result = operand1 / operand2;
          break;
         default: assert false : "Unknown operator: " + operator;  // not
allowed here
      }
      System.out.println(operand1 + " " + operator + " " + operand2 + " = " +
result);
   }

    }
```

In the preceding code, an assertion is implemented that either of the `+`,`-`,`*`, or `/` operators should be used as an operator. If any other operator is used, then the program will terminate with `AssertionError`.

Activity 6.2: Implementing Assertions

Handling Tips

Discuss the problem statement with the students.

The solution file, **CalculatorApp.java**, for this activity is located at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 06\Activity 6.2\Solution**

FAQs

- *Differentiate between the `throw` and `throws` keywords.*

  Ans: The `throw` keyword is used to throw an exception explicitly. The `throws` keyword is used to tell the compiler that a particular piece of code may raise a specific exception.

- *Do we need to catch all types of exceptions?*

  Ans: Yes, you need to catch all types of exceptions that are thrown in a Java program.

- *How many exceptions can we associate with a single* `try` *block?*

  Ans: There is no limit to the number of exceptions that can be associated with a single `try` block. All the exceptions associated with a `try` block should be handled by using multiple `catch` blocks.