UNIT4

Managing Errors and Exceptions

Introduction

It is quite common to make mistakes while typing program. A mistakes might lead program to produce wrong results which is called an errors.

Types of Errors

Errors may broadly be classified into two categories:

1. Compile-time errors
2. Run time errors

Compile-Time Errors

1. All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors.
2. Whenever the compiler displays an error, it will not create the **.class** file. Therefore it is necessary that we fix all the errors before we can compile and run the program.

```
// Illustration of compile-time error
class Error1
{
        public static void main(String args[ ])
        {
                System.out.println("Hello Java")                    //Missing ;
        }
}
```

3. The Java compiler does a nice job of telling us where the errors are in the program. Ex the following message will be displayed in the screen:

   Error1.java :7: ';' expected

System.out.println(“Hello Java!”)

^

1 error

4. Now we can go to the appropriate line, correct the error, and recompile the program.
5. Compile-time errors are due to typing mistakes. The most common problems are:
    - Missing semicolons
    - Missing brackets in classes and methods
    - Misspelling of identifiers and keywords
    - Missing double quotes in strings
    - Use of undeclared variables
    - Incompatible types in assignments
    - Use of = in place of = = operator etc…

Run-Time Errors

1. Sometimes a program may compile successfully creating the **.class** file but may not run properly.
2. Such programs may produce wrong results. Most common run-time errors are:
    - Dividing an integer by zero
    - Accessing an element that is out of bounds of an array
    - Trying to store a value into an array of incompatible type
    - Passing a parameter that is not in a valid range or value of a method
    - Trying to illegally change the state of the thread
    - Converting invalid string to a number

- Accessing character that is out of bounds of a string etc…

3. When such errors encountered Java typically generates an error message and aborts the program.
4. The program below illustrates how a run-time errors causes termination of execution of program.

```
// Illustration of run-time errors

class Error2
{
        public static void main(String args[ ])
        {
                int a = 10;
                int b = 5;
                int c = 5;
                int x = a/(b-c);                //Division by zero
                System.out.println("x = " + x);
                int y = a/(b+c);
                System.out.println("y = " + y);
        }
}
```

5. After compilation it displays the following message without executing further statements.

```
java.lang.ArithmeticException: / by zero
        at Error2.main(Error2.java:8)
```

Exceptions

1. An exception is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error, it creates exception object and throws it.
2. If the exception object is not caught and handled properly, the interpreter will display an error message causing the program to terminate.
3. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and display an appropriate message. This task is known as exception handling.
4. The mechanism incorporates an error handling code that performs the following tasks:
    - Find the problem(Hit the exception)
    - Inform that an error has occurred(Throw the exception)
    - Receive the error information(Catch the exception)
    - Take corrective actions(handle the exception)
5. Some common exceptions that we must watch out for catching are listed below in the table.
6. Exceptions in Java can be categorized into two types:
    - **Checked exceptions:** These exceptions are explicitly handled in the code itself with the help of catch blocks. Checked exceptions are extended from the **java.lang.Exception** class.
    - **Unchecked exceptions:** These exceptions are not essentially handled in the program code, instead the JVM machine handles such exceptions. Unchecked exceptions are extended from the **java.lang.RuntimeException** class.

Table 13.1 Common Java Exceptions

Exception Type	*Cause of Exception*
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverFlowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Syntax of Exception Handling Code

1. The basic concept of exception handling are throwing an exception and catching it. This is illustrated in Fig

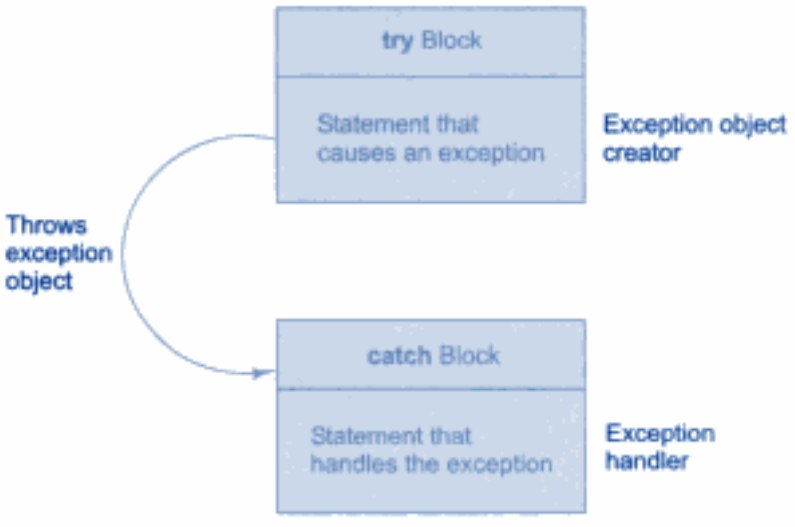

Fig. 13.1 *Exception handling mechanism*

2. Java uses the keyword try to preface a block of code that is likely to cause an error condition and "throw" an exception
3. A catch block defined by keyword **catch** "catches" the exception "thrown" by the try block and handles it appropriately.
4. The **catch** block is added immediately after the **try** block. The following example illustrates the use of simple **try** and **catch** statements:

```
……
……
try
{
        statement;
}
catch (Exception-type e)
{
        statement;
}
……
……
```

5. The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.
6. The catch block too can have one or more statements that is necessary to process the exception.
7. Every try statement should be followed by at least one catch statement.
8. The catch statement is passed a single parameter, which is an reference to the exception object thrown by the try block.
9. If the catch block matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught the execution will be terminated.

10. Program below illustrates use of try catch for exception handling.

```
class Error3
{
    public static void main(String args[ ])
    {
        int a = 10;
        int b = 5;
        int c = 5;
        int x, y;
        try
        {
            x=a / (b-c);
        }
        catch (ArithmeticException e)
        {
            System.out.println("Division by Zero");
        }
        y = a / (b+c);
        System.out.println("Y = " + y);
    }
}
```

This program displays the following output:

```
Division by Zero
y = 1
```

11. Here the program does not stop at the point of exception condition. It catches the error condition, prints the error message and then continues the execution, as if nothing has happened.
12. Consider another example of using exception handling mechanism.

```
class Error4
{
    public static void main(String args[ ])
    {
            int c;
            int a[ ] = {10,5};
            try
            {
                    c = a[0] / a[2];
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                    System.out.println("Caught Exception " + e);
            }
    }
}
```

Here program produces the following output:

Caught Exception: java.lang.ArrayIndexOutOfBoundsException : 2

13. There is a possibility of generation of multiple exceptions of different types within a particular block of the program code. We can use nested try

statements in such situations. The program below shows the example of nested try statements.

```
class Error5
{
        public static void main(String args[ ])
        {

                try
                {
                        int a = 10, b = 5, c = 5, x;
                        int p[ ] = {10,5};
                        c = p[0] / p[2];
                        try
                        {
                                x=a / (b-c);

                        }

                        catch (ArithmeticException e)

                        {

                                System.out.println("Division by Zero");

                        }

                }

                catch(ArrayIndexOutOfBoundsException e)

                {

                        System.out.println("Array Index is Out Of Bounds ");

                }

        }

}
```

Multiple Catch Statements

It is possible to have more than one catch statement in the catch block as illustrated below.

```
…….
…….
try
{
    statement;
}
catch(Exception-Type-1 e)
{
    statement;
}
catch(Exception-Type-2 e)
{
    statement;
}
.
.
.
catch(Exception-Type-N e)
{
    statement;
}
```

When an exception in the try block is generated, the first statement whose parameter matches with the exception object will be executed and remaining statements will be skipped.

```
class Error6
{
    public static void main(String args[ ])
    {
        int a[ ] = {5,10};
        int b  = 5;
        try
        {
            int x = a[2] / b – a[1];
        }
        catch (ArithmeticException e)
        {
            System.out.println(“Division by Zero”);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(“Array Index Error“);
        }
        catch(ArrayStoreException e)
        {
            System.out.println(“Wrong data type “);
        }
        int y = a[1] / a[0];
        System.out.println(“Y = “ +y);
    }
```

```
}
```

The above program produces the following output:

Array Index Error

Y = 2

Note that the array a[2] does not exist because array a is defined to have only two elements, a[0] and a[1] Therefore the index 2 is outside the array boundary thus causing the block.

```
catch(ArrayIndexOutOfBoundsException e)
{
        System.out.println("Wrong data type ");
}
```

to catch and handle the error remaining catch blocks are skipped.

Using Finally Statement

1. Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.
2. **finally** block can be used to handle any exception that is generated within a try block.
3. It may be added immediately after the try block or after the last catch block as shown below.
4. When finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown.

```
try
{
    …….
    ……..
}
finally
{
    ……...
    ……...
}
```

Or

```
try
{
    …….
    ……..
}
catch( ……)
{
    ……...
    ……...
}
catch( ……)
{
    ……...
    ……...
}
.
.
```

.

```
finally
{
    ……...
    ……...
}
```

5. The program below shows the use of finally block.

```
class Error6
{
        public static void main(String args[ ])
        {
                int a[ ] = { 5,10};
                try
                {
                        int x = a[1] / 0;
                }
                catch (ArithmeticException e)
                {
                        System.out.println("Division by Zero");
                }
                finally
                {
                        int y = a[1] / a[0];
                        System.out.println("y = " +y);
                }
```

```
        }
    }
```

The program produces the following output:

Division by Zero

y = 2

Throwing our Own Exceptions

1. There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

   throw new ***Throwable subclass;***

   Examples: throw new ArithmeticException( );

   throw new NumberFormatException( );

2. The below program demonstrates the use of a user-defined subclass of Throwable class. Note that Exception is a subclass of Throwable and therefore MyException is a subclass of Throwable class.

```
import java.lang.Exception;
class MyException extends Exception
{
    MyException(String message)
    {
        super(message);
    }
}
class Demo
{
    public static void main(String args[ ])
    {
        int age=15;
```

```
            try
            {
                if(age < 21)
                throw new MyException("Your Age is less");
            }
            catch(MyException e)
            {
                System.out.println("This is My Exception");
                System.out.println(e.getMessage( ));
            }
            finally
            {
                System.out.println("Finally Block : End of the Program");
            }
        }
    }
```

A run of program produces the following output:

This is My Exception

Your Age is less

Finally Block : End of the Program

3. The object e which contains the error message "Your Age is less" is caught by the catch block which then displays the message using getMessage( ) method

4. Note that the program also illustrates the finally block.
5. The throw class can be used when method throws certain kinds of an exceptions but there is no exception handling mechanism within the method.
6. It is specified immediately after the method declaration

```
class Error7
{
        static void divide_m( ) throws ArithmeticException
        {
                int x = 10, y = 0, z;
                z = x / y;
        }
        public static void main(String args[ ])
        {
                try
                {
                        divide_m( );
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Caught Exception" +e);
                }
        }
}
```

The program produces the following output:

Caught Exception java.lang.ArithmeticException : / by zero