**UNIT- 4**

Managing Errors and Exceptions-Syntax of Exception Handling Code-Using Finally Statement-Throwing Our Own Exceptions-Applet Programming-Applet Life Cycle-Graphics Programming-Managing Input/output Files: Concept of Streams-Stream Classes-Byte Stream Classes-Character Stream Classes – Using Streams-Using the File Class-Creation of Files-Random Access Files-Other Stream Classes.

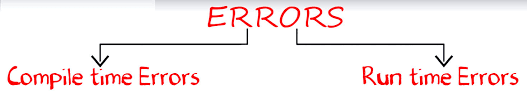
**MANAGING ERRORS AND EXCEPTIONS:**

**ERRORS AND VARIOUS TYPES OF ERRORS:**

* Errors are the wrongs that can make a program go wrong.
* In computer terminology errors may be referred to as **bugs**.
* It is common to make mistakes while developing as well as typing a program.
* A mistake might lead to an error causing to program to produce unexpected results.
* An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash.
* It is therefore important to detect and manage properly all the possible errors.

**TYPES OF ERRORS:**

Errors may broadly be classified into two categories:

1. Compile-time errors
2. Run-time errors

**COMPILE-TIME ERRORS:**

* All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as **compile-time errors**.
* Whenever the compiler displays an error, it will not create the .class file.
* It is necessary to fix these errors to get it compiled.
* It becomes an easy job to a programmer to correct these errors because Java compiler tells us where the errors are.
* Most of the compile-time errors are due to typing mistakes.
* Typographical errors are hard to find.
* We may have to check the code word by word, or even character by character.

The most common errors are:

* + - * Missing semicolons
      * Missing (or mismatch of) brackets in classes and methods
      * Misspelling of identifiers and keywords
      * Missing double quotes in strings
      * Use of undeclared variables Incompatible types in assignments / initialization
      * Bad references to objects
      * Use of = in place of = = operator

**RUN-TIME ERRORS:**

* Sometimes, a program may compile successfully creating the .class file but may not run properly these errors are known as **Run time error**.
* Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.

Most common run-time errors are:

* Dividing an integer by zero
* Accessing an element that is out of the bounds of an array
* Trying to store a value into an array of an incompatible class or type
* Trying to cast an instance of a class to one of its subclasses
* Passing a parameter that is not in a valid range or value for a method
* Trying to illegally change the state of a thread
* Attempting to use a negative size for an array
* Using a null object reference to access a method or a variable.
* Converting invalid string to a number
* Accessing a character that is out of bounds of a string

**Example:** The following code generates division-by-zero runtime error.

***int x=5, y=0;***

***System.out.println(x/y);***

It displays the following message and stops without executing further statements.

Java.lang.ArithmeticException: **/ by zero**

**EXCEPTION AND EXCPETION HANDLING:**

**EXCEPTION:**

* Exception is a condition caused by a runtime error in a program.
* It is an abnormal event that arises during the execution of the program and terminates the normal flow of the program.
* Abnormality occurs when the program is running.

**For example:**

* + - If a user enters a string where an integer is expected, an error occurs at runtime that causes the program to be stopped intermediately.
    - Java has a built-in mechanism for handling runtime errors, referred to as exception handling.
    - This is to ensure that we can write robust (error-free) programs.

**EXCEPTION HANDLING:**

* Handling runtime errors is exception handling.
* In java, when an abnormal condition occurs within a method then the exceptions are thrown in form of Exception Object
* i.e. the normal program control flow is stopped and an exception object is created to handle that exceptional condition.
* If the exception object is not caught and handled properly, the interpreter will display an error message and it will terminate the program.
* If we want the program to continue with the execution of the remaining code, then we should try to catch the exception.
* The Exception handling mechanism performs the following tasks:

1. Find the problem.
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

* The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.
* Exception handling mechanism includes the key words: ***try****,* ***catch****,* ***throw****,* ***throws***and***finally****.*

**ADVANTAGES OF EXCEPTION-HANDLING:**

* It ensures safer termination of a program.
* It demands proper input entry or proper handling of a program.
* User has a chance to correct the accidental mistakes while running a program.
* The program becomes robust and more user-friendly.
* With this mechanism the working code and the error-handling code can be disintegrated.
* It allows different handling code-blocks for different types of errors.

**TYPES OF EXCEPTION**

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

**Checked Exception**

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

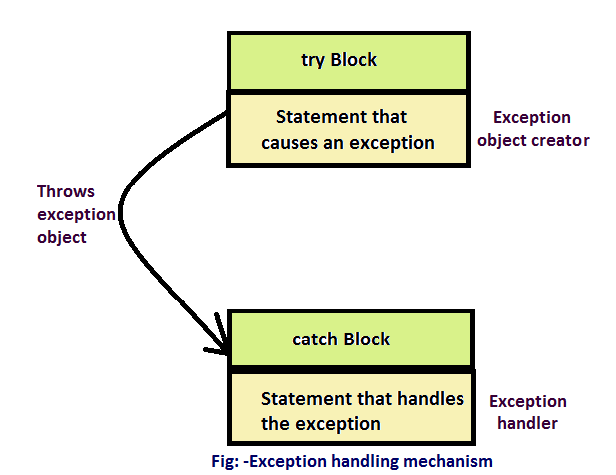
**Unchecked Exception**

The classes that extend RuntimeException are known as unchecked exceptions. e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

**Error**

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

**SYNTAX OF EXCEPTION HANDLING MECHANISM:**

**Syntax:**Try-Catch-Finally Blocks

try  
{

          Statements that may generate the exception

}  
catch(Exception-Type1 a)

{

          Statements to process the exception ‘a’

}  
catch(Exception-Type2 b)

{

          Statements to process the exception ‘b’

}

….

finally  
{

Statements to be executed before exiting exception handler

}

**Try - Block:**

* “try” block contains the code in which runtime error may occur.
* It throws Exception object.
* A try block must be followed by at least one *catch*block or a *finally* block.
* But, it can be followed by multiple catch blocks.
* A try block allows another try block within.

**Catch - Block:**

* “catch” block contains the handling code for runtime errors i.e. when run time error occurs, instead of terminating the program, control comes to “catch” block.
* Program control enters into “catch” block only when the corresponding runtime error occurs.
* After executing the handling code the control continues with the remaining part of the program.
* Multiple “catch” blocks can be written to handle different errors. In a “catch” block the exception type must be mentioned.

**Finally - block:**

* The “finally” block is always executed, regardless of whether or not an exception is thrown.
* It is recommended for actions like closing file streams and releasing system resources.
* Writing the finally block for try-block is optional unless there is no catch-block.
* If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and
* finally-block will be executed after the try-block.

**SYSTEM DEFINED EXCEPTIONS:**

All exceptions are classes. They are subclasses of either ‘Exception’ or ‘RuntimeException’ class. We can also instantiate them. Some common **system-defined exceptions** are listed below.

|  |  |
| --- | --- |
| **Exception Type** | **Cause of Exception** |
| ArithmeticException | Caused by math errors such as division by zero |
| ArraylndexOutOfBoundsException | 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 |
| lOException | 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 |
| OutOfMemory Exception | 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 |
| StackOverFlowExceptlon | Caused when the system runs out of stack space |
| StringlndexOutOfBoundsExceptlon | Caused when a program attempts to access a nonexistent character position in a string |

**EXAMPLES:**

**PROGRAM 1:** (ArithmeticException)

* The following program demonstrates exception handling.
* If zero is supplied for the second argument through command line, the code attempts to divide the first value with zero which raises a runtime error.
* But, as we have used exception handling mechanism here with appropriate catch block, the exception will be caught and the program will not be abruptly terminated.

class Test

{

            public static void main(String as[])

            {

                        int a=Integer.parseInt(as[0]);

                        int b=Integer.parseInt(as[1]);

                        try

                        {

                            System.out.println(“Division=”+ (a/b));

                        }

                        catch(ArithmeticException e)

                        {

                        System.out.println("Please check the Denominator”);

                        System.out.println(e);

                        }

            } }

The above program produces the following output.

**Output 1:** (No Exception)

D:\java>java Test a=6 b=2  
Division =3

**Output 2:** (With Exception)

D:\java>java Test a=6 b=0  
Please check the Denominator

java.lang.ArithmeticException: / by zero

**PROGRAM 2**: (**ArrayIndexOutOfBoundsException**)

The following program demonstrates catching **ArrayIndexOutOfBoundsException**. It is raised in a program when we use invalid index i.e. an index value i.e. greater than or equal to its size is used.

class Test

{

   public static void main(String as[])

   {

      int a[]={10,20,30,40};

      try

      {

         for(int i=0;i<10;i++)

          System.out.print("\t"+a[i]);

      }

      catch(ArrayIndexOutOfBoundsException ae)

      {

         System.out.println("Array has "+a.length+" elements only");

      } }}

The above program produces the following output.

   10      20      30      40

   Array has 4 elements only

**MULTIPLE CATCH BLOCK:**

* Several types of exceptions may arise when executing the program.
* They can be handled by using multiple catch blocks for the same try block.
* In such cases, when an exception occurs the run time system will try to find match for the exception object with the parameters of the catch blocks in the order of appearance.
* When a match is found the corresponding catch block will be executed.
* If there is no matching catch block the program will be terminated abruptly.
* Multiple catch blocks are necessary because handling code is different for different types of errors.

**Example:**

The Following program uses Multiple catch blocks and code in a catch block will be executed only when the corresponding errors occurs. Sample output is also shown.

class Test

{

   public static void main(String as[])

   {

      try

      {

        int x=Integer.parseInt(as[0]);

        int y=Integer.parseInt(as[1]);

        System.out.print("Division="+ (x/y));

      }

      catch(ArithmeticException ae)

      {

         System.out.println("Denominator was zero");

      }

      catch(NumberFormatException ne)

      {

         System.out.println("Please supply numbers");

      }

      catch(ArrayIndexOutOfBoundsException ie)

      {

         System.out.println("Please supply two arguments");

      }   } }

The above program produces the following output in different runs.

c:\jdk1.5\bin>java Test 8 0

              Denominator was zero

c:\jdk1.5\bin>java Test xx yy

              Please supply numbers

c:\jdk1.5\bin>java Test 5

              Please supply two arguments

**USING FINALLY STATEMENT:**

* The finally block is always executed, regardless of whether or not an exception is thrown.
* It is recommended for actions like closing file streams and releasing system resources.
* Writing the finally block for try-block is optional unless there is no catch-block.
* If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block.
* The finally block may be added after the catch block, or after the last catch block.

SYNTAX:

|  |  |
| --- | --- |
| try {    Statements that may generate    the exception  } finally {   Statements to be executed before exiting  exception handler  }  Moves to finally block but does not handle the exception | try {     Statements that may generate the     exception  } catch(Exception-Type1 a)  {    Statements to process the exception ‘a’  } …. //  other catch blocks  …. finally {   Statements to be executed before exiting   exception handler  }  Handles the exception and moves to finally block |

**EXAMPLE:**

class Test

{

            public static void main(String as[])

            {

                        int a=Integer.parseInt(as[0]);

                        int b=Integer.parseInt(as[1]);

                    try

                        {

                            System.out.println(“Division=”+ (a/b));

                        }

                        catch(ArithmeticException e)

                        {

                        System.out.println("Please check the Denominator”);

                        }

finally

                        {

                                    System.out.println("Inside finally block”);

                        }

            }}

The above program produces the following output.

**Output 1:**  (No Exception)

D:\java>java Test 6 2  
Division =3

Inside finally block // Finally block

**Output 2:** (With Exception)

D:\java>java Test 6 0  
Please check the Denominator // catch block

Inside finally block // Finally block

**THROWING OUR OWN EXCEPTION: (USER DEFINED EXCEPTION):**

* Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors that we might encounter.
* While defining a user defined exception, we need to take care of the following aspects:
* The user defined exception class should be a sub class of *Exception* class (mandatory).
* The toString() method should be overridden in the user defined exception class in order to display Meaningful information about the exception (Suggested).

**Throw:**

‘throw’ is a keyword used in exception handling mechanism. It is used to raise the errors manually. To throw errors, we need to first create the instances of Exceptions.

* The user defined exceptions must be explicitly thrown using the *throw* statement whose syntax is,

*throw ThrowableInstance;*

* ThrowableInstance must be an object of type Throwable or a subclass of Throw able.

**Syntax:**The following is the syntax for creating user-defined exception

class exceptionclassname extends *Exception*  
{  
            exceptionclassname(parameter)  
            {

                        //statements  
             }

              public String toString()  
             {  
                   return String

             }

}

**Example:**

For example the below program creates a user defined exception named “InvalidAge” and it is thrown when the value for age is invalid i.e. less than or equal to zero or greater than 100. This exception is caught as we have provided the appropriate catch block.

class InvalidAge extends *Exception  //*Creating own exception  
{  
             public String toString()  
             {  
                   return new String(“The age is not valid”);

             }

}

class Test

{

   public static void main(String as[])

   {

      int age = Integer.parseInt(as[0]);

      if (age<=0 || age >100)

      {

 **throw new InvalidAge();**

          System.out.print("Hello”);

      }

      catch(InvalidAge a)

      {

         System.out.println("Age not valid”);

      }

   }}

The above program produces the following output:

c:/jdk1.5/bin> java Test 120

Age not valid

# Difference between final, finally and finalize:

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |
| --- | --- | --- |
| **final** | **finally** | **finalize** |
| Final is used to apply restrictions on class, method and variable.  Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| Final is a keyword. | Finally is a block. | Finalize is a method. |

**APPLET PROGRAMMING:**

**Applet:**

* Applet is a special type of Java program that is used in web applications.
* Applets are embedded within a HyperText Markup Language (HTML) document.
* Applets provide a way to give life to a web page.
* Applets can be used to handle client-side validations.
* Browsers are required for their execution.
* Applets allow event-driven programming.

**Working of Java Applet**

* Java applets like Java programs consist of one or more class definitions.
* Once compiled, these classes are stored as files with a .class extension and they consist of Java bytecode.
* Java bytecode is created by a Java-compatible compiler.
* Unlike Java applications, applets are executed within a Java-enabled web browser such as Internet Explorer or tools like appletviewer.
* Applets are embedded within an HTML document via the APPLET tag that references the Java applet's compiled .class file.

**Types of Applets:**

Web pages can contain two types of applets which are named after the location at which they are stored.

1.      Local Apple

2.      Remote Applet

**Local Applets:**

* A local applet is the one that is stored on our own computer system.
* When the Web-page has to find a local applet, it doesn't need to retrieve information from the Internet.
* A local applet is specified by a path name and a file name as shown below in which the codebase attribute specifies a path name, whereas the code attribute specifies the name of the byte-code file that contains the applet's code.

*<applet   codebase="MyAppPath" code="MyApp.class" width=200 height=200> </applet>*

**Remote Applets:**

* A remote applet is the one that is located on a remote computer system.
* This computer system may be located in the building next door or it may be on the other side of the world.
* No matter where the remote applet is located, it's downloaded onto our computer via the Internet.
* The browser must be connected to the Internet at the time it needs to display the remote applet.
* To reference a remote applet in Web page, we must know the applet's URL (where it's located on the Web) and any attributes and parameters that we need to supply.
* A local applet is specified by a URL and a file name as shown below.

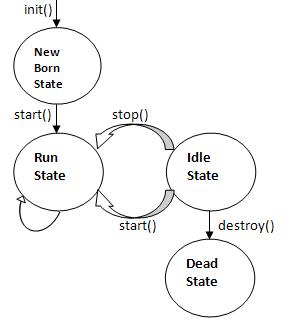
*<applet   codebase="http://www.apoorvacollege.com" code="MyApp.class" width=200 height=200> </applet>*

**LIFE CYCLE OF APPLET:**

* Applet runs in the browser and its lifecycle-methods are called by JVM at its birth, its death and when it is running or idle.
* When an applet is made to work, it undergoes a series of changes as discussed below.
* Every Applet can be said to be in any of the following states

1. New Born state
2. Running state
3. Idle state
4. Dead or Destroyed state

The following figure represents the life cycle of the Applet

[](http://2.bp.blogspot.com/-OLmVuz8TFDc/TV_BQ99DaYI/AAAAAAAABKg/gawAsvjPfDU/s1600/s1003.jpg)

**New Born State**

* An applet enters into initialization state when the applet is first loaded into the browser by calling the init() method.
* The init() method is called only one time in the life cycle on an applet.
* The init() method retrieves the parameters through the PARAM tag of html file.
* At this stage, we may create Objects needed by applets, setup initial values, load images or fonts and setup colors.
* After the initialization of the init() method user can interact  with the Applet.
* We can override this method.

Syntax:

*public void init()*

*{  
        Statements*

*}*

**Running State:**

* After initialization, this state will automatically occur by invoking the start method of applet.
* The running state can be achieved from idle state when the applet is reloaded.
* This method may be called multiple times when the Applet needs to be started or restarted.
* If the user leaves the applet and returns back, the applet may restart its running.
* We can override this method.

Syntax:

*public void start()*

*{  
        Statements*

*}*

**Idle State:**

* The idle state will make the execution of the applet to be halted temporarily.
* Applet moves to this state when the currently executed applet is minimized or when the user switches over to another page. At this point the stop method is invoked.
* From the idle state the applet can move to the running state.
* The stop() method can be called multiple times in the life cycle of applet.
* We can override this method.

Syntax:

*public void stop()*

*{  
        Statements*

*}*

**Dead State:**

* When the applet programs terminate, the destroy function is invoked which brings an applet to its dead state.
* The destroy() method is called  only one time in the life cycle of Applet like init() method.
* In this state, the applet is removed from memory.
* We can override this to cleanup resources.

Syntax

*public void destroy()*

*{  
        Statements*

*}*

**Display State:**

* The applet is said to be in display state when the *paint* or *update* method is called.
* This method can be used when we want to display output in the screen.
* This method can be called any number of times.
* Overriding paint() method is a must when we want to draw something on the applet window.
* paint() method takes *Graphics* object as argument
* paint() method is automatically called each time the applet window is redrawn i.e. when it is maximized from minimized state or resized or when other windows uncover overlapped portions.
* It can also be invoked by calling “repaint()” method.

Syntax:

*public void paint(Graphics g)*

*{*

*Statements  
}*

**GRAPHICS PROGRAMMING:**

**Steps to Create and Execute Applet:**

* An applet is a Java program that runs in a Java-compatible browser such as Internetexplorer.
* This feature allows users to display graphics and to run programs over the Internet.
* An applet allows web documents to be both animated and interactive.

**Step 1: Import applet package and awt package:**

* To create an applet, our program must import the Applet class.
* This class is found in the *java.applet*package.
* The Applet class contains code that works with a browser to create a display area.
* We also need to import the *java.awt*package.
* "awt” stands for *“Abstract Window Toolkit*”.
* The java.awt package includes classes like *Graphics.*

**Step 2: Extend the Applet class:**

* Then, a class must be defined that inherits from the class ‘Applet’.
* It contains the methods to paint the screen.
* The inherited class must be declared public.

**Step 3: Override the paint method to draw text or graphics:**

* The paint method needs the Graphics object as its parameter.

*public void paint(Graphics g) { … }*

* The Graphics class object holds information about painting.
* We can invoke methods like drawLine(), drawCircle() and etc using this object.

**Syntax:  The program looks something like this.**

import java.applet.\*;

import java.awt.\*;

*public* class MyApplet *extends Applet*

 {

    public void *paint*(Graphics g)

    {

         g.drawString("Welcome to Apoorva”,50,50);

    }}

**Step 4: Compiling the Program:**

* After writing the program, we compile it using the command "javac MyApplet.java".
* This command will compile our code so that we now have MyApplet.class file.

**Step 5: Adding applet to HTML document:**

* To run the applet we need to create the HTML document.
* The BODY section of the HTML document allows APPLET tag.
* The HTML file looks something like this:

<HTML>  
<BODY>

 <APPLET  codebase="MyAppPath" code="MyApp.class" width=200 height=200> </APPLET>

</BODY>  
</HTML>

**Step 6: Running an applet:**

The applet can be run in two ways

1. Using appletviewer: To run the appletviewer,  type   *appletviewer filename.html*
2. Using web browser:  Open the web browser, type in the full address of html file

**APPLET TAGS AND ATTRIBUTES:**

* Applets are embedded in HTML documents with the <applet> tag.
* Any number of <applet> tags can be embedded in a single HTML file.
* It contains attributes that identify the applet to be displayed and, optionally, give the Web browser hints about how it should be displayed.
* The size-indicating attributes, such as height and width can be used inside the applet tag.
* Any number of <param> tags that contain application-specific parameters to be passed to the applet can be written within the <APPLET> tag.
* *Code*, *Width* and *Height* attributes are minimum requirements to place the applet on a web-page.

**Syntax:**

<APPLET   code="MyApp.class" width=200 height=200  [attribute=value] .. >

       [<param *name*=parameter-name *value*=“param-value”>]

       [<param *name*=parameter-name *value*=“param-value”>]

        …..

       </APPLET>

**Attributes:**

* Attributes are name and value pairs that are interpreted by a Web browser or applet viewer.
* Attributes of the <applet> tag specify general features that apply to all applets, such as size and alignment.
* Three attributes, code, width, and height, are always required in the <applet> tag.
* All other attributes are optional.
* Attributes allowed in <APPLET> tag are listed below.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Value** | **Description** |
| align | URL | Deprecated − Defines the text alignment around the applet |
| alt | URL | Alternate text to be displayed in case browser does not support applet |
| archive | URL | Applet path when it is stored in a Java Archive ie. jar file |
| code | URL | A URL that points to the class of the applet |
| codebase | URL | Indicates the base URL of the applet if the code attribute is relative |
| Height, width | pixels | The initial width and height (in pixels) of the applet display area. |
| hspace | pixels | Deprecated − Defines the left and right spacing around the applet |
| name | name | Defines a unique name for the applet |
| object | name | Specifies the resource that contains a serialized representation of the applet's state. |
| title | test | Additional information to be displayed in tool tip of the mouse |
| vspace | pixels | Deprecated − Amount of white space to be inserted above and below the object. |