###### Unit-IV

**Exception Handling** –Dealing with errors- benefits of exception handling- the classification of exceptions –exception hierarchy- checked exceptions and unchecked exceptions- usage of try- catch-throw-throws and finally-rethrowing exceptions- exception specification- built in exceptions- creating own exception sub classes.

###### Introduction

* An exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction.

Or

* An abnormal condition that disrupts Normal program flow.
* There are many cases where abnormal conditions happen during program execution, such as
  + Trying to access an out - of –bounds array elements. o The file you try to open may not exist.
  + The file we want to load may be missing or in the wrong format.
  + The other end of your network connection may be non –existence.
* If these cases are not prevented or at least handled properly, either the program will be aborted abruptly, or the incorrect results or status will be produced.
* When an error occurs with in the java method, the method creates an exception object and hands it off to the runtime system.
* The exception object contains information about the exception including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error.
* **In java creating an exception object and handling it to the runtime system is called throwing an exception.**
* Exception is an object that is describes an exceptional ( i.e. error) condition that has occurred in a piece of code at run time.
* When a exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
* Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
* System generated exceptions are automatically thrown by the Java runtime system

General form of Exception Handling block try {

// block of code to monitor for errors

}

catch (*ExceptionType1 exOb*) {

// exception handler for *ExceptionType1*

}

catch (*ExceptionType2 exOb*) {

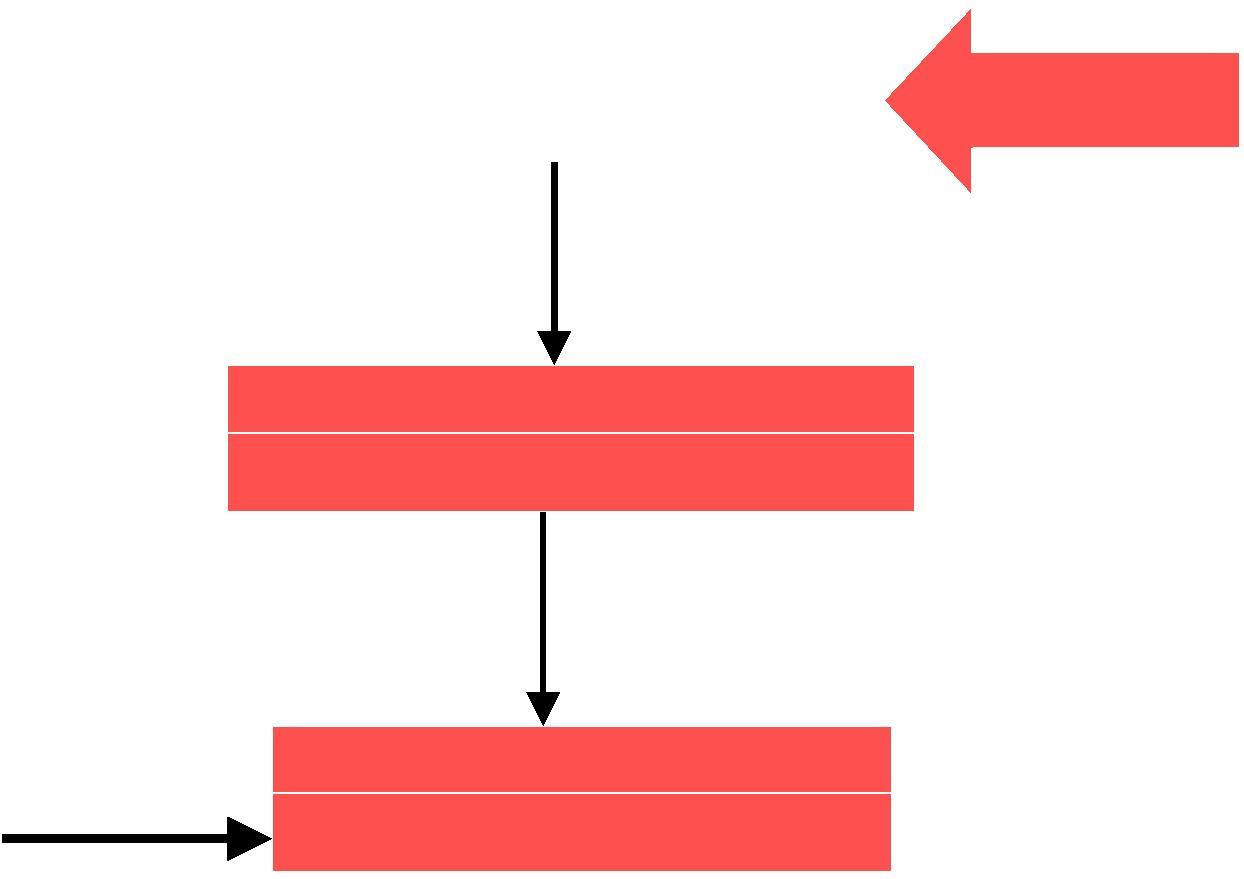
// exception handler for *ExceptionType2*

}

// ... finally {

// block of code to be executed before try block ends

}



No

Exception

Throws exception object

arise or

No

Catch Block

Exceptional Handler

appropriate Catch

block

Finally Block

Optional part

Try Block

 By using exception to managing errors, Java programs have have the following advantage over traditional error management techniques:

* Separating Error handling code from regular code.
* Propagating error up the call stack.
* Grouping error types and error differentiation.

For Example:

class Exc0 {

public static void main(String args[]) {

int d = 0;

int a = 42 / d;

}

}

When the Java run-time system detects the attempt to divide by zero, it constructs a

new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt

with immediately.

In this example, we haven’t supp the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the output generated when this example is executed.

java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**

###### Try and Catch Blocks

If we don’t want to prevent the programapthe to trap the exception using the try block. So we can place the statements that may causes an exception in the try block.

try

{

}



If an exception occurs within the try block, the appropriate exception handler that is associated with the try block handles the exception immediately following the try block, include a catch clause specifies the exception type we wish to catch. A try block must have at least one catch block or finally that allows it immediately.

catch block

The catch block is used to process the exception raised. A try block can be one or more catch blocks can handle a try block.

 Catch handles are placed immediately after the try block.

catch(exceptiontype e)

{

//Error handle routine is placed here for handling exception

}

Program 1

Class trycatch

{

public static void main(String args[])

{

int[] no={1,2,3};

try

{

System.out.println(no[3]);

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println(―Outof bounds‖);

}

System.out.println(―Quit‖);

}

}

###### Output

Out of the Range Quit

Program 2 Class ArithExce

{

Public static void main(String args[])

{

Int a=10; Int b=0; try

{

a=a/b;

System.out.println(―Won’t Print‖);

}

Catch(ArithmeticException e)

{

System.out.println(―Division by Zero error‖);

System.out.println(―Change the b value‖);

}

System.out.println(―Quit‖);

}

}

###### Output

Division By zero error Please change the B value Quit

Note:







A try and its catch statement form a unit. We cannot use try block alone.

The compiler does not allow any statement between try block and its associated catch block

Displaying description of an Exception

 Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

We can display this description in a println statement by simply passing the exception as an argument.

catch (ArithmeticException e) { System.out.println("Exception: " + e); a = 0; // set a to zero and continue

}

When this version is substituted in the program, and the program is run, each divide-by- zero error displays the following message:

* Exception: java.lang.ArithmeticException: / by zero

###### Multiple Catch Blocks

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block. The following example traps two different exception types:

// Demonstrate multiple catch statements. class MultiCatch {

public static void main(String args[]) { try {

int a = args.length; System.out.println("a = " + a); int b = 42 / a;

int c[] = { 1 }; c[42] = 99;

} catch(ArithmeticException e) { System.out.println("Divide by 0: " + e);

} catch(ArrayIndexOutOfBoundsException e)

{ System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}

This program will cause a division-by-zero exception if it is started with no commandline parameters, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an

**ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways: C:\>java MultiCatch

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero After try/catch blocks.

C:\>java MultiCatch TestArg a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException After try/catch blocks.

###### Throw Keyword

So far, we have only been catching exceptions that are thrown by the Java Run –Time systems. However, it is possible for our program to throw an exception explicitly, using the throw statement.

 Throw throwableInstance

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions

 There are two ways you can obtain a **Throwable** object:

* using a parameter into a **catch** clause
* creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace

// Demonstrate throw. class ThrowDemo { static void demoproc()

{ try {

throw new NullPointerException("demo");

} catch(NullPointerException e) { System.out.println("Caught inside demoproc."); throw e; // rethrow the exception

}

}

public static void main(String args[]) { try {

demoproc();

} catch(NullPointerException e) { System.out.println("Recaught: " + e);

}

}}

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown. Here is the resulting output:

 Caught inside demoproc.

 Recaught: java.lang.NullPointerException: demo

The program also illustrates how to create one of J close attention to this line:

 throw new NullPointerException("demo");

Here, **new** is used to construct an instance of **NullPointerException**. All of- Java’ in run-time exceptions have at least two constructors: one with no parameter and one that

takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object

 is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to

**getMessage( )**, which is defined by **Throwable**.

###### Throws Keyword

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You

do this by including a **throws** clause in the method’s **throws**clauselistsdeclar the types of exceptions that a method might throw. This is necessary for all exceptions,

except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a **throws** clause:

*type method-name(parameter-list)* throws *exception-list*

{

// body of method

}

 Here, *exception*-*list* is a comma-separated list of the exceptions that a method can throw

Program

class ThrowsDemo {

static void throwOne() throws IllegalAccessException

{ System.out.println("Inside throwOne."); throw new IllegalAccessException("demo");

}

public static void main(String args[]) { try {

throwOne();

} catch (IllegalAccessException e)

{ System.out.println("Caught " + e);

}

}

}

 Here is the output generated by running this example program:

 inside throwOne

 caught java.lang.IllegalAccessException

###### S.No. throw throws

1. Java throw keyword is used to explicitly Java throws keyword is used to declare an throw an exception. exception.
2. Checked exception cannot be propagated Checked exception can be propagated with using throw only. throws.
3. Throw is followed by an instance.
4. Throw is used within the method.
5. You cannot throw multiple exceptions.

###### Finally block

Throws is followed by class.

Throws is used with the method signature.

You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

 When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear

path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely.

This could be a problem in some methods. For example, if a method

opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

**finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that

might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

// Demonstrate finally. class FinallyDemo {

// Through an exception out of the method. static void procA() {

try {

System.out.println("inside procA"); throw new RuntimeException("demo");

} finally { System.out.println("procA's finally");

}

}

// Return from within a try block. static void procB() {

try {

System.out.println("inside procB"); return;

} finally { System.out.println("procB's finally");

}

}

// Execute a try block normally. static void procC() {

try {

System.out.println("inside procC"); } finally {

System.out.println("procC's finally");

}

}

public static void main(String args[]) { try {

procA();

} catch (Exception e) { System.out.println("Exception caught");

}

procB();

procC();

}

}

 In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**’s**try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the

**try** statement executes normally, without error. However, the **finally** block is still executed. *If a* ***finally*** *block is associated with a* ***try****, the* ***finally*** *block will be executed upon conclusion of the* ***try****.*

 Here is the output generated by the preceding program: inside procA

procA’s finally

Exception caught inside procB

procB’s finally inside procC procC’sly final

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

|  |  |
| --- | --- |
| **Java final example** |  |
| 1. class FinalExample{ 2. public static void main(String[] args){ 3. final int x=100; 4. x=200;//Compile Time Error   5. }} |  |  |

# Java finally example

* 1. class FinallyExample{
  2. public static void main(String[] args){
  3. try{

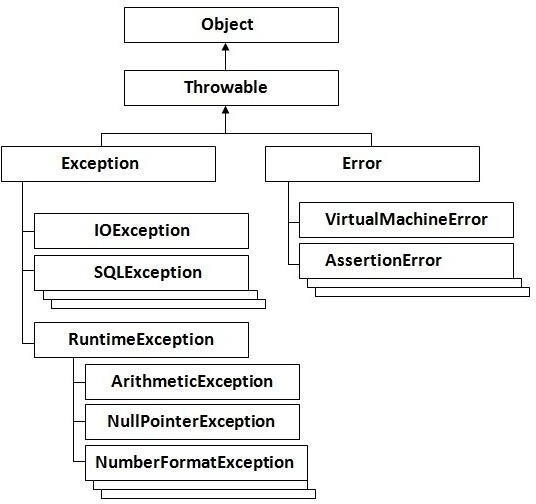
4. int x=300;

1. }catch(Exception e){System.out.println(e);
2. }
3. finally{System.out.println("finally block is executed");} 7. }}

# Java finalize example

1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc(); 9. }
9. }

# Hierarchy of Java Exception classes



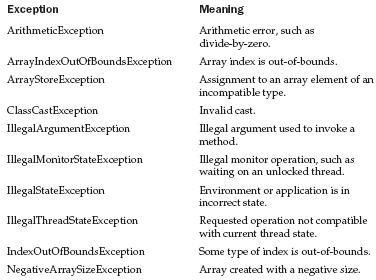
###### Java Built –In Exceptions

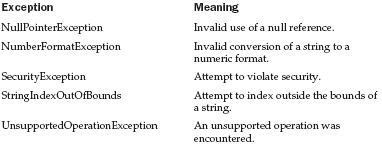
Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included **throws**list. inInthe language any of method’sJava,these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 10-1. Table

* 1. lists those exceptions defined by **java.lang** that must be included**throws**list in if that method can generate one of these exceptions and does not handle it itself. These are called

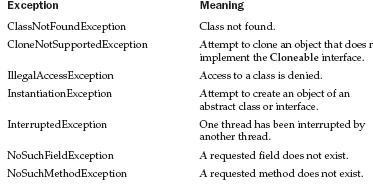
*checked exceptions.* Java defines several other types of exceptions that relate to its various class libraries

###### List of Unchecked exceptions





**List of Checked exceptions**



###### User defined exceptions

 We can create our own exception by extending exception class.

 The throw and throws keywords are used while implementing user defined exceptions

class ownExcepion extends Exception

{

ownException(String msg)

{

super(msg);

}

}

class test

{

public static void main(String args[]) int mark=101;

try

{

if(mark>100)

{

throw new ownException(―Marks>100‖);

}

}

catch(ownException e)

{

System.out.println (―Exception caughtr‖); System.out.println.(―e.getMessage());

}

finally

{

ystem.out.println(―Endof prg‖);

}

}

}