**Java Exception Handling**

* Introduction
* Runtime stack mechanism
* Default exception handling in Java
* Exception handling hierarchy
* Customized exception handling by using try/catch
* Control flow in try-catch
* Methods to print exception information
* try with multiple catches
* finally block
* Different between final, finally and finalize
* throw keyword
* throws keyword

**What is an exception**

An exception is an event which occurs during the execution of the program that disrupts the normal flow of program’s instructions.

An unexpected unwanted event that disturbs the normal flow of the program is called the exception.

Example : TypePuncturedException, SleepingException, FileNotFoundException

**What is the purpose of Exception Handling?**

It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of the program.

Exception handling does not mean repairing an exception, it is the concept of providing an alternative way to continue the rest of the program normally.

For example, the program requirement is to read data from a remote file location at London. At runtime, if that file is not available, the program should not terminate abnormally. Provide some local file to continue the test of the program normally. This way of defining alternative is the concept of exception handling.

try{

read data from remote file location at London

}catch(FileNotFoundException e){

Use local file and continue rest of the program normally

}

Note: In Java exception handling provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.

**Runtime Stack Mechanism**

class Test{

public static void main(String[] argos){

doStuff();

}

public static void doStuff(){

doMoreStuff();

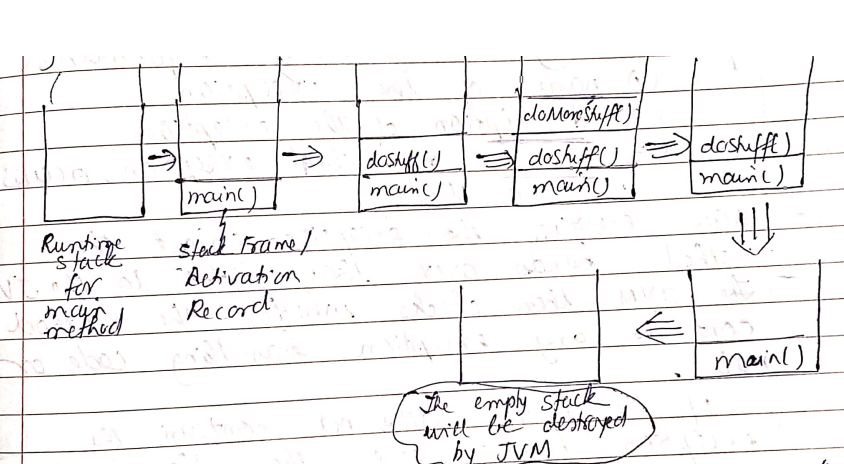
}

public static void doMoreStuff(){

System.out.println(“Hello”);

}

}



* For every thread the JVM will create a runtime stack, each and every method call performed by that thread will be stored in the corresponding stack.
* Each entry in the stack is called stack frame or activation record.
* After completing every method call, the corresponding entry from the stack will be removed
* After completing all the method calls the stack becomes empty and this empty stack will be destroyed by the JVM just before terminating the thread.

**Default Exception Handling in Java**

Inside a method if any exception occurs the method in which the exception is raised is responsible to create the exception object by including the following information

1. Name of the exception
2. Description of the exception
3. Location at which the exception occurs (stack trace)

After creating the exception object the method hands over that object to the JVM.

The JVM then checks whether the method contains any exception handling code or not.

* If the method does not contain the exception handling code then the JVM terminates that method abnormally and removes the corresponding entry from the stack.
* The JVM then identifies the caller method and checks whether the caller method

contains any handling code or not. If the caller method does not contain the handling code then the JVM terminates that caller method also abnormally and removes the corresponding entry from the stack.

* This process will be continued until the main() method and if the main() method also does not contain the handling code then the JVM terminates the main() method also abnormally and removes corresponding entry from the stack.
* Now JVM hands over the responsibility of exception handling to the default exception handler which is part of the JVM.
* The default exception handler prints the exception information in the following format and terminates program abnormally

Exception in thread “xxx” name of exception:

Description

Stack Trace

*Example 1:*

class Test{

public static void main(String[] argos){

doStuff();

}

public static void doStuff(){

doMoreStuff();

}

public static void doMoreStuff(){

System.out.println(10/0);

}

}

Exception in thread “main” java.lang.ArithmeticException:/ by zero

at Test.doMoreStuff()

at Test.doStuff(0

at Test.main()

*Example 2:*

class Test{

public static void main(String[] argos){

doStuff();

System.out.println(10/0);

}

public static void doStuff(){

doMoreStuff();

System.out.println(“Hi”);

}

public static void doMoreStuff(){

System.out.println(“Hello”);

}

}

The output will be:

Hello

Hi

Exception in thread main: / zero

at Test.main

Note: In a program if at least one method terminates abnormally then the entire program terminates abnormally - Abnormal Termination

However, if all the methods in a program terminates normally then the program terminates normally - Normal Termination

**Exception Hierarchy**

The *Throwable class* acts as the root for Java exception hierarchy.

The Throwable class defines two child classes

1. Exception
2. Error

*1. Exception*

Most of the times, the exceptions are caused by our program which are however, recoverable e.g. if the program requires to read data from a remote file locating at London; at runtime if the requested file is not available then runtime exception FileNotFoundException is thrown. If this exception occurs, a local file can be provided and continue with rest of the program normally.

try{

read data from remote file

locating at London

}catch(FileNotFoundException e){

use local file and continue normally

}

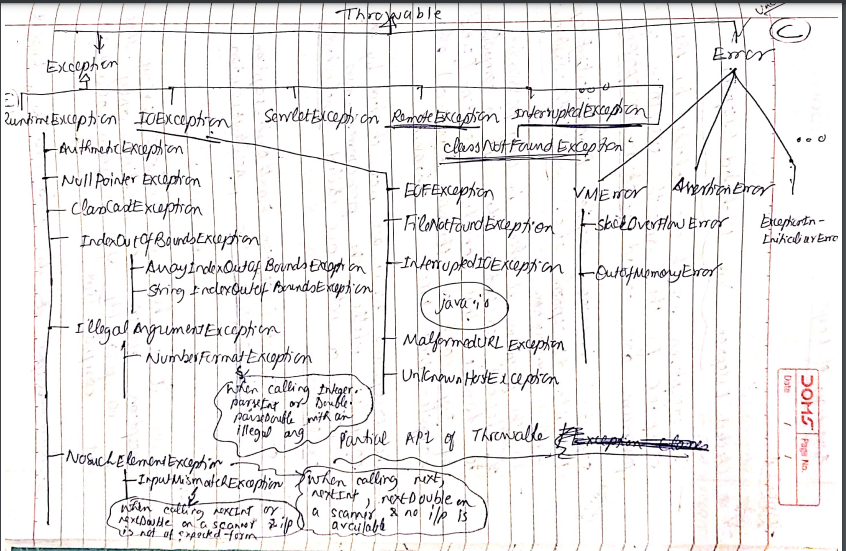
*2. Error*

Most of the times, errors are not caused by our programs, but are cause due to lack of

system resources.

Errors are not recoverable.

Example: If OutOfMemoryError occurs, the programmer cannot handle this error and hence terminated abnormally. The system admin or server admin is responsible for increasing the heap memory.



**Checked Exception versus Unchecked Exception**

The exceptions which are checked by the *compiler* for the smooth execution of the program are called *checked exceptions.*

Example: HallTicketMissingException, PenNotWorkingException, FileNotFoundException, DieselNotAvailableException etc.

In the program, if there is any possibility of raising a checked exception then it is compulsory to handle that checked exception (either by try-catch or throws keyword) otherwise a compile-time error is thrown.

The exceptions which are not checked by the compiler whether the programmer has handled or not are called *unchecked exceptions*

Example: ArithmeticException, EarthQuakeException, BuildingCollapseException, AccidentException

Note:

1. Whether it is checked or unchecked every exception occurs at runtime only. There is no possibility of exception occurring at compile-time.
2. RuntimeException and its child classes, Error and its child classes are unchecked exceptions. Except these all the remaining are checked exceptions

**Fully checked versus partially checked Exception**

A checked exception is said to be fully checked if and only if its child classes are also checked

E.g., IOException, InterruptedException

A checked exception is said to be partially checked if and only if some of its child classes are unchecked e.g., Exception class, Throwable class.

Note: The only possible partially checked exceptions in Java are

1. Exception and 2. Throwable

**Customized exception handling by using try-catch statement**

* It is highly recommended to handle exceptions
* The code which may raise an exception is called *risky code* and this code has to be defined inside a try block and the corresponding handling code is to be defined in the catch block.

| try{  risky code  }catch(Exception e){  //handling code  } |
| --- |

| **Without try-catch** | **With try-catch** |
| --- | --- |
| class Test{  public static void main(String[] args){  System.out.println(“Hello”);  System.out.println(10/0);  System.out.println(“Hi”);  }  } | class Test{  public static void main(String[] args){  System.out.println(“Hello”);  try{  System.out.println(10/0);    }catch(ArithmeticException e){  System.out.println(10/2);  }  System.out.println(“Hi”);  }  } |
| Abnormal Termination  O/P:  Hello  Exception in thread “main” java.lang.ArithmeticException: /zero  at Test.main() | Normal Termination  O/P:  Hello  5  Hi |

**Control flow in try-catch**

| try{  stmt1;  stmt2;  stmt3;  }catch(Exception e){  stmt4;  }  stmt5; |
| --- |

*Case 1:*

If there is no exception, the order of execution is : stmt1, stmt2, stmt3, stmt5 -> normal termination

*Case 2:*

If an exception is raised at stmt2 and the corresponding catch block matches, then the order is: stmt1, stmt4, stmt5 -> normal termination

*Case 3:*

If an exception is raised at stmt2 and the correspond catch block does not match then the order is: stmt1 -> abnormal termination

*Case 4:*

If an exception is raised at stmt4 or stmt5 then it is always and abnormal termination

Note:

1. Within the try block if anywhere an exception is raised then the rest of the try block will not be executed even though we handle that exception therefore, within the try block we have to take only risky code and the length of the try block should be as less as possible.
2. In addition to try block there may be the possibility of raising an exception in the catch and finally block
3. If any statement which is not part of a try block raises an exception then it is always an abnormal termination.

**Methods to print exception information**

The Throwable class defines the following methods to print exception information

| **Method** | **Printable Format** |
| --- | --- |
| printStackTrace() | name exception: description  stack trace |
| toString() | Name of exception: description |
| getMessage() | description |

Example:

| class Test{  public static void main(String[] args){  try{  System.out.println(10/0);  }catch(ArithmeticException e){  e.printStackTrace(); //1.  System.out.println(e); //System.out.println(e.toString()); //2.  System.out.println(e.getMessage()); //3.  }try |
| --- |

1. java.lang.ArithmeticException: /by zero

at Test.main

1. java.lang.ArithmeticException:/ by zero
2. / by zero

**try with multiple catch blocks**

The way of handling exceptions is different from one exception to another exception; therefore for every exception type it is highly recommended to take a separate catch block, that is, try with multiple catch blocks is possible and recommended.

| try{  risky code  }catch(Exception e) | try{  risky code  }catch(ArithmeticException e){  Perform arithmetic operations  }catch(SQLException e){  Use MySQL database instead of Oracle database  }catch(FileNotFoundException e){  Use local file instead of remote file  }catch(Exception e){  Default exception handling  } |
| --- | --- |
| NOT RECOMMENDED | HIGHLY RECOMMENDED |

**Conclusion:**

* If *try* with multiple *catch* block is used then the order of the catch blocks is very important.
* The child class exception must be written before the parent class exception.

| try{  risky code  }catch(Exception e){  }catch(ArithmeticException e){  } | try{  risky code  }catch(ArithmeticException e){  }catch(Exception e){  } | try{  risky code  }catch(ArithmeticException e){  }catch(ArithmeticException e){  } |
| --- | --- | --- |
| Compilation Error: java.lang.ArithmeticException  Exception has already been caught | Correct code | Compilation Error: Exception java.lang.ArithmeticException has already been caught |

* Two catch block for the same exception cannot be declared at the same time

**final, finally and finalize**

1. *final*

* final is a modifier that is applicable for classes, methods and variables
* If a class is declared as final then that class cannot be extended that is the child class for that class cannot be created that is inheritance is not possible for final classes
* If a method is final then that method cannot be overridden in the child class
* If a variable is declared as a final variable then that variable cannot be reassigned new values

1. *finally*

* finally is a block which is always associated with a try-catch statement to maintain clean-up code

| try{  risky code  }catch(Exception e){  handling code  }finally{  clean-up code  } |
| --- |

* The speciality of finally block is that this block is ALWAYS executed whether the exception occurs or not and whether handled or not handled

1. *finalize() method or java.lang.finalize()*

* It is a method which is always invoked by the garbage collector just before it destroys the object to perform clean-up activities.
* Once the finalize() method completes its execution the garbage collector destroys that object.

Note:

* The finally block is responsible to perform the clean-up activities related to try-catch block that is whatever resources was opened at the try block will be closed inside the finally block (try-level clean-up activities)
* Whereas the finalize() method is responsible to perform clean-up activities related to the object, that is, whatever resources associated with the object will be deallocated before destroying an object by using the finalize() method. (object-level clean-up activities)

**Various possible combinations of try-catch-finally**

* In the try-catch-finally the order is important
* A try statement must be followed by a catch or finally.
* A try without a catch or finally is invalid
* For a catch block, a try block is compulsory that is catch without try is invalid
* For a finally block a try block is compulsory that is finally block without try block is invalid
* Inside the try-catch-finally blocks we can nest try-catch-finally blocks that is nesting of try-catch-finally is allowed.
* Curly braces are compulsory for try-catch-finally blocks

| try {  }catch(X e){  } | try {  }catch(X e){  }catch Y e){  } |
| --- | --- |
| Valid | Valid |
| try {  }catch(X e){  }finally{  } | try {  }catch(X e){  }catch(X e){  } |
| Valid | Invalid |
| try {  }finally{  } | try {  }catch(X e){  }  try {  }catch(Y e){  } |
| Valid | Valid |
| try {  }catch(X e){  }  try {  }finally{  } | try{  } |
| Valid | CE: try without catch or finally |
| catch(X e){  } | finally{  } |
| CE: catch without try | CE: finally without try |
| try {  }finally{  }catch(X e){  } | try {  }  System.out.println(“Hello”);  catch(X e){  } |
| CE: catch without try | CE 1: try without catch or finally  CE 2: catch is without try |
| try {  }catch(X e){  }  System.out.println(“Hello”);  catch(Y e){  } | try {  }catch(X e){  }  System.out.println(“Hello”);  finally{  } |
| CE: catch without try | CE: finally without try |
| try{  try{    }catch(X e){  }  }catch(X e){  } | try{  try{    }  }catch(X e){  } |
| Valid | CE: try without catch |
| try{  try{    }finally{  }  }catch(X e){  } | try{    }  }catch(X e){  try{    }finally{  }  } |
| Valid | Valid |
| try{    }  }finally{  try{    }catch(X e){  }  } | try{    }  }catch(X e){  finally{  }  } |
| Valid | CE: finally without try |
| try{    }  }catch(X e){  }finally{  finally{    }  } | try{    }  }catch(X e){  }  }finally{  }finally{  } |
| CE: finally without try | CE: finally without try |

**The throw keyword**

Programmers can create an exception object explicitly and handover to the JVM manually. The keyword throw is used for this purpose.

Example:

class Test{

public static void main(String[] argos){

throw new ArithmeticException(“ / by zero”);

}

}

Here

The throw keyword is handing over created exception to the JVM manually

The new ArithmeticException constructor creates object explicitly

The output of the above code:

Exception in thread “main”: java.lang.ArithmeticException: /by zero

at Test.main

The main objective of the throw keyword is the handover the created exception object to the JVM manually/explicitly.

The result of the following program are exactly same

| class Test{  public static void main(String[] argos){  System.out.println(10/0);  }  } | class Test{  public static void main(String[] args){  throw new ArithmeticException(“ / by zero”);  }  } |
| --- | --- |
| Exception in thread “main”: java.lang.ArithmeticException: / by zero  at Test.main | Exception in thread “main”: java.lang.ArithmeticException: / by zero  at Test.main |
| In this case the main method is responsible to create the exception object and handover to the JVM | In this case, the programmer creates the exception object explicitly and handover to the JVM |

The throw keyword is highly recommended for user-defined exceptions or customized exceptions for example InsufficientFundException, InvalidPinException

*Case 1:*

throw e;

If the e reference variable refers to a null literal then a NullPointerException will be thrown

| class Test{  static ArithmeticException e = new ArithmeticException();  public static void main(String[] args){  throw e;  }  } | class Test{  static ArithmeticException e;  public static void main(String[] args){  throw e ;  }  } |
| --- | --- |
| Runtime Exception: ArithmeticException | Runtime Exception: NullPointerException |

*Case 2:*

After the throw statement, it is not allowed to write any other statement directly. If we do then a compile-time error “unreachable statement” is thrown.

| class Test{  public static void main(String[] args){  System.out.println(10/0);  System.out.println(“Hello”);  }  } | class Test{  public static void main(String[] args){  throw new ArithmeticException(“/ by zero”);  System.out.println(“Hello”);  }  } |
| --- | --- |
| Runtime Exception:  ArithmeticException: / by zero | Compilation Error: Unreachable statement |

*Case 3:*

The throw keyword is used to throw only Throwable types otherwise compile-time error is thrown [incompatible types]

| class Test{  public static void main(String[] args){  throw new Test();  }  } | class Test extends RuntimeException{  public static void main(String[] args){  throw new Test();  }  } |
| --- | --- |
| CE: incompatible types  found: Test  Required: java.lang.Throwable | Runtime : Exception in thread “Test”: Test  at Test.main |

**The throws keyword**

* If the program has the possibility of raising checked exception then it is compulsory to handle that checked exception else compile time error is thrown

CE: unreported exception XXX; must be caught or declared to be thrown

Example 1:

import java.io,\*;

class Test{

public static void main(String[] args){

PrintWriter pw = new PrintWriter(“abc.txt”);

pw.println(“Hello”);

}

}

CE: unreported exception java.io.FileNotFoundException; must be caught or declared to

be thrown

Example 2:

class Test{

public static void main(String[] args){

Thread.sleep(10000);//10 seconds

}

}

CE: unreported exception java.io.InterrupteException; must be caught or declared to

be thrown

This compile-time error can be handled by using the following two ways:

1. try-catch
2. throws

| **1. Using try-catch**  class Test{  public static void main(String[] args){  try{  Thread.sleep(10000);//10 secs  }catch(InterruptedException e){}  }  } | **2. Using throws**   * The throws keyword is used to delegate the responsibility of exception handling to the caller (JVM or another method) * The caller is responsible for handling that exception   class Test{  public static void main(String[] args) throws InterruptedException{    Thread.sleep(10000);//10 seconds  } } |
| --- | --- |

*Conclusion:*

1. The throws keyword is required only for checked exceptions and the use of throws keyword for unchecked exceptions is meaningless (no impact)
2. The throws keyword is required to convince the compiler
3. The use of throws keyword does not prevent abnormal termination of the program

Example 1:

class Test{

public static void main(String[] args) throws InterruptedException{

doStuff();

}

public static void doStuff() throws InterruptedException{

doMoreStuff();

}

public static void doMoreStuff() throws InterruptedException{

Thread.sleep(10000);

}

}

The above code is valid.

In case we omit the “throws” then we get the

CE: unreported exception java.io.InterrupteException; must be caught or declared to .

be thrown

Example 2:

class Test{

public static void main(String[] args) throws InterruptedException{

doStuff();

}

public static void doStuff(){

doMoreStuff();

}

public static void doMoreStuff(){

Thread.sleep(10000);

}

}

In the above code if at least one throws statement is removed then the code will not compiled

CE: unreported exception java.io.InterrupteException; must be caught or declared to be thrown

Summary

The throws clause:

1. We can use to delegate responsibility of exception handling to the caller(JVM or another method)
2. It is required only for checked exceptions and not for unchecked exceptions
3. It is required only to convince the compiler; does not prevent abnormal termination of program

**Four cases**

Case 1:

Example 1:

class Test throws Exception {

}

The above code is invalid. Dont apply throws with class

Example 2:

class Test{

Test() throws Exception{

}

public void m1() throws Exception{

}

}

The above code is valid.

The throws keyword is applicable only for methods and constructors but not for classes

Case 2:

Example 1

class Test{

public void m1() throws Test{ }

}

The above code will throw

CE: incompatible types

found: Test

required: java.lang.Throwable

Example 2

class Test extends RuntimeException{

public void m1() throws Test{

}

}

The above code is valid

The throws keyword can be used only for Throwable types; it is not applicable for throwing normal Java classes since a compile time error is thrown.

Case 3:

class Test{

public static void main(String[] argos){

throw new Exception();

}

}

The above code will throw compilation error : unreported exception: java.lang.Exception must be caught or declared to be thrown

class Test{

public static void main(String[] args){

throw new Error();

}

}

The above code will throw runtime error:

Exception in “main”: java.lang.Error

at Test.main

Case 4:

| class Test{  public static void main(String[] args){  try{  System.out.println(“Hello”);  }catch(ArithmeticException e){  }  }  } | class Test{  public static void main(String[] args){  try{  System.out.println(“Hello”);  }catch( Exception e){  }  }  } |
| --- | --- |
| The above code is valid (unchecked) | The above code is valid (partially checked) |
| import java.io.\*;  class Test{  public static void main(String[] args){  try{  System.out.println(“Hello”);  }catch(IOException e){  }  }  } | class Test{  public static void main(String[] args){  try{  System.out.println(“Hello”);  }catch(InterruptedException e){  }  }  } |
| Fully checked  CE: exception java.io.IOException is never thrown in body of corresponding try statement | Fully Checked  CE: exception java.io.IOException is never thrown in body of corresponding try statement |
| class Test{  public static void main(String[] args){  try{  System.out.println(“Hello”);  }catch(InterruptedException e){  }  }  } |  |
| Within a try block if there is no possibility of raising exception then catch block for this exception should not be written otherwise the compile time error is thrown:  Exception XXX is never thrown in body of corresponding try statement  This rule is applicable only for fully checked exceptions. |  |

Exception Handling Keywords Summary

1. try - to maintain risky code
2. catch - to maintain exception handling code
3. finally - to maintain cleanup code
4. throw- to handover our created exception object to JVM manually
5. throws - to delegate responsibilities of exception handling to the caller

**Custom Exceptions**

* Java also lets you create and use custom exceptions - classes of your own exception as per application need which will be used to represent errors.
* Normally, you create a custom exception to represent some type of error within your application - to give a new, distinct meaning to one or more problems that can occur within your code.
* There are three basic steps to create your own exception

Example:

Bank account balance

To have flexi-deposit functionality that is when account balance goes beyond 20K, a new deposit will be created.

*Step 1: Define the exception class*

Create a subclass of an existing exception class

public class AccountBalanceException extends Exception{

private double account;

public AccountBalanceException(double d){

super();

this.account = d;

}

//error message

public AccountBalanceException(String message){

super(message);

}

public double getAccount(){

return account;

}

}

Note: When you subclass any exception, you automatically inherit some standard

features from Throwable class such as

* Error message
* Stack trace
* Exception wrapping

*Step 2: Declare that your exception- producing method throws your custom exception*

import java.util.Scanner;

public class AccountManagement{

private double currentBal = 15000;

public static void main(String[] argos){

Scanner sc = new Scanner(System.in);

System.out.println(“Please enter amount to add in your balance”);

double newAmount = sc.nextDouble();

try{

}

}