Exception Handling in Java

=========================================================

The Exception Handling in Java is one of the powerful mechanism to handle the runtime

errors so that normal flow of the application can be maintained.

In this page, we will learn about Java exceptions, its type and the difference

between checked and unchecked exceptions.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program.

It is an object which is thrown at runtime.

What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException,

IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

------------------------------------

The core advantage of exception handling is to maintain the normal flow of the application.

An exception normally disrupts the normal flow of the application that is why we use

exception handling. Let's take a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5,

the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.

If we perform exception handling, the rest of the statement will be executed.

That is why we use exception handling in Java.

Do You Know?

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by

two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:

hierarchy of exception handling

Types of Java Exceptions

--------------------------------------

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the

unchecked exception. According to Oracle, there are three types of exceptions:

-->Checked Exception

-->Unchecked Exception

-->Error

Types of Java Exceptions

Difference between Checked and Unchecked Exceptions

1) Checked Exception

--------------------------------------

The classes which directly inherit Throwable class except RuntimeException and

Error are known as checked exceptions e.g. IOException, SQLException etc.

Simple example program on checked Exception:

============================================

package com.test;

import java.io.FileReader;

public class Test11 {

public static void main(String[] args) {

try

{

FileReader f=new FileReader("d:\test.txt");

}

catch(Exception e)

{

System.out.println("file not found");

}

}

}

2) Unchecked Exception:

-----------------------------------

The classes which inherit RuntimeException are known as unchecked exceptions e.g.

ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException ,numberformat etc.

Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

package com.exception;

public class Simple\_Example

{

public static void main(String args[])

{

/\*

\* int a=100,b=0,c; c=a/b; System.out.println(c);//arithmetic exception

\*/

/\*

\* String s=null; System.out.println(s.length());//null pointer exception

\*/

/\*

\* int arr[]=new int[5]; arr[10]=20; System.out.println(arr[10]);//arry

\* indexoutof bound exception

\*/

String s="abc";

int i=Integer.parseInt(s);//number format exception

System.out.println(i);

}

}

3) Error:

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError ,network failure,

harddisk error etc.

Java Exception Keywords: for only uncheckedexception

--------------------------------------------------------------

There are 5 keywords which are used in handling exceptions in Java.

Keyword Description:

1. try The "try" keyword is used to specify a block where we should place exception code.

The try block must be followed by either catch or finally. It means, we can't use try block alone.

2.catch The "catch" block is used to handle the exception. It must be preceded by try block

which means we can't use catch block alone. It can be followed by finally block later.

3.finally: The "finally" block is used to execute the important code of the program.

It is executed whether an exception is handled or not.

4.throw The "throw" keyword is used to throw an exception.

5.throws The "throws" keyword is used to declare exceptions.

It doesn't throw an exception. It specifies that there may occur an exception in the method.

It is always used with method signature.

Java Exception Handling Example:

--------------------------------------------------------------------

Let's see an example of Java Exception Handling where we using a try-catch

statement to handle the exception.

public class JavaExceptionExample

{

public static void main(String args[]){

try

{

//code that may raise exception

throws int data=100/0;

}

catch(ArithmeticException e)

{

System.out.println(e);

}

//rest code of the program

System.out.println("rest of the code...");

}

}

Test it Now

Output:

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

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable

throws a NullPointerException.

String s=null;

System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException.

Suppose I have a string variable that has characters, converting this

variable into digit will occur NumberFormatException.

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result

in ArrayIndexOutOfBoundsException as shown below:

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Simple program on Unchecked Exception:

=========================================

package com.exception;

public class Simple\_Example

{

public static void main(String args[])

{

int a=100,b=0,c;

c=a/b;

System.out.println(c);//arithmetic exception

String s=null;

System.out.println(s.length());//null pointer exception

int arr[]=new int[5];

arr[10]=20;

System.out.println(arr[10]);//arryindexoutof bound exception

String s="abc"; int i=Integer.parseInt(s); System.out.println(i);

}

}

example program on Runtime exceptions:

===================================

package com.test;

public class Demo

{

public static void main(String[] args) {

int i,j,k;

int a[]= {10,20,30,40};

String s=null;

String s1="ajay";

i=10;

j=0;

try

{

//k=i/j;

//a[5]=10;

//s.length();

int n=Integer.parseInt(s1);

}

catch(ArithmeticException e)

{

System.out.println(e);

}

catch(ArrayIndexOutOfBoundsException ae)

{

System.out.println(ae);

}

catch(NullPointerException ne)

{

System.out.println(ne);

}

catch(NumberFormatException nf)

{

System.out.println(nf);

}

finally

{

System.out.println("exception handling is done through");

}

}}

Simple program on try keyword:

====================================

package com.exception;

class Demo\_throw

{

static void fun()

{

try

{

throw new NullPointerException("demo");

}

catch (NullPointerException e)

{

System.out.println("Caught inside fun().");

throw e;

}

}

public static void main(String args[])

{

try

{

fun();

}

catch (NullPointerException e)

{

System.out.println("Caught in main.");

}

}

}

simple program on finally keyword:

======================================================

package com.exception;

public class Demo\_finally

{

public static void main(String args[])

{

int a=100,b=0,c;

try

{

c=a/b;

}

catch (ArithmeticException e)

{

System.out.println(e);

}

finally //it won't matter to execute or not exception

{

System.out.println("finally block is alwaysexecuted");

}

}

}

example program on multicatch statement:

===================================================

package com.exception;

public class Demo\_Multiplecatch {

public static void main(String[] args)

{

int a=100,b=0,c;

try

{

int arr[]=new int[5];

arr[10]=30;

c=a/b;

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic exception");

}

catch(ArrayIndexOutOfBoundsException ae)

{

System.out.println("ArrayOutOfBoundsException");

}

}

}

custom exception: OR User Defined Exception

=====================

Exception which is created by the user is kown as User Defined Exception

package com.test;

class InvalidAgeException extends Exception

{

InvalidAgeException()

{

System.out.println("your under Age");

}

}

public class Demo

{

public static void main(String[] args) throws InvalidAgeException

{

int age=17;

if(age<18)

{

throw new InvalidAgeException();

}

else

{

System.out.println("eligible for vote");

}

}

}

Multitasking:

Definition:executing multiple task simultaniously

Multithreading:

========================================================

Java Multithreading is mostly Ex: used in games, animation,Ticket booking etc.

Advantages of Java Multithreading:

---------------------------------------------------

1) It doesn't block the user because threads are independent and you can perform

multiple operations at the same time.

2) You can perform many operations together, so it saves time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to

utilize the CPU. Multitasking can be achieved in two ways:

----->Process-based Multitasking (Multiprocessing) ,operating system level multiprocessing is best

(Ex: writing proggram, listening music, downloading music etc)

---->Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

========================================================

Each process has an own address in memory. In other words, each process allocates

a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading registers,

memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

========================================================

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

-------------------------

A thread is a lightweight subprocess, the smallest unit of processing.

It is a separate path of execution.

Threads are independent. If there occurs exception in one thread,

it doesn't affect other threads.

It uses a shared memory area.

Java Multithreading

--------------------------

As shown in the above figure, a thread is executed inside the process.

There is context-switching between the threads. There can be multiple processes inside

the OS, and one process can have multiple threads.

Life cycle of a Thread (Thread States)

=========================================

1.New

2.Runnable

3.Running

4.Non-Runnable (Blocked)

5.Terminated

A thread can be in one of the five states. According to sun, there is only 4 states

in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM.

The java thread states are as follows:

1) New

The thread is in new state if you create an instance of Thread class but

before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method,

but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

==========================

There are two ways to create a thread:

--->By extending Thread class

--->By implementing Runnable interface.

Commonly used methods of Thread class:

===========================================

-->public void run(): is used to perform action for a thread.

--->public void start(): starts the execution of the thread.JVM calls the run()

method on the thread.

---->public void sleep(long miliseconds): Causes the currently executing thread to

sleep (temporarily cease execution) for the specified number of milliseconds.

------->public void join(): waits for a thread to die.

----->public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

----->public int getPriority(): returns the priority of the thread.

----->public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

--->public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

----->public boolean isAlive(): tests if the thread is alive.

----->public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

----->public void suspend(): is used to suspend the thread(depricated).

----->public void resume(): is used to resume the suspended thread(depricated).

---->public void stop(): is used to stop the thread(depricated).

public boolean isDaemon(): tests if the thread is a daemon thread.

public void setDaemon(boolean b): marks the thread as daemon or user thread.

public void interrupt(): interrupts the thread.

public boolean isInterrupted(): tests if the thread has been interrupted.

public static boolean interrupted(): tests if the current thread has been interrupted.

Runnable interface:

=====================================

The Runnable interface should be implemented by any class whose instances are intended

to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

A new thread starts(with new callstack).

The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

-------------------------------

class Multi extends Thread

{

public void run()

{

System.out.println("thread is running...");

}

public static void main(String args[])

{

Multi t1=new Multi();

t1.start();

}

}

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

==========================================================

class Multi3 implements Runnable

{

public void run()

{

System.out.println("thread is running...");

}

public static void main(String args[])

{

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1);

t1.start();

}

}

Output:thread is running...

Thread Scheduler in Java:

===============================================================================

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

Sleep method in java:

=======================================================================================

The sleep( ) method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep( ) method in java:

The Thread class provides two methods for sleeping a thread:

----public static void sleep(long milliseconds)throws InterruptedException

Example prograM;

package com.multithread;

class Demo\_Sleep extends Thread

{

public void run( )

{

for (int i = 0; i < 5; i++)

{

try

{

Thread.sleep(7000);// throws interrupted exception

}

catch (InterruptedException e)

{

System.out.println(e);

}

System.out.println(i);

}

}

public static void main(String args[])

{

Demo\_Sleep t1 = new Demo\_Sleep();

Demo\_Sleep t2 = new Demo\_Sleep();

t1.start();

t2.start();

}

}

Thread Execution is on Only one Thread:

===================================================================

No. After starting a thread, it can never be started again. If you does so,

an IllegalThreadStateException is thrown. In such case,

thread will run once but for second time,

it will throw exception.

Let's understand it by the example given below: the given example program

package com.multithread;

public class Demo\_Multiple extends Thread

{

public void run()

{

for(int i=0;i<=10;i++)

{

System.out.println("running...");

}

}

public static void main(String args[]) throws Exception

{

Demo\_Multiple t1=new Demo\_Multiple();

// Demo\_Multiple t2=new Demo\_Multiple();

t1.start();

// t2.start();

t1.start();

}

}

output:

==============

java.lang.IllegalThreadStateException

at java.lang.Thread.start(Thread.java:708)

at com.test.MyThread.main(MyThread.java:20)

Main thread example:

=========================

in this example first it executes Main thread first later it executes child thread

Always it gives priority to main thread only.

package com.test;

class Demo extends Thread

{

public void run()

{

for(int i=0;i<5;i++)

{

System.out.println("i value is"+i);

}

}

public static void main(String[] args)

{

Demo d=new Demo();

d.start();

for(int j=0;j<5;j++)

{

System.out.println("j value is"+j);

}

}

}

run() method calling :

=======================================================

In the following example to invoke run() method directly u will get the different output.'

thread won't work, because of no start() method .

Now this time executes as normal method call.

class TestCallRun2 extends Thread

{

public void run()

{

for(int i=1;i<5;i++)

{

try

{

Thread.sleep(500);

}

catch(InterruptedException e)

{

System.out.println(e);

}

System.out.println(i);

}

}

public static void main(String args[])

{

TestCallRun2 t1=new TestCallRun2();

TestCallRun2 t2=new TestCallRun2();

t1.start();

t2.start();

}

}

we can prevent a thread execution by using the following methods

1) yield()

2) join()

3) sleep()

yield();

===============================

causes to pass current executing the thread to give chance for waiting

threads for same prority. if there is waiting thread all waiting thread

low priority then same thread can continue it execution .

join();

========================================================================

if u complete the a perticular thread then we cann go for join() method

after it execution of the thread

it stop to execute the statement, then automatically die in that thrread.

\*\*\*\*\*to find that thread isalive or not by using isAlive() method

package com.multithread;

class TestJoinMethod1 extends Thread

{

public void run()

{

for(int i=1;i<=5;i++)

{

try

{

Thread.sleep(500);

}

catch(Exception e)

{

System.out.println(e);

System.out.println(i);

}

}

public static void main(String args[])

{

TestJoinMethod1 t1=new TestJoinMethod1();

TestJoinMethod1 t2=new TestJoinMethod1();

TestJoinMethod1 t3=new TestJoinMethod1();

t1.setPriority(4);

System.out.println("get priority"+t1.getPriority());

t2.setPriority(7);

System.out.println("get priority"+t2.getPriority());

System.out.println("get priority"+t3.getPriority());

t1.start();

try

{

t1.join();

}

catch(Exception e)

{

System.out.println("exception");

}

t2.start();

t3.start();

System.out.println(t1.isAlive()); // it returns false, because non exsistence

}

}

synchronization:

=======================

block level and method level only not for class level

while a thread executing synchronized method and the given object and

remaining threads not allow to execute any synchronized methods simlutaniously

on the same object.

but remaining thread are allow to execute non synchronized methods simultaniously.

class X

{

synchronized m1();

synchronized m2();

m3();

}

t1 --->m1() method to execute here t1 is on lock(m1())

t2---->m1() waiting

t3---->m2() waiting

t4--->m3() valid

lock concept on object.is implemented based on object but not based on method

class x

{

synchronized

{

Ex: Booking ticket

update, delete, insert, replace where state of object changing

}

non synchronized

{

Ex: check availability in booking reservation

object state won't be changed like Read() operation

}

}

Mutual Exclusive:

===========================================

Mutual Exclusive helps keep threads from interfering with one another while sharing data.

This can be done by three ways in java:

-->by synchronized method

-->by synchronized block

-->by static synchronization

Concept of Lock in Java

--------------------------

Synchronization is built around an internal entity known as the lock or monitor.

Every object has an lock associated with it. By convention, a thread that needs

consistent access to an object's fields has to acquire the object's lock before accessing them,

and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks

contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

class Table

{

void printTable(int n)

{ //method not synchronized

for(int i=1;i<=5;i++){

System.out.println(n\*i);

try

{

Thread.sleep(400);

}

catch(Exception e){System.out.println(e);

}

}

}

}

class MyThread1 extends Thread

{

Table t;

MyThread1(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(5);

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(100);

}

}

class TestSynchronization1

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

Output: 5

100

10

200

15

300

20

400

25

500

Java synchronized method

=========================

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object

and releases it when the thread completes its task.

//example of java synchronized method

class Table

{

synchronized void printTable(int n)

{ //synchronized method

for(int i=1;i<=5;i++){

System.out.println(n\*i);

try{

Thread.sleep(400);

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

}

}

}

class MyThread1 extends Thread

{

Table t;

MyThread1(Table t){

this.t=t;

}

public void run(){

t.printTable(5);

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t){

this.t=t;

}

public void run(){

t.printTable(100);

}

}

public class TestSynchronization2

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

Output: 5

10

15

20

25

100

200

300

400

500

Synchronized Block:

=========================

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines,

you can use synchronized block.

If you put all the codes of the method in the synchronized block,

it will work same as the synchronized method.

Points to remember for Synchronized block

------------------------------------------------

Synchronized block is used to lock an object for any shared resource.

Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

synchronized (object reference expression)

{

//code block

}

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

class Table

{

void printTable(int n)

{

synchronized(this)

{//synchronized block

for(int i=1;i<=5;i++)

{

System.out.println(n\*i);

try

{

Thread.sleep(400);

}

catch(Exception e)

{System.out.println(e);}

}

}

}//end of the method

}

class MyThread1 extends Thread{

Table t;

MyThread1(Table t){

this.t=t;

}

public void run(){

t.printTable(5);

}

}

class MyThread2 extends Thread{

Table t;

MyThread2(Table t){

this.t=t;

}

public void run(){

t.printTable(100);

}

}

public class TestSynchronizedBlock1{

public static void main(String args[]){

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

Test it Now

Output:5

10

15

20

25

100

200

300

400

500

InterThread communication OR Co-operation:

========================================

Two threads communicates with each other with

wait(), notify(), notifyall() method

is known as InterThread communication or co-operation.

In Multithreading First thread wants to implements the code in synchronized mode then

that thread send wait() method, after that it release the lock and it should be

in the waiting state, other thread (Second thread) completes implementation then after it

sends notify()method and at the same time it release the lock on method, then the First

thread start the execution over on method.

package com.test;

class ThreadB extends Thread

{

int total=0;

public void run()

{

for(int i=1;i<=100;i++)

{

total=total+i;

}

}

}

public class ThreadA1

{

public static void main(String[] args) {

ThreadB t=new ThreadB();

t.start();

//t.join() or t.sleep(1000)

System.out.println(t.total);

}

}

output: 0

zero comes because the main thread by default executes first

OR

if remove the comment on join or sleep() method , then main thread

temporerly blocked state then mean while child thread executes

and it retuens the value.

output:5050

why we can go to wait() and notify() method becuase if we write sleep() and

or join() then temporerly it stops the execution in main thread , but it is not

a good programming ,

becuase in that time lakhs of code execution is pending so that it decrease the

performance of the program.

to overcome this it means to increase the performance we need togo for

wait() or notify() or notifyall() method.

wait() and notify() methods:

===================================

package com.test;

class ThreadB extends Thread

{

int total=0;

public void run()

{

for(int i=1;i<=100;i++)

{

total=total+i;

}

this.notify();

}

}

public class ThreadA1

{

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

ThreadB t=new ThreadB();

t.start();

t.wait();

System.out.println(t.total);

}

}

Error because there is no synchronized area.

output:

Exception in thread "main" Exception in thread "Thread-0" java.lang.IllegalMonitorStateException

at java.lang.Object.notify(Native Method)

at com.test.ThreadB.run(ThreadA1.java:12)

java.lang.IllegalMonitorStateException

at java.lang.Object.wait(Native Method)

at java.lang.Object.wait(Object.java:502)

at com.test.ThreadA1.main(ThreadA1.java:21)

Adding Synchronized Area:

======================================================

package com.test;

class ThreadB extends Thread

{

int total=0;

public void run()

{

synchronized(this)

{

for(int i=1;i<=100;i++)

{

total=total+i;

}

this.notify();

}

}

}

public class ThreadA1

{

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

ThreadB t=new ThreadB();

t.start();

synchronized(t)

{

t.wait();

}

System.out.println(t.total);

}

}

DeadLock:

===========

two thread are waiting for each other for ever such type of infinte waiting is called

Dead lock mechanism.

Daemon thread:

==================

Garnage collector

background execution

the threads which are executinh in the backgroud are called daemon threads.

Example: Garbage collector

Singnal Dispatcher

Attach listner etc

The main Objective of Daemon thread is to provide support for non daemon threads

(main Thread). Ex: if main thread runs low memory then JVM runs garbage collector

todispatch useless objects , so that no of bytes of free memory will be improved.

with this free memory main thread is continue its execution,