Exception Handling

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

* In a program, we can have 3 types of errors.
1. syntax error
2. runtime error
3. logical error
* syntax errors, also called compile-time errors.
* runtime errors, also called exceptions.
* syntax erros will be shown by the compiler.
* exceptions will be shown by the JVM.
* logical errors must be identified manually, based on the input and the output.
* syntax errors will occur, because of the coding mistakes.
* exceptions will occur, because of the data given by the user at rutime.
* To identify logical errors, we do unit testing.
* The default flow is, at runtime if an application gets unexpected error, the application execution is immediately terminated. If it is a server-side application, then it becomes unavailable for the users.
* The other possibility is, at runtime if an application gets unexpected error, we can handle those errors gracefully, and we can continue the execution of the application, without termination.
* To implement the above, we have to use a mechanism called Exception handling.

what happens when an exception occurs?

 

Exception handling keywords:

1. try
2. catch
3. finally
4. throw
5. throws

 try {

 //the code from which exception

 //expected is kept here

 }

 catch(Exception ex) {

 //the code, which can

 //handle the exception is kept here

 }

 finally {

 //the code, that must execute

 //irrespective of the exception

 //shoud be kept here

 }

Displaying exception in catch block:

* It can be done in 3 ways.
1. System.out.println(ex)
2. System.out.println( ex.getMessage() );
3. ex.printStackTrace()
* first way, will display exception classname and message
* second way, will display only message of the exception.
* third way, will display stack trace of the exception.
* stack trace of the exception contains, 4 details.
1. exception class name
2. message
3. method call stack
4. location.
* The output of the first way, looks like below.

 java.lang.ArithmeticException: / by zero

* The output of the second way, looks like below.

 / by zero

* The output of the third way, looks like below.

 java.lang.ArithmeticException: / by zero

 at in.ashokit.demo.ClassA.divide(Main.java:6)

 at in.ashokit.demo.Main.main(Main.java:18)

defining try and catch blocks:

* When you define a try block, you must also define a corresponding catch block or a finally block or both.



* If an exception occurs in try block, the remaining statements of the try block are not executed. The

control jumps to the nearest catch block.



* If no exception occurs in the try block, then catch block will not be executed and directly and the control jumps to the finally block.



* If you write any statement after the finally block, then those statements will be executed, irrespective of the exception.



* If we write return statement in try block or in catch block, then also finally block will be executed.

 **try** {

 **double** c = a / b;

 System.***out***.println(c);

 }

 **catch**(Exception ex) {

 ex.printStackTrace();

 **return**;

 }

 **finally** {

 System.***out***.println("I am executing..");

 }

* If exit statement is added in try block or in catch block then finally block will not be executed.
* **try** {
* **double** c = a / b;
* System.***out***.println(c);
* }
* **catch**(Exception ex) {
* ex.printStackTrace();
* System.*exit*(1);
* }
* **finally** {
* System.***out***.println("I am executing..");
* }

exception classes hierarchy:



* The exceptions which are known to the compiler are checked exceptions and the exceptions which are un-known to the compiler are unchecked exceptions.
* The compiler knows that a checked exception my occur at runtime.
* For checked exceptions, at compile-time, the compiler checks to see whether a programmer is handling exception with try and catch blocks or atleast propagating the exception to the caller or not. If not done, then the compiler generates an error.
* For un-checked exceptions, at compile-time, the compiler doesn’t check to see whether a programmer is handling exception with try and catch blocks or atleast propagating the exception to the caller or not.

try with multiple catch blocks:



ex2:



* In this example, there is no catch block to handle ArrayIndexOutOfBoundsException. So, the exception is propagated to the caller method.
* If caller method is also not handling this exception then, it reaches to JRE. Then JRE will terminate the program execution and prints the exception stack trace.

ex3:

**class** A {

 **void** m1() {

 **int**[] arr = **new** **int**[5];

 **try** {

 System.***out***.println("hello");

 arr[5] = 90;

 System.***out***.println("Done");

 }

 **catch**(ArithmeticException ex) {

 System.***out***.println(ex);

 }

 **catch**(NullPointerException ex) {

 System.***out***.println(ex);

 }

 }

}

**public** **class** Main {

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

 A aRef = **new** A();

 **try** {

 aRef.m1();

 }

 **catch**(Exception ex) {

 System.***out***.println(ex.getMessage());

 }

 }

}

* In the above example, exception is not handled in the method where exception is raised. But it is handled in the caller method.

ex4:

**class** A {

 **void** m1() {

 **int**[] arr = **new** **int**[5];

 String str = **null**;

 **try** {

 System.***out***.println("hello");

 str = str.toUpperCase();

 arr[5] = 90;

 System.***out***.println("Done");

 }

 **catch**(ArithmeticException ex) {

 System.***out***.println(ex);

 }

 **catch**(NullPointerException ex) {

 System.***out***.println(ex);

 }

 **catch**(Exception e) {

 System.***out***.println(e.getMessage());

 }

 }

}

**public** **class** Main {

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

 A aRef = **new** A();

 aRef.m1();

 }

}

* In the above example, the second catch block will execute.

ex5:

 writing the catch blocks for specific exceptions, followed by writing the catch block for generic exception is valid.

But writing the catch block for generic exceptions, followed by writing the catch block for specific exception is a compile-time error. The compile-time error is “unreachable catch block”.



multi-exception catch block:



nested try block:

 try {

 //line1

 //line2

 try {

 //line3

 //line4

 }

 catch(Exception ex) {

 //line5

 //line6

 }

 }

 catch(Exception ex) {

 //line7

 //line8

 }

* if an exception occurs in inner try block, then inner catch block will handle that exception.
* If that inner catch block is unable to handle it, then outer catch block will handle the exception.
* if an exception occurs in the outer try block, then only outer catch block can handle that exception.

custom exceptions:

* user defined exception classes are called custom exceptions.
* To create custom checked exception, extend Exception class.
* To create custom un-checked exception, extend RuntimeException class.

ex1:

 public class InvalidAgeException extends Exception

 {

 }

ex2:

 public class NameNotFoundException extends RuntimeException {

}

* If you wish to display only custom exception class, without any message then make the custom exception class empty.
* If you wish to display custom exception class, with message then define a parameterized constructor with a String parameter and call the super class constructor with the String parameter.

ex;

 public class InvalidAmountExcepton extends RuntimeException {

 public InvalidAmountException(String msg) {

 super(msg);

 }

}

throw keyword:

ex1:

 try {

 int x = 5 / 0; throw new ArithmeticException(“/ by zero”);

 }

 catch(ArithmeticException ex) {

 S.o.p(ex);

 }

ex2:

 try {

 if(age < 0) {

 throw new InvalidAgeException(“age < 0 is

 invalid”);

 }

}

catch(InvalidAgeException ex) {

 S.o.p(ex);

}

* from the try block, if any predefined exception occurrs, then internally JVM uses throw keyword to throw the exception object from the try block.
* if any custom exception occurrs, then the programmer has to explicitly define the throw statement, to thorw the exception object from the try block.

throws keyword:

 throws keyword is used to propagate the exception to the caller method.

 throws keyword is used in the method header/signature.

 If an exception occurs in a method then it can be handled in 2 places.

1. within that method only using try and catch blocks.
2. in the caller method.
* to send/transfer/propagate an exception to the caller method, we use throws keyword.

 void m1() throws Exception {

 //line1

 //line2

 //line3

 }

Q) write the difference between throw and throws?

A) write here



**package** com.ashokit.custom;

**public** **class** InvalidAmountException **extends** RuntimeException {

 **public** InvalidAmountException(String message) {

 **super**(message);

 }

}

**package** com.ashokit.custom;

**public** **class** InsufficientBalanceException **extends** RuntimeException {

 **public** InsufficientBalanceException(String message) {

 **super**(message);

 }

}

**package** com.ashokit.bank;

**import** com.ashokit.custom.InsufficientBalanceException;

**import** com.ashokit.custom.InvalidAmountException;

**public** **class** BankAccount {

 **private** **double** balance;

 **public** BankAccount(**double** balance) {

 **this**.balance = balance;

 }

 **public** **void** deposit(**double** amount) {

 **try** {

 **if**(amount <= 0) {

 **throw** **new** InvalidAmountException("The given amount <=0, it can't be deposited");

 }

 balance += amount;

 System.***out***.println("balance after deposit : "+balance);

 }

 **catch**(InvalidAmountException ex) {

 System.***out***.println(ex);

 }

 }

 **public** **void** withdraw(**double** amount) {

 **try** {

 **if** (amount <= 0) {

 **throw** **new** InvalidAmountException("The given amount <=0, it can't be withdrawn");

 }

 **if**(amount > balance) {

 **throw** **new** InsufficientBalanceException("The given amount > balance, so you can't withdraw");

 }

 balance -= amount;

 System.***out***.println("remaining balance after withdraw : " + balance);

 }

 **catch**(InvalidAmountException ex) {

 System.***out***.println(ex);

 }

 **catch**(InsufficientBalanceException ex) {

 System.***out***.println(ex);

 }

 }

}

**package** com.ashokit.main;

**import** com.ashokit.bank.BankAccount;

**public** **class** MainClass {

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

 BankAccount account = **new** BankAccount(9000.0);

 account.deposit(1000.0);

 account.withdraw(-3000);

 }

}