CSE2005 (Object Oriented Programming Systems)

Module 3: Exception Handling

Dr. Arundhati Das

Exception Handling in Java

"An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world."—Java complete reference book

"When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method."-Java complete reference book

Exception Handling in Java

Error:

An Error indicates serious problem that a reasonable application should not try to catch.

Exception:

Exception indicates conditions that a reasonable application might try to catch.

Dictionary Meaning: Exception is an abnormal condition.

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 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 the problem

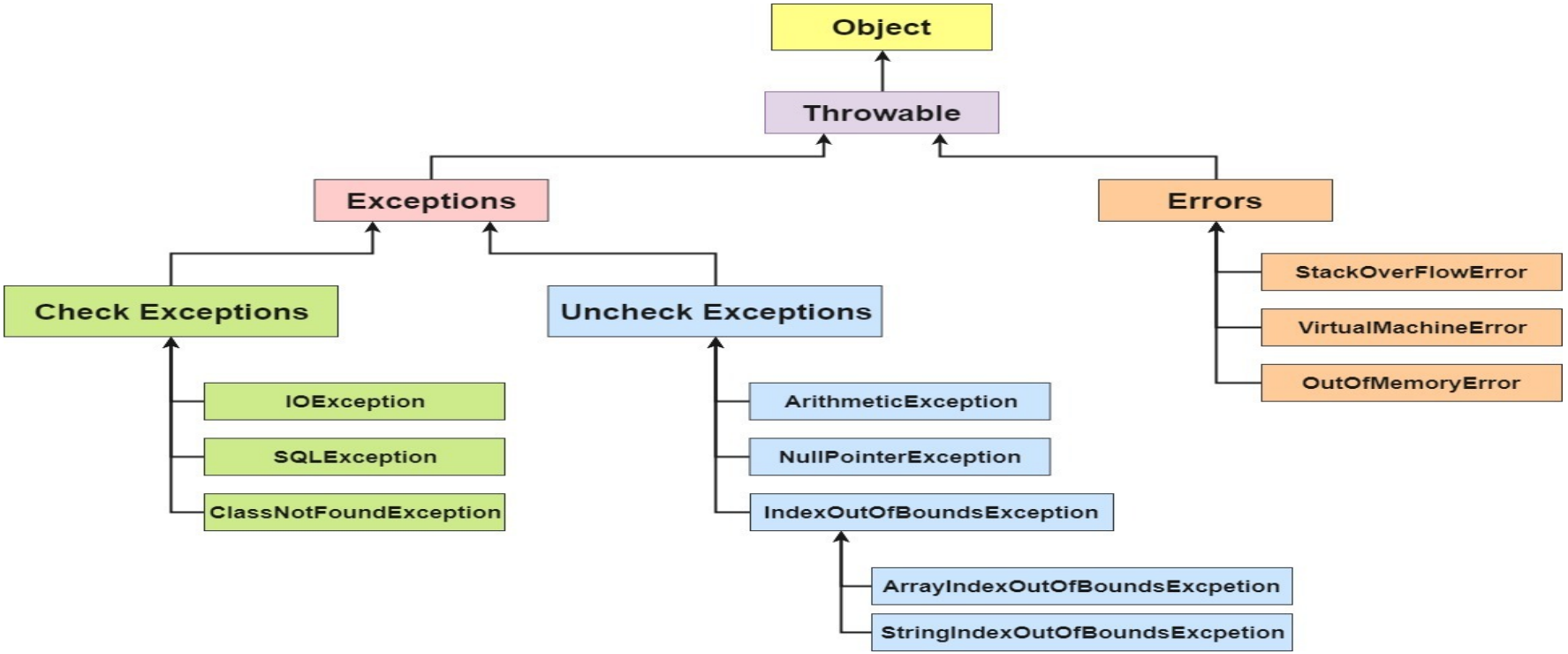
```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Why we go for exception Handling?

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.

Exception Hierarchy

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error.



Exception Handling in Java

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

1. A user has entered an invalid data.
2. A file that needs to be opened cannot be found.
3. A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception Handling in Java

If there is no exception handling code present during runtime for the thrown exception, then default exception handler will be provided by JRE.

If thrown Exception is Caught by JRE's Default Exception Handler then the Name of the Exception in string form will be displayed over System.out and the execution of the program will be stopped.

It will display print stack trace i.e, it will display program name, class name and corresponding method where exact that exception is raised and terminating the program abnormally or unsuccessfully.

Categories of Exceptions

We have three categories of Exceptions.

(According to Oracle, there are three types of exceptions)

1. Checked Exception
2. Unchecked Exception
3. Error

1. Checked Exception:

- A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions.

- These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

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

Checked Exception

For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error:
unreported exception
FileNotFoundException; must be caught
or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Unchecked Exception:

2 Unchecked Exception:

- An unchecked exception is an exception that occurs at the time of execution.
- These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API.
- Runtime exceptions are ignored at the time of compilation.
- The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Unchecked Exception

Unchecked Exception:

```
public class Unchecked_Demo {

   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)

Errors

Errors –

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

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Scenarios of Exception

1) A scenario where ArithmeticException occurs

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

```
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

Important blocks

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.

Important blocks

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 try-catch block

- If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.
- Java catch block is used to handle the Exception by declaring the type of exception.

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

```
try{
//code that may throw an exception
}finally{}
```

Exception Handling in Java

Problem without exception handling

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

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

    }

}
```

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

Solution with exception handling

Solution with exception handling

```
public class TryCatchExample2 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         //handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

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

```
// displaying the custom message
System.out.println("Can't divided by zero");
```

Output: java.lang.ArithmeticException: / by zero rest of the code

Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler.
- At a time only one exception occurs and at a time only one catch block is executed.

All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

If we generate **NullPointerException**, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class Exception will invoked.

The exception without maintaining the order of exceptions (i.e. from most specific to most general) will throw error.

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
        catch(ArithmeticException e)
           {
            System.out.println("Arithmetic Exception occurs");
           }
        catch(ArrayIndexOutOfBoundsException e)
           {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
           }
        catch(Exception e)
           {
            System.out.println("Parent Exception occurs");
           }
        System.out.println("rest of the code");
   }
}
```

Arithmetic Exception occurs
rest of the code

```
public class MultipleCatchBlock2 {

    public static void main(String[] args) {

        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

ArrayIndexOutOfBounds Exception occurs
rest of the code

```
public class MultipleCatchBlock3 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Arithmetic Exception occurs
rest of the code

```
public class MultipleCatchBlock4 {

   public static void main(String[] args) {

        try{
            String s=null;
            System.out.println(s.length());
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Parent Exception occurs
rest of the code

```
class MultipleCatchBlock5{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
 }
}
```

Compile-time error

Nested try block

- The try block within a try block is known as nested try block in java.

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

- In case of nested try blocks, context of that exception is pushed onto stack.
- Inner try block may/or may not have catch statements associated with it.
- If an exception is thrown from inner try block then first inner catch statements are matched (if present) . If no match is found then outer try block are matched. If there also no match found then default handler will be invoked.
- However, if outer try block throws the exception then only outer try blocks are matched.

```
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
```

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");
 }
}
```

```
going to divide
java.lang.ArithmeticException: /
by zero
java.lang.ArrayIndexOutOfBou
ndsException: 5
other statement
normal flow..
```

finally block

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

```
public class ExcepTest {

   public static void main(String args[]) {
      int a[] = new int[2];
      try {
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }finally {
         a[0] = 6;
         System.out.println("First element value: " + a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

Difference between final, finally and finalize

No.	final	finally	finalize
1)	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.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

EXAMPLES tested during class:

```
class A
{
void methodName()
{
try {
int arr[]= {10,20};
arr[2]=30/0;


}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("ArrayIndexOutOfBound
s Exception");
}
catch(ArithmeticException e)
{
System.out.println("Don't divide by 0");
System.out.println(e);
}
}
}
```

```
public class TestException2 {
public static void main(String[]
args) {
// TODO Auto-generated method stub
A a=new A();
a.methodName();


}
}
```

EXAMPLES:

```
public class TestException1 {

public static void main(String[]
args) {
// TODO Auto-generated method stub

try {


try {

int z=10/0;
}
catch(ArithmeticException e)
{
System.out.println("inner try1");
}
```

```
try
{
int arr[]= {10,20};
arr[2]=30;
}

finally
{
System.out.println("inner
finally");
}
}
catch(ArrayIndexOutOfBoundsExcepti
on e)
{
System.out.println("handled from
outer try");
}
finally {}


}
}
```

EXAMPLES:

```
public class TestException {
public static void main(String[]
args) {

try {
int x=0;
System.out.println("Hello1");
int arr[]={10,20,30};
System.out.println(arr[3]);
int a=20/x;
System.out.println(a);
}
catch(ArrayIndexOutOfBoundsExcepti
on e)
{
System.out.println("element not
available");
System.out.println(e);
}
catch(ArithmeticException e)
{
System.out.println("Don't divide
by 0");
System.out.println(e);
}
catch(Exception e)
{
System.out.println("caught inside
Exception parent class");
//System.out.println("caught
inside Exception parent class");
}
//try {int arr[]={10,20,30};
//System.out.println(arr[3]);}
//catch(ArrayIndexOutOfBoundsExcep
tion e)
//{
//System.out.println("element not
available");
//System.out.println(e);
//}
System.out.println("Hello");
}}
```

throw Clause [statement]

- **'throw' clause in Java is used to throw Exceptions**

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw**

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 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.

- **Syntax**

1. ***throw ThrowableInstance***

   Where ThrowableInstance must belong to an Object of Type Throwable or any of its sub class

2. ***throw new Exception-Name()***

   Where Exception-Name can be either a Exception or any of its sub-class

3. ***throw new Exception-Name(parameters)***

   In this form parameters can be supplied with exception [Assumption: The desired exception class must supplies a parameterized constructor]

‘throws’ clause in Java

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 declaration. A **throws** clause lists the types of exceptions that a method might throw.

- Specifically required if a method throws a Checked Type Exception
- Optional if a method throws an Un-Checked Type Exception

- Syntax

```
return-type     methodName(parameters)  throws     exception-list
{
        ….. Method Body
}
```

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

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try** / **catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
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: demo
```

```
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
     try{
     validate(13);
     }catch(Exception m){System.out.println("Exception occured: "+m);}

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

Output:Exception occured: InvalidAgeException:not valid

rest of the code...

Java's 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**. As previously explained, these exceptions need not be included in any method's **throws** list. In the language of Java, 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 10-2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java's Checked Exceptions Defined in **java.lang**

User defined exceptions: Writing Your Own Exceptions

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.

If you are creating your own Exception that is known as custom exception or user-defined exception.

Java user-defined exceptions are used to customize the exception according to user need.

By the help of user-defined exception, you can have your own exception and message.

User defined exceptions: Writing Your Own Exceptions

- **Programmer Can Write Either a Checked Exception OR an Unchecked Type Exception.**

- **To Create a Checked Type Exception**
  - **Make Your Exception class a direct subclass of Exception OR any one of its subclass Except RunTimeException.**

  class AException extends Exception { …}

  class BException extends IOException { ..}

- **To Create an Unchecked type Exception**
  - **Make Your Exception class a subclass of RuntimeException OR any one of its subclass .**

  class XException extends RuntimeException { … }

  class YException extends AritmeticException { … }

EXAMPLES tested during class:

```
class A
{
void methodName()
{
try {
int arr[]= {10,20};
arr[2]=30/0;


}
catch(ArrayIndexOutOfBoundsExcepti
on e)
{
System.out.println("ArrayIndexOutO
fBounds Exception");
}
catch(ArithmeticException e)
{
System.out.println("Don't divide
by 0");
System.out.println(e);
}}}
```

```
public class TestException2 {

public static void main(String[]
args) {
// TODO Auto-generated method stu
A a=new A();
a.methodName();


}
}
```

EXAMPLES tested during class:

```
public class TestException3 {

public static void main(String[]
args) {
// TODO Auto-generated method stub
try
{
int arr[]= {10,20};
arr[2]=30;
try {
//int z=10/0;
}
catch(ArithmeticException e)
{
System.out.println("Arithmetic
Exception");
}
}
catch(ArrayIndexOutOfBoundsExcepti
on e)
{
System.out.println("Array index
out of bound");
}}}
```

EXAMPLES tested during class:

```
public class TestException4 {
public static void main(String[] args) {
// TODO Auto-generated method stub
try{
//throw new ArithmeticException();
//throw new ArrayIndexOutOfBoundsException();
throw new NullPointerException();
}
catch(ArithmeticException e)
{
System.out.println("Using throw keyword to throw exception manually
and catching the thrown exception");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Using throw keyword to throw exception manually
and catching the thrown exception");
}
catch(NullPointerException e)
{
System.out.println("Using throw keyword to throw exception manually
and catching the thrown exception");
}}}
```

EXAMPLES tested during class:

```
public class DemoThrow {
 void Check(int marks) throws
ArithmeticException,
ArrayIndexOutOfBoundsException,
NullPointerException
 {
 try {
 if(marks<50)
 {
 throw new ArithmeticException();
 }
 else
 { System.out.println("marks"+marks);
 }
 }
catch(ArithmeticException e)
{
System.out.println("Exception has occured :
invalid input");
System.out.println(e);
throw e;
}
finally
 {

System.out.println("F
inally here: sure to
excute");
 }
//throw new
NullPointerException(
);
 }
public static void
main(String[] args) {
// TODO Auto-
generated method stub
DemoThrow dt=new
DemoThrow();
dt.Check(40);
}
}
```

EXAMPLES tested during class:

```
class MyException extends Throwable
{
String str;
MyException(String s)
{
str=s;
}
}
public class UserDefinedException2 {

public static void main(String[] args) {
// TODO Auto-generated method stub

throw new MyException("error message");

}

}
```

EXAMPLES tested during class:

```
class InsufficientFundsException1 extends RuntimeException
{
String msg;
InsufficientFundsException1(String s)
{
msg=s;
}
}
class Amount1
{
int balance=5000;
int Balance()
{
return balance;
}
void Withdraw(int amount) throws InsufficientFundsException1
{try{if(amount>balance)
{
throw new InsufficientFundsException1("Out of balance");
}else
{balance=balance-amount;
System.out.println("current balance "+balance);
}}
catch(InsufficientFundsException1 e)
{System.out.println("user defined exception is handled here");
}}}
public class UserDefinedException3 {

public static void main(String[] args) {
// TODO Auto-generated method stub
Amount1 a=new Amount1();
a.Withdraw(10000);
}}
```

User defined exceptions

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
```

User defined exceptions

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

```
Caught
GeeksGeeks
```

Assignment (Model question from Module 3 for CAT-2): A Professor has planned to host workshops for the Students. Students who are having even registration number are going to attend Machine Learning and those who have odd registration number are going to attend Blockchain Technology workshop. Take user inputs for the registration number. Generate a user defined exception with message “You are not allowed for the Workshop” when the student has any backlog. Otherwise print “Welcome to the Workshop” along with the workshop name (i.e., Machine Learning or Blockchain Technology). Print the exception in catch block.