Chapter-7 Java Exception Handling

Error and Exception:

- **Both are** subclass of the built-in class "**Throwable**"

Error:

- Errors are the conditions which cannot get recovered by any handling techniques (or code of the program) and occur due to the lack of the system resources. Errors can't be recovered by any means because they can't be created, thrown, caught or replied. It surely cause termination of the program abnormally.
- Errors are caused due to the catastrophic failure which usually can't be handled by your program.
- Errors belong to unchecked type, as compiler do not have any knowledge about its occurrence and error always occur in runtime.
- Some of the examples of errors are Out of memory error or a System crash error.

Exceptions

- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time that disrupts the normal flow of the program's instructions.
- Exceptions are the conditions that occur at runtime and may cause the termination of program but they are recoverable using try, catch and throw keywords
- Most of the times exceptions are caused due to the code of our program. But, exceptions can be handled by the program itself, as exceptions are recoverable.
- Exceptions are handled by using three keywords "try", "catch", "throw".
- Simply we can say that the mistakes occurred due to the improper code are called exceptions. For example, if a requested class is not found, or a requested method is not found.

Difference between Error and Exception:

Error	Exception
1. Errors are the conditions which occur due to the lack of the system resources.	**1.** Exception is an unwanted or unexpected event, which occurs during the execution of a program and may cause the termination of program.
2.Errors cannot get recovered by any handling techniques (or code of the program)	**2.** The use of **try-catch** blocks can handle exceptions and recover the application from them.
3. Errors are classified as "unchecked" In java because they occur at run-time and are not known by the compiler.	**3.** Exceptions in java can be "checked" or "unchecked," meaning they may or may not be caught by the compiler.
4. They are defined in java.lang.Error package.	**4**. They are defined in java.lang.Exception package
5. Example: OutOfMemory" and "StackOverflow"	**5**. **5.1**. **Example of Checked Exceptions →** SQLException, IOException **5.2. Example of Unchecked** Exceptions→ArrayIndexOutOfBoundException, NullPointerException, ArithmeticException.

Exception Handling in Java

What is Exception in Java?

On Dictionary: Exception is an abnormal condition.

Programmatic approach: 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 in java?

It is a mechanism to handle runtime errors.

What type of runtime Errors?

Like, ClassNotFoundException,

IOException,

SQLException,

RemoteException, etc.

Why we use Exception Handling?

- ✓ An exception normally disrupts the normal flow of the application that is why we use exception handling.

Example: Suppose there are 5 statements in your program

```
Statement 1;
Statement 2;
Statement 3; //exception occurs
Statement 4;
Statement 5;
```

Suppose, exception is occurs at statement 3, then he rest of the code will not be executed i.e. statement 4 and 5 will not be executed. If exception handling is implemented, the rest of the statement will be executed.

What is the core Advantage of Exception Handling?

- ✓ To maintain the normal flow of the application**.**

Exception Hierarchy in java

- ❖ Root or Base class of Exception hierarchy → **Throwable (java.lang.Throwable)**
- ❖ Sub class **a**) Exception
  b) Error (both **a** and **b** are inherited)
- ❖ See following figure

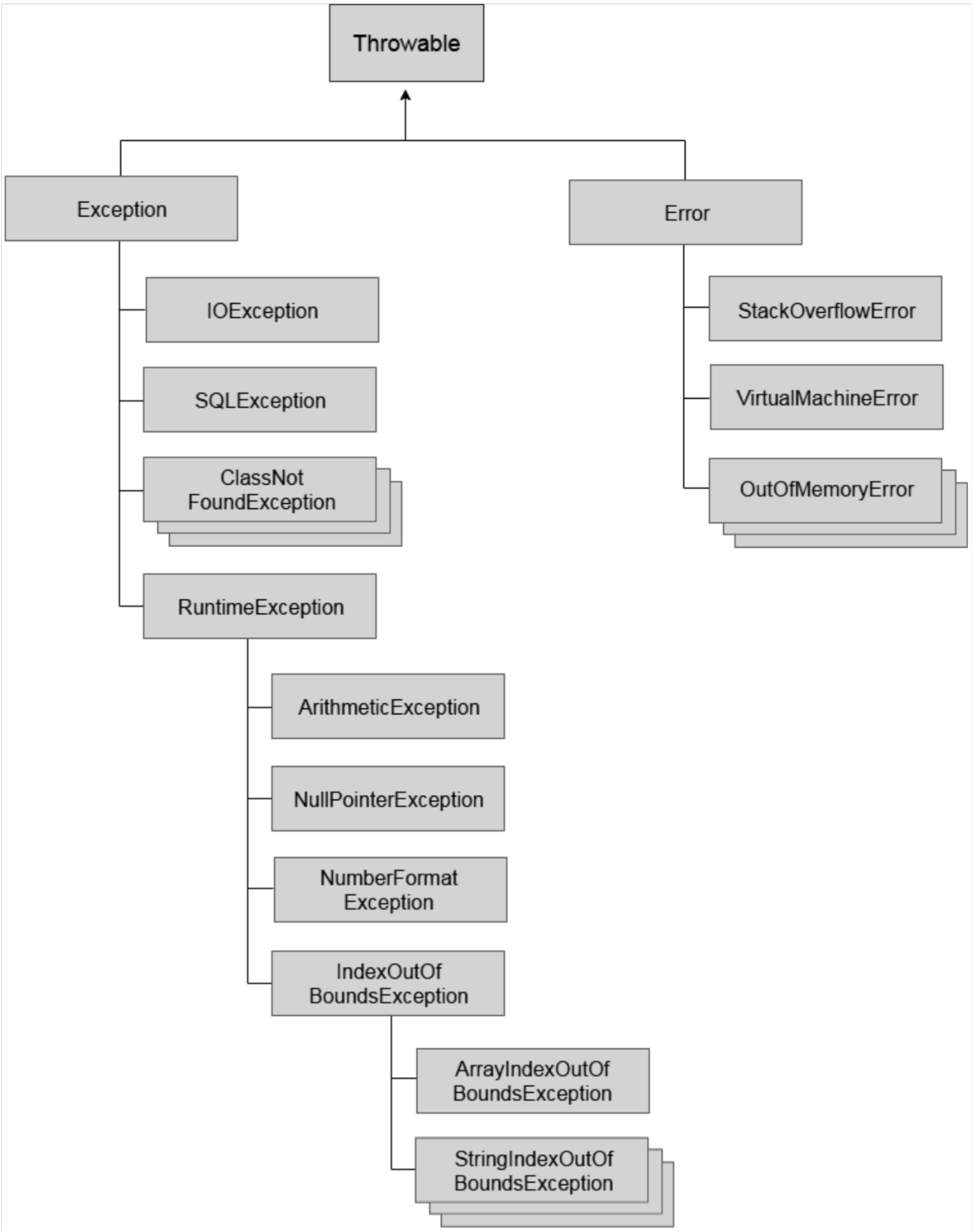


Figure: exception Hierarchy in java

Types of Java Exceptions

Generally,

- ❖ Exceptions in java are classified into **2 (Two)** types (Checked and Unchecked) and
- ❖ Error is considered as Unchecked Exception

But, **Oracle**, categorizes the java exception into **3 (Three)** different types including Error.

1) Checked Exception:

- ❖ Exceptions that are checked or occurs at compile time.
- ❖ Also called as **compile time exceptions**
- ❖ These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

OR

If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

For example:

- ✓ IOException
- ✓ SQLException
- ✓ DataAccessException
- ✓ ClassNotFoundException
- ✓ InvocationTargetException

Let us consider the following Java program

Program which opens file at location "C:\MukuJava\Chk_Exp_Eg.txt" and prints the first TEN lines of it.
Let's simplify the statement,
Location: ***MukuJava*** folder inside **C** drive of your PC, File name: *Chk_Exp_eg* and file type → *text*.

Therefore "*C:\MukuJava\Chk_Exp_Eg.txt*"

```
import java.io.*;

Class ChkExpEg
    {
      Public static void main (String [] args)
      {
        FileReader file = new FileReader("C:\\MukuJava\\Chk_Exp_eg.txt");

            BufferedReader fileInput = new BufferedReader(file);

        // Print first TEN (10) lines of file "C:\MukuJava\Chk_Exp_eg.txt"

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

          System.out.println(fileInput.readLine());

        fileInput.close();
      }
    }
```

- The above program doesn't compile, because the function **main ()** uses **FileReader()** and **FileReader()** throws a checked exception ***FileNotFoundException*.**
- Above program also uses **readLine()** and **close()** methods, and these methods also throw checked exception ***IOException***

<u>O / P</u>

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
code - unreported exception java.io.FileNotFoundException; must be caught or
declared to be thrown at ChkExpEg.main(ChkExpEg.java:5)
```

- ❖ By using throws, or try-catch block. This type of exception can be fixed.

2) **Unchecked Exception**:

- ❖ Exceptions that are not checked at compiled time.
- ❖ Also called as **Runtime Exceptions**

- ❖ Runtime exceptions are ignored at the time of compilation.
- ❖ An unchecked exception is an exception that occurs at the time of execution. For example bugs, logic error etc.
- ❖ Exceptions under *Error* and *RuntimeException* classes are unchecked exceptions in java.

<u>Example:</u>

- ✓ NullPointerException
- ✓ ArrayIndexOutOfBound
- ✓ IllegalArgumentException
- ✓ IllegalStateException etc.

<u>Let us consider the following Java program.</u>

```
Class UnChkExpEg {

Public static void main(String args[])
       {
          int a = 0;
          int b = 12;
          int c = b / a;
       }
   }
```

- ✓ Above program compiles successfully, but throws *ArithmeticException* when run. Because, *ArithmeticException* is an unchecked exception Therefore compiler allows it to compile.

<u>O / P:</u>

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
UnChkExpEg.main(UnChkExpEg.java:5)

Java Result: 1
```

Exception Handling

- ✓ Exception Handling is the Process of maintaining the normal flow of the application
- ✓ Exception handling prevents the abnormal termination of the program.

<u>Used five key words</u>

Try:

- ✓ The try block contain statements which may generate exceptions.
- ✓ The try block must be followed by either catch or finally.
- ✓ Try block can't be used alone.

Catch:

- ✓ The catch block defines the action to be taken, when an exception occur.
- ✓ Catch block must be preceded by try block
- ✓ Catch block can't be used alone. It can be followed by finally block later.

Throw:

- ✓ When an exception occur in try block, it is thrown to the catch block using throw keyword.

Throws:

- ✓ Throws keyword is used in situation, when we need a method to throw an exception.
- ✓ 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.

Finally:

- ✓ If exception occur or not, finally block will always execute.

- ✓ Finally block is used to execute the important code of the program

Let's make a story:

- ❖ Suppose we are in a smartphone manufacturing company

1. After the smartphone is made (**Production Department**) every company test the product

   If the product pass the test (**TEST DEPARTMENT**) then it is OK to ship. This is our Try block

2. But if test fails then it is resend to production line **(Manufacturing Department).** This is our throw

3. Production line is like our catch block who know how to fix that error

<u>But there can be multiple catch statements….. WHY?</u>

- ❖ Because there can be multiple types of error for example

Errors while producing smartphones

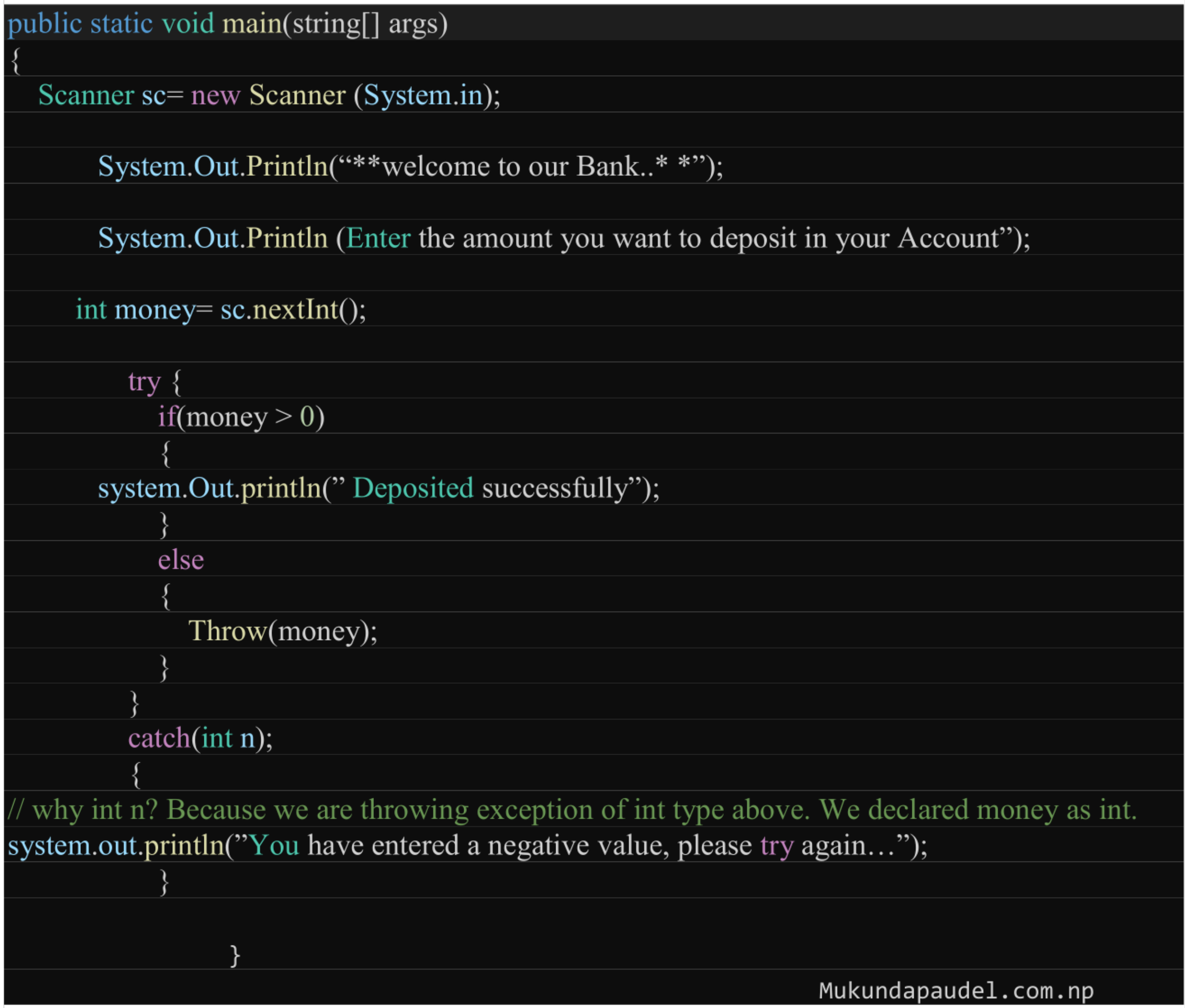
1. **D**isplay screen error
2. Some hardware fault
3. Some software fault etc.

So, there will be multiple catch statements depending on possible error

How to use try catch throw in program?

Example: 01

Let’s write a program for bank in which the user enters the amount to deposit in his / her account:



```
public static void main(string[] args)
{
   Scanner sc= new Scanner (System.in);

         System.Out.Println(“**welcome to our Bank..* *”);

         System.Out.Println (Enter the amount you want to deposit in your Account”);

       int money= sc.nextInt();

             try {
                if(money > 0)
                {
         system.Out.println(” Deposited successfully”);
                }
                else
                {
                   Throw(money);
                }
             }
             catch(int n);
             {
// why int n? Because we are throwing exception of int type above. We declared money as int.
system.out.println(”You have entered a negative value, please try again…”);
                }


                       }
```

Mukundapaudel.com.np

Example: 02 Java Exception Handling where we using a try-catch statement to handle the exception.

```java
public class JavaExceptionExample
        {
            public static void main(String args[])
            {
                try
                {
                    //code that may raise exception
                    int Result = 100 / 0;
                }
                catch (ArithmeticException e)
                        {
                            System.out.println(e);
                        }
                //rest code of the program

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

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
Rest of the code...
```

Key Points

Java exception handling is involves following keywords

- ✓ **try**,
- ✓ **catch**,
- ✓ **throw**,
- ✓ **throws**, and
- ✓ **Finally**.

- ➢ Program statements that raise exceptions are placed within a **try** block.
- ➢ If an exception occurs within the try block, it is thrown.

- By using **catch** block thrown exception will catches and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a try block completes is put in a **finally** block.

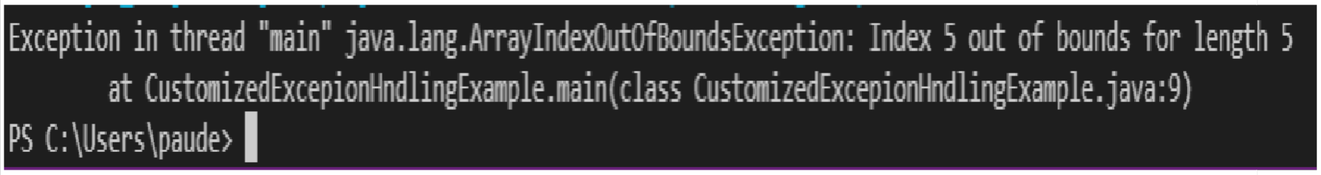
Example: 03

Let's consider the java program by defining an array of size 5. Suppose you are trying to access the elements at index 5 then it will throws an exception (because you can access elements only from index 0 to 4) and program terminates.

Therefore, the statement inside System.out.println(); will never execute. To execute it, we must handle the exception using try-catch method.

```
class ItsMyExcepionHndlingExample {

    public static void main (String[] args) {

        // array of size 5.
        int[] arr = new int[5];

        // this statement causes an exception
        int i = arr[5];

        // the following statement will never execute
   System.out.println("This is the example of User defined Exception Handling");
    }
}
```

Output:



```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
        at CustomizedExcepionHndlingExample.main(class CustomizedExcepionHndlingExample.java:9)
PS C:\Users\paude>
```

How to use try-catch clause to handle such exception?

```
try {
    // block of code to monitor for errors
    // the code you think can raise an exception
    }
    catch (ExceptionType1 exObj1) {
    // exception handler for ExceptionType1
    }
    catch (ExceptionType2 exObj2) {
    // exception handler for ExceptionType2
    }
        finally {
    // block of code to be executed after try block ends or write the important
code here which must execute in any cost. And finally block is optional.

    }

```

Example: 04

Consider a java program to demonstrate the use of try, catch and finally block in exception handling

```
public class ExampleEec
        {

            public static void main(String[] args)
            {
                try
                {
                    System.out.println(7 / 0);

System.out.println("Hello iam Try block. i have code that may rise exception..");
                }
                catch (ArithmeticException e)
                {
      System.out.println("I am the catch block and I am the solution of
Exception: " + e.getMessage());
                }
                catch (NullPointerException e)
                {
                    System.out.println("Exception:" + e.getMessage());
                }

finally
```

```
                {

System.out.println("hey ! I am finally block and je bhaye ni ma chalchhu..");
                }

            }

        }
    }
```

Output

$javac ExampleEec.java
$java -Xmx128M -Xms16M ExampleEec
Exception: / by zero
hey! I am finally block and Je bhaye ni ma chalchhu..

Example-05: Use of try, catch and finally in java exception Handling

```
class ExampleofTCF {
    Public static void main(String args[]) {
      //try block
      try{
             System.out.println("This is Try block");
             int num=55/0;
             System.out.println(num);
       }
       //catch block
       catch(ArithmeticException e){
           System.out.println("This is Catch block");
          System.out.println("ArithmeticException --> Number divided by zero");
       }
       //finally block
       finally{
           System.out.println("This is Finally block");
       }
       System.out.println("Rest of the code continues...");
    }
 }
```

Output

```
This is Try block
This is Catch block
ArithmeticException --> Number divided by zero
This is Finally block
Rest of the code continues…
```

Throwable

- The **Throwable** class provides a string variable that can be set by the subclasses to provide a detail message that provides more information of the exception occurred.
- All classes of **Throwables** define a one-parameter constructor that takes a string as detail message.
- The class **Throwable** provides getMessage() functions to retrieve an exception.

Throw vs Throws

THROW	THROWS
1. Throw keyword is used to throw an exception explicitly. throw new ArithmeticException("Arithmetic Exception");	1. **Throws** keyword is used to declare an exception with the method name. It works like the try-catch block because the caller needs to handle the exception thrown by **throws.** throws ArithmeticException;
2. **Throw** is followed by an instance of Exception class	2.**Throws** is followed by exception class names
3.**Throw** keyword always use the object of exception class	3. **Throws** keyword uses the exception class.
4**.Throw** keyword is used within the body of a method	4. **Throws** is used in method signature to declare the exceptions.

```
void usingThrowKeyword()

{

   try

   {

 //throwing arithmetic exception using throw

throw new ArithmeticException("You can’t divide a number by zero.");

   }

   catch (ArithmeticException e)

   {

      // handling exception

   }

}
```

```
//arithmetic exception using throws

void usingThrowsKeyword() throws ArithmeticException

{

   //Statements

}
```

5. By using the **throw** keyword only one exception can be declare at a time.
 	
void usingThrowKeyword()
 	
{
 	
//By using of throw keyword single exception is throwing
 	
throw new ArithmeticException("You can't divide a number by zero.");
 	
}
	5. Multiple exceptions can be handle by declaring them using the throws keyword. The throws keyword uses the comma (,) to separate the exceptions.
	
	void myMethod() throws ArithmeticException, NullPointerException
	
	{
	
	// handling exception
	
	}

6.**Example:**
 	
public class ThrowExample {
 	
public static void main(String[] args)
 	
{
 	
// Use of unchecked Exception
 	
try {
 	
// double x=7 / 0;
 	
throw new ArithmeticException();
 	
}
 	
catch (ArithmeticException e)
 	
{
	6. **Example:**
	
	import java.io.IOException;
	
	public class UseOfThrowAndThrows {
	
	public static void main(String[] args) throws IOException
	{
	}
	}
	
	Output:
	
	Exception in thread "main" java.io.IOException
	
	atUseOfThrowAndThrows.main(UseOfThrow.java:7)

```
        e.printStackTrace();
}
}
}
```
 Output: java.lang.ArithmeticException: / by zero atUseOfThrow.main(UseOfThrow.java:8)	

User-defined / Custom Exception in Java

- ❖ Java provides the facility to create your own exception class and throws that exception using **'throw'** keyword is called User defined Custom Exception in java.
- ❖ This can be implemented by extending the class Exception.

Simply we can say that,

Custom exceptions in Java are those exceptions which are created by a programmer to meet their specific requirements of the application.

For example:

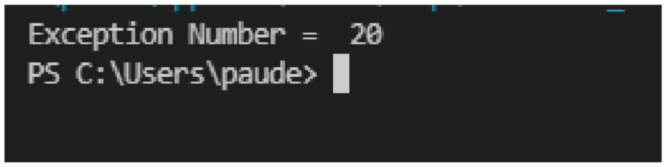
A banking application, a customer whose age is lower than 18 years, the program throws a custom exception indicating "needs to open joint account".

suppose a Voting Application in Nepal, If a person's age entered is less than 18 years, the program throws "Not eligible for Voting" as a custom exception in context of Nepal.

See the Following Example:

```
class UserDefinedExpEg
{
   public static void main(String args[])
    {
  try
  {
      throw new ItsMyException(20);
      // throw is used to create a new exception and throw it.
  }
 catch(ItsMyException e)
 {
   System.out.println(e) ;
 }
}
}
// User-defined exception must extend Exception class
 class ItsMyException extends Exception
 {
   int x;

   ItsMyException(int y)
    {
     x=y;
    }
   public String toString()
    {
     return ("Exception Number =  "+x) ;
  }
}
```

Output:



```
Exception Number =  20
PS C:\Users\paude>
```

Tips on Java Exception Handling

1. Never swallow the exception in catch block

```
catch (NoSuchMethodException e)
 {
  return null;
}
```

- ❖ Without handling with meaningful description or re-throwing the exception, returning null it totally swallows the exception, losing the cause of error forever.
- ❖ If you don’t know the reason of failure, how you would prevent it in future?

2. Declare the specific checked exceptions that your method can throw

```
public void MyMethod() throws Exception //Incorrect way
{

}
```

- ❖ Declare the specific checked exceptions that your method can throw. If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message.
- ❖ Refactor code if possible.

```
public void MyMethod() throws SpecificException1, SpecificException2  //Correct way
{

}
```

3. Do not catch the Exception class rather catch specific sub classes

```
try {
  MyMethod();
} catch (Exception e) {
  LOGGER.error("method has failed", e);
}
```

- ❖ If the method you are calling later adds a new checked exception to its method signature, the developer’s intent is that you should handle the specific new exception.

- ❖ If your code just catches Exception (or Throwable), you'll never know about the change and the fact that your code is now wrong and might break at any point of time in runtime.

4. Never catch Throwable class

- ❖ Java errors are also subclasses of the Throwable.
- ❖ Errors are irreversible conditions that cannot be handled by JVM itself and for some JVM implementations, JVM might not actually even invoke your catch clause on an Error.

5. Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost

```
//Wrong and this destroys the stack trace of the original exception
catch (NoSuchMethodException e) {
  throw new MyServiceException("Some information: " + e.getMessage());  //Incorrect way
}
//Do This:
catch (NoSuchMethodException e) {
  throw new MyServiceException("Some information: " , e);  //Correct way
}
```

6. Either log the exception or throw it but never do the both

```
catch (NoSuchMethodException e) {
  LOGGER.error("Some information", e);
  throw e;
}
```

- ❖ In this example, logging and throwing will result in multiple log messages in log files, for a single problem in the code, and makes life hell for the engineer who is trying to dig through the logs.

7. Never throw any exception from finally block

```
try {
  MyMethod();  //Throws exception One
} finally {
  cleanUp();    //If finally also threw any exception the exceptionOne will be lo
st forever
}
```

- ❖ In this example, if MyMethod() throws an exception, and in the finally block also, cleanUp() throws an exception, that second exception will come out of method and the original first exception (correct reason) will be lost forever.
- ❖ If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it come out of the finally block.

8. Always catch only those exceptions that you can actually handle

- ❖ Most important concept.
- ❖ Don't catch any exception just for the sake of catching it.
- ❖ Catch any exception only if you want to handle it or, you want to provide additional contextual information in that exception.

```
catch (NoSuchMethodException e) {
  throw e; //Avoid this as it doesn't help anything
}
```

9. Use finally blocks instead of catch blocks if you are not going to handle exception

- ❖ If inside your method you are accessing Mymethod 2, and method 2 throw some exception which you do not want to handle in method 1, but still want some cleanup in case exception occur, then do this cleanup in finally block.
- ❖ Do not use catch block.

```
try {
  MyMethod();  //Method 2
} finally {
  cleanUp();    //do cleanup here
}
```

10. Remember "Throw early catch late" principle

- ❖ Most famous principle about Exception handling.
- ❖ It basically says that you should throw an exception as soon as you can, and catch it late as much as possible. You should wait until you have all the information to handle it properly.

11. Always clean up after handling the exception

- ❖ If you are using resources like database connections or network connections, make sure you clean them up.
- ❖ If the API you are invoking uses only unchecked exceptions, you should still clean up resources after use, with try – finally blocks.
- ❖ Inside try block access the resource and inside finally close the resource. Even if any exception occur in accessing the resource, then also resource will be closed gracefully.

12. Throw only relevant exception from a method

- ❖ Relevancy is important to keep application clean.
- ❖ A method which tries to read a file; if throws NullPointerException then it will not give any relevant information to user.
- ❖ Instead it will be better if such exception is wrapped inside custom exception e.g. NoSuchFileFoundException then it will be more useful for users of that method.

13. Validate user input to catch adverse conditions very early in request processing

- ❖ Always validate user input in very early stage, even before it reached to actual controller.
- ❖ It will help you to minimize the exception handling code in your core application logic.
- ❖ It also helps you in making application consistent if there is some error in user input.

<u>For example:</u> consider a user registration application and you are following below logic.

```
// Incorrect approach. Leaves database in inconsistent state
1) Validate User
2) Insert User
3) Validate address
4) Insert address
5) If problem the Rollback everything
```

```
// correct, validate everything first-->take user data in DAO layer-->update DB.
1) Validate User
2) Validate address
3) Insert User
4) Insert address
5) If problem the Rollback everything
```

14. Always include all information about an exception in single log message

```
// Dont do This
LOGGER.debug("Using cache sector A");
LOGGER.debug("Using retry sector B");

//Do this
LOGGER.debug("Using cache sector A, using retry sector B");
```

- ❖ Using a multi-line log message with multiple calls to LOGGER.debug() may look fine in test case, but when it shows up in the log file of an app server with thousands of threads running in parallel, all dumping information to the same log file, your two log messages may end up spaced out 1000 lines apart in the log file, even though they occur on subsequent lines in your code.

15. Pass all relevant information to exceptions to make them informative as much as possible

- ❖ It is important to make exception messages and stack traces useful and informative.
- ❖ What is the use of a log, if you are not able to determine anything out of it?
- ❖ These type of logs just exist in your code for decoration purpose.

16. Always terminate the thread which it is interrupted

```
while (true) {
  try {
    Thread.Stop(100000);
  } catch (InterruptedException e) //Don't do this
  {

  }

  TakeAction();
}
```

- ❖ InterruptedException is a clue to code, that it should stop whatever it's doing.
- ❖ Some common use cases for a thread getting interrupted are the active transaction timing out, or a thread pool getting shut down.
- ❖ Instead of ignoring the InterruptedException, your code should do its best to finish up what it's doing, and finish the current thread of execution.

```
while (true) {
  try {
    Thread.Stop(100000);
  } catch (InterruptedException e) {
    break;
  }
}
TakeAction();
```

17. Use template methods for repeated try-catch

- ❖ There is no use of having a similar catch block in 100's places in your code.
- ❖ It increases code duplicity which does not help anything.
- ❖ Use template methods for such cases.

18. Document all exceptions in the application with javadoc

- ❖ Make it a practice to **javadoc** all exceptions which a piece of code may throw at runtime.
- ❖ Also try to include possible course of action, user should follow in case these exception occur.

Debugging Techniques in Java

- process of determining and fixing bugs or errors present in the code, project, or application is known as Debugging
- Debugging of code written in Java is a tough task.
- Debugging helps to improve the quality of the code.
- Debugging also helps to understand the flow of program code.
- It is a must-have skill for every Java programmer.

There are various Technique available for debugging of Java code but some of them are IDE Specific. Following are the most common techniques for java code debugging.

1. **Use conditional breakpoint**
    - It's a breakpoint with a specified condition where the thread will stop at the targeted line when the specified condition is true, unlike a line breakpoint.
2. **Use exception breakpoints**
    - For the debugging of NullPointerException, exception breakpoints is solution.
3. **Watchpoint**
    - The *watchpoint* is a breakpoint set up on a field or variable.
    - Each time the targeted field or variable is accessed or changed, the execution of the program will get stop and then you can debug.
4. **Step filtering**
    - While performing Step Into during debugging process, sometimes it happens that the control of the program goes from one class to other class and eventually, you are moved to the external libraries or JDK classes like *System* or *String*.
    - In case you do not want to move to the JDK classes or external libraries, then step filtering is used.

- ❖ Step filtering helps to filter out the JDK classes from *Step Into*.
- ❖ This feature will assist in skipping some particular packages during the debugging process.

5. **Evaluate (inspect and watch)**

- ❖ Evaluate enables to check the value of expressions while debugging Java programs.
- ❖ All you need to do is right-click the statement and click on inspect. It will show you the value of the selected expression during the debugging process.
- ❖ The value will appear in front of you over the watch window.

6. **Drop to frame**

- ❖ It allows you to re-run a part of your program.

7. **Environment variables**

8. **Show logical structure**

- ❖ The logical structure option is very useful, especially when trying to determine the contents of Java collection classes such as HashMap or ArrayList.
- ❖ Instead of displaying the detailed information, the logical structure will present only the necessary content such as the key and value of a HashMap.

9. **Modify the value of a variable**

- ❖ There is no need to restart your debugging session with minor changes in the code. You can continue to debug the program code. It will save time.

10. **Stop in Main**

- ❖ When a program is debugged with this feature enabled, the execution will stop at the first line of the main function.

Stream, Zip File Stream, Object Stream and Handling File

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen.

Java uses the concept of **stream** to make I/O operation fast with the packages java.io

Stream

- ❖ A stream can be defined as a sequence of data and it is composed of bytes.
- ❖ A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- ❖ The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.
- ❖ All streams represent an input source and output destination.
- ❖ The stream in the java.io package supports many data such as primitives, object, localized characters, etc.
- ❖ A stream does not store data and, in that sense, it is not a data structure.
- ❖ Stream also never modifies the underlying data source.

There are Three (3) types of standard Streams

1. **InPutStream**
   - ✓ This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**. (i.e. used to read data from a source)
2. **OutPutStream**
   - ✓ This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**. (i.e. used for writing data to a destination)
3. **ErrorStream**
   - ✓ This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

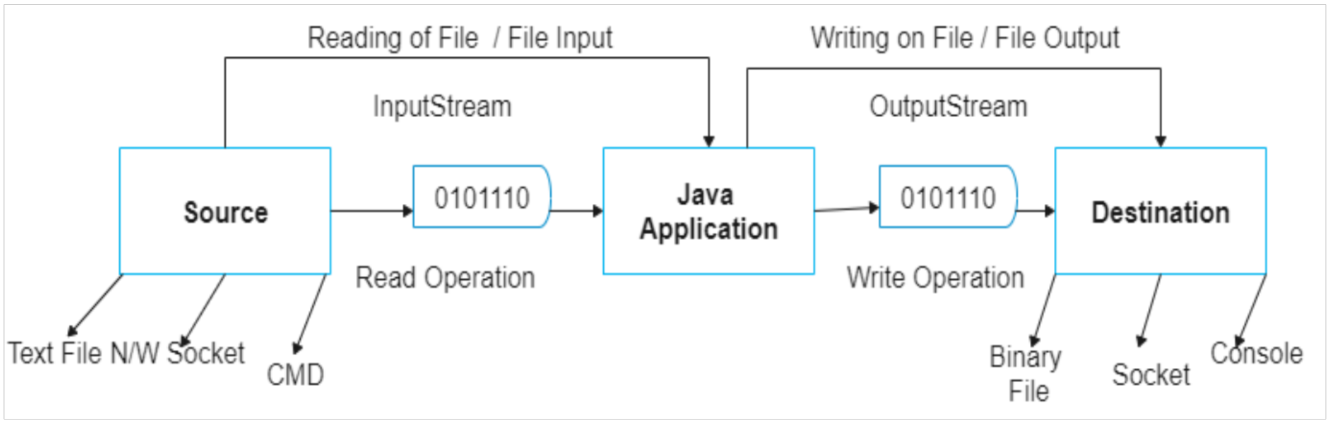


Figure: Stream in Java I/O

Methods of I/O Stream

Output Stream classes

```
Void write(int) throws IOException
// This method is used to write byte (single character) to the current output
in every call of these method you have to handle exception i.e. write these method into try block
Void Write(byte[])throws IOException
//This method is used to write the array of bytes
Void flush()throws IOException
//This method flushes the current stream
Void close()throws IOException
//This method is used to close the current stream
```

Input Stream Classes

```
int read() throws IOException
//This method is used to read next byte
int availabe () throws IOException
// This method returns the number of byte available
void close() throws IOException
// This method is used to close the current input stream
```

Reading and Writing Files

For reading and writing in file there are two most important stream are available: *FileInputStream* and *FileOutputStream*

FileInputStream

- FileInputStream is used to read data from the files.
- Objects can be created using the keyword **new** and there are several types of constructors available.
- Following constructor takes a file name as a string to create an input stream object to read the file

```
InputStream f = new FileInputStream("C:/PT_Java/Hello");
```

- Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows

```
File f = new File("C:/PT_java/hello");
InputStream f = new FileInputStream(f);
```

- Once *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

FileOutputStream

- FileOutputStream is used to create a file and write data into it.
- The stream would create a file, if it doesn't already exist, before opening it for output.
- There are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("C:/PT_java/Hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows

```
File f = new File("C:/PT_java/Hello");
OutputStream f = new FileOutputStream(f);
```

❖ Once *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Example To demonstrate *InputStream* and *OutputStream*

Following example creates file "*Myfile.txt*" and writes given numbers in binary format. Same would be the output on the standard output screen

```
/*Example of inputStream and OutputStream*/
import java.io.*;
public class fileStreamExample {

   public static void main(String args[]) {

      try {
         byte bWrite [] = {10,15,20,25,30};
         OutputStream opstrm = new FileOutputStream("Myfile.txt");

      for(int x = 0; x < bWrite.length ; x++) {
            opstrm.write( bWrite[x] );   // writes the bytes
         }
         opstrm.close();

         InputStream ipstrm = new FileInputStream("Myfile.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.println((char)is.read() + "  ");
         }
         ipstrm.close();
      } catch (IOException e) {
         System.out.print("There is Exception….Alert !");
      }
   }
}
```

Writing and Reading objects in Java

❖ By using the serialization process objects can be read and write in java file.
❖ Serialization is a process to convert objects into a writable byte stream.
❖ Once converted into a byte-stream, these objects can be written to a file. The reverse process of this is called de-serialization.

A Java object is serializable if its class or any of its superclasses implement either the ***java.io.Serializable*** interface or its subinterface, ***java.io.Externalizable***.

The objects can be converted into byte-stream using ***java.io.ObjectOutputStream.*** In order to enable writing of objects into a file using ***ObjectOutputStream.***

On reading objects, the ***ObjectInputStream*** directly tries to map all the attributes into the class into which we try to cast the read object. If it is unable to map the respective object exactly then it throws a ***ClassNotFound*** exception.

Reading objects in Java are similar to writing object ***using ObjectOutputStream*** *and* ***ObjectInputStream***.

Consider the example of writing objects and reading objects in Java.

Java Object.

```
      //Student.java

import java.io.Serializable;

public class Student implements Serializable {

   private String name;
   private int age;
   private String gender;

   Student(String name, int age, String gender) {
      this.name = name;
      this.age = age;
      this.gender = gender;
   }
   public String toString() {
      return "Name of the Student:" + name + "\nAge: " + age + "\nGender: " + gender;
   }
}
```

Let’s serialize the class student then.

```
                /*WriteAndReadObjectEg.java*/
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class WriteAndReadObjectEg {

   public static void main(String[] args) {

      student std1 = new student("Sita", 20, "Female");
      student std2 = new student("Ram", 22, "Male");

      try {
         FileOutputStream fopstrm = new FileOutputStream(new File("myObjects.txt"));
         ObjectOutputStream objopstrm = new ObjectOutputStream(fopstrm);

         // Write objects to file
         objopstrm.writeObject(std1);
         objopstrm.writeObject(std2);

         objopstrm.close();
         fopstrm.close();

         FileInputStream fipstrm = new FileInputStream(new File("myObjects.txt"));
         ObjectInputStream oipstrm = new ObjectInputStream(fipstrm);

         // Read objects
         student s1 = (student) oipstrm.readObject();
         student s2 = (student) oipstrm.readObject();

         System.out.println(s1.toString());
         System.out.println(s2.toString());

         oipstrm.close();
         fipstrm.close();

      } catch (fileNotFoundException e) {
         System.out.println("File not found");
```

```
        } catch (IOException e) {
            System.out.println("Error initializing stream");
        } catch (ClassNotFoundException e) {
            system.out.println("class not found exception occurs");
            e.printStackTrace();
        }

    }

}
```

Output

```
Name of the Student:Sita
Age: 20
Gender: Female
Name of the Student:Ram
Age: 22
Gender: Male
```

User-defined / Custom Exception in Java

- ❖ Java provides the facility to create your own exception class and throws that exception using **'throw'** keyword is called User defined Custom Exception in java.
- ❖ This can be implemented by extending the class Exception.

Simply we can say that,

Custom exceptions in Java are those exceptions which are created by a programmer to meet their specific requirements of the application.

For example:

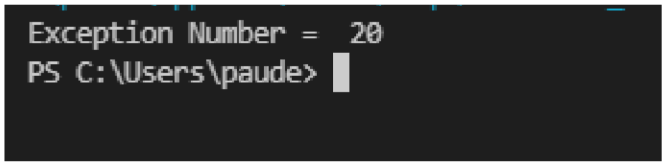
A banking application, a customer whose age is lower than 18 years, the program throws a custom exception indicating "needs to open joint account".

suppose a Voting Application in Nepal, If a person's age entered is less than 18 years, the program throws "Not eligible for Voting" as a custom exception in context of Nepal.

See the Following Example:

```
class UserDefinedExpEg
{
   public static void main(String args[])
    {
  try
  {
      throw new ItsMyException(20);
      // throw is used to create a new exception and throw it.
  }
 catch(ItsMyException e)
 {
   System.out.println(e) ;
 }
}
}
// User-defined exception must extend Exception class
 class ItsMyException extends Exception
 {
   int x;

   ItsMyException(int y)
    {
     x=y;
    }
   public String toString()
    {
     return ("Exception Number =  "+x) ;
   }
}
```

Output:



```
Exception Number =  20
PS C:\Users\paude>
```