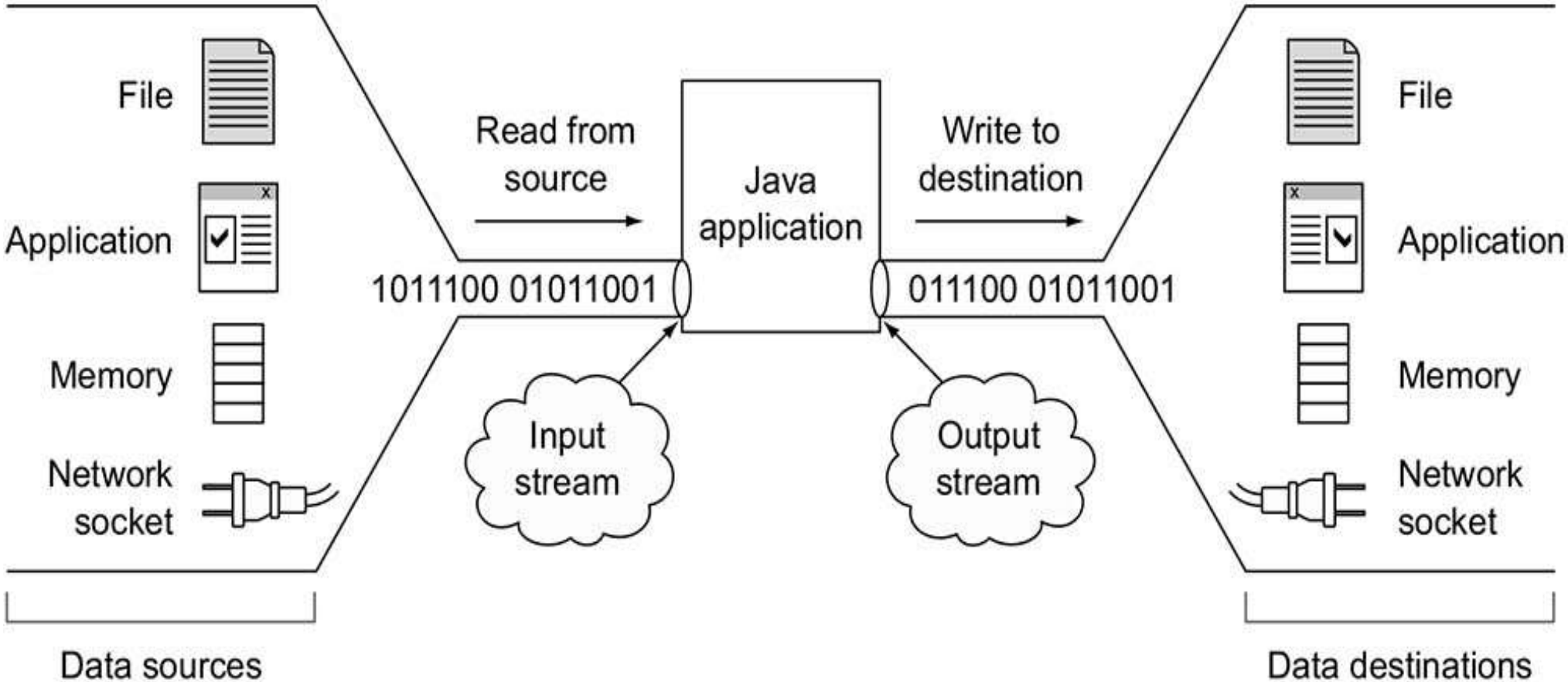
Unit-III

IO Streams

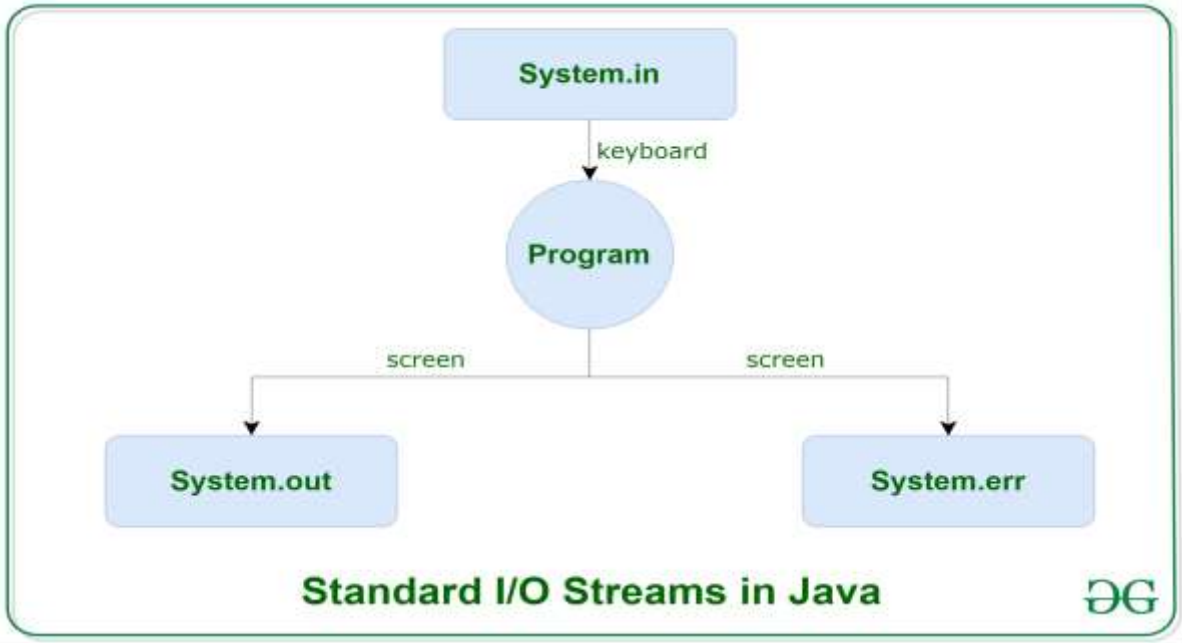
Stream?

- A stream is a sequence of data.
- In Java, Stream is a channel or a path along which data flows between source and destination.
- Java brings various Streams with its **I/O package** that helps the user to perform all the input-output operations.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.


File
Application
Memory
Network socket
Data sources
Read from source
1011100 01011001
Java application
Input stream
Write to destination
011100 01011001
Output stream
File
Application
Memory
Network socket
Data destinations

Standard(Default) IO Streams in Java

- There are three standard IO Streams in Java
  - ❖System.in
  - ❖System.out
  - ❖System.err



Types of Streams

- Depending on the **type of operations**, streams can be divided into two primary classes.
  - InputStream
  - OutputStream

InputStream

- These streams are used to read data that must be taken as an input from a source array or file or any peripheral device.

- The Java **InputStream** class is the base class (superclass) of all input streams in the Java IO API.

- Each subclass of InputStream typically has a very specific use, but can be used as an InputStream.

Example:

- FileInputStream,
- BufferedInputStream,
- ByteArrayInputStream etc.

OutputStream

- These streams are used to write data as outputs into an array or file or any output peripheral device.

- The Java **OutputStream** class is the base class (superclass) of all output streams in the Java IO API.

- Each subclass of OutputStream typically has a very specific use, but can be used as an OutputStream

Example:

- FileOutputStream,
- BufferedOutputStream,
- ByteArrayOutputStream etc.

Types of Streams(Based on file types)

- Based on file types Streams can be divided into two primary classes

  - ByteStream
  - CharacterStream

ByteStreams:

- Programs use byte streams to perform input and output of (8-bit) bytes.
- All byte stream classes are descended from **InputStream** and **OutputStream**.

CharacterInputStream:

- Java **Character** streams are used to perform input and output for 16-bit Unicode.

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

File Handling

- File handling in Java implies reading data from the file and writing data to a file.
- The File class from the **java.io** package, allows us to work with different formats of files.
- In order to use the File class, you need to create an object of the class and specify the filename or directory name.

Ex:

```
// Import the File class
import java.io.File;
// Specify the filename
File obj = new File(“sample.txt");
```

File and Directory

- A file is a named location that can be used to store related information.
- Ex: main.java is a Java file that contains information about the Java program.
- A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

File handling methods

Method	Type	Description
canRead()	Boolean	It tests whether the file is readable or not
canWrite()	Boolean	It tests whether the file is writable or not
createNewFile()	Boolean	This method creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	It tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

Example: Creating a File

```
import java.io.IOException;
public class FileDemo
{
   public static void main(String[] args)
   {
      File f=new File("sample.txt");
      if(f.exists())
      {
         System.out.println("File already existed..");
      }
      else
      {
         try
         {
            f.createNewFile();
            System.out.println("File created successfully..");
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Example: No of Files and Directories in a Directory

```
import java.io.File;
public class Ex2
{
    public static void main(String[] args)
    {
        File f1=new File(".");
        String[] list=f1.list();
        int count=0;
        for (String name:list)
        {
            count++;
            File f2=new File(name);
            if(f2.isFile())
            {
               System.out.println("File      :"+name);
            }
            else if (f2.isDirectory())
            {
                System.out.println("Directory:"+name);
            }

        }
        System.out.println("No of files and dir :"+count);
    }
}
```

Example: File Information

```
import java.io.File;
import java.util.Scanner;
public class Ex3
{
    public static void main(String[] args) {
        System.out.println("Enter your file name");
        Scanner scr=new Scanner(System.in);
        String fname=scr.next();
        File f=new File(fname);
        if(f.exists())
        {
            System.out.println("Size of the file :"+f.length());
            System.out.println("Absolute path    :"+f.getAbsolutePath());

            if(f.canRead())
                System.out.println(f.getName()+":  Is readable file");
            else
                System.out.println(f.getName()+":  Is not readable file");

            if(f.canWrite())
                System.out.println(f.getName()+":  Is writeable file");
            else
                System.out.println(f.getName()+":  Is not writeable file");

            if(f.canExecute())
                System.out.println(fname+":  Is Executable file");
            else
                System.out.println(fname+":  Is not Executable file");
        }
        else
        {
            System.out.println("No such file existed..");
        }
    }
}
```

Example: Deleting Specified File

```
import java.io.File;
import java.util.Scanner;
public class Ex4
{
    public static void main(String[] args)
    {
        System.out.println("Enter your file name");
        Scanner scr=new Scanner(System.in);
        String fname=scr.next();
        File f=new File(fname);
        if(f.exists())
        {
            if(f.delete())
            {
                System.out.println(f.getName()+" is deleted successfully..");
            }
            else
            {
                System.out.println(f.getName()+" is not deleted..");
            }
        }
        else
        {
            System.out.println("No such file existed..");
        }
    }
}
```

Exception Handling

What is an Exception?

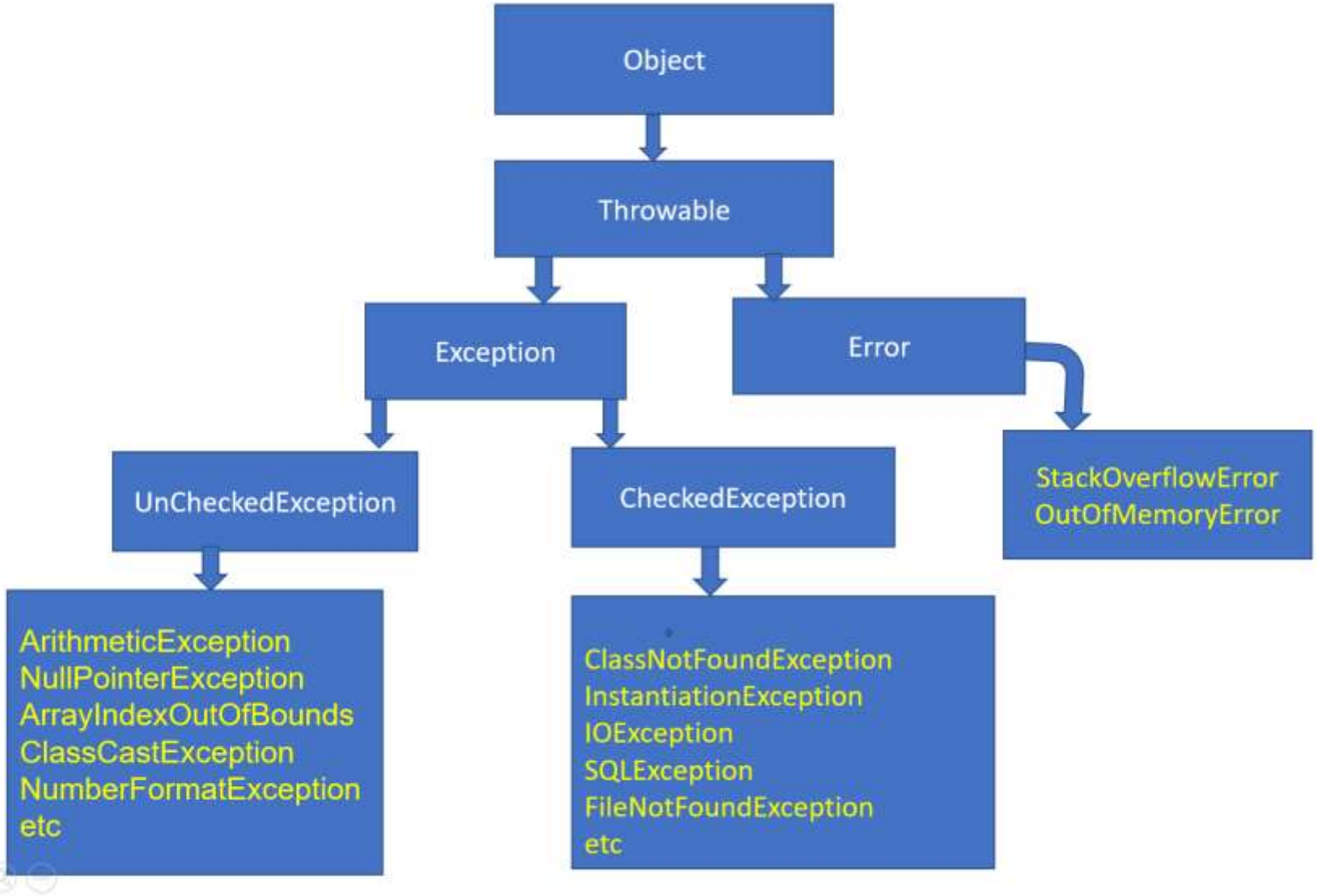
- It is an unexpected unwanted event which disturbs entire execution flow of the program.

Ex:

- SleepingException
- TirePuncharedException
- PowerCutException
- ArithmaticException

- Exception also called as **runtime error**.

Exceptions Hierarchy



- All exceptions and errors are sub classes of class **Throwable.**
- **Throwable** class having two child classes.
  - **Error**
  - **Exceptions**

- Both **Error and Exceptions** are present in **java.lang** package.

Error

- **Error** is an event caused by lack of System resources.
- These are exceptional conditions that are external to the application, and that the application usually cannot anticipate .
- Which cannot get recovered by any handling techniques.
- It surely cause termination of the program abnormally.
- Errors belong to **unchecked** type and mostly occur at runtime.

Ex:

- OutOfMemoryError
- StackOverFlowError

Exception:

- An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- These are recoverable by using programing techniques.

- Exceptions are divided into two types
  - UnCheckedExceptions (RuntimeExceptions)
  - CheckedExceptions (CompileTimeExceptions)

UnChecked Exceptions

- Also called as Runtime Exceptions
- Compiler does not check this type of Exceptions
- This type of programs are not connected to External resources(like files, printers, scanner).
- Exception handling is **optional** i.e, If we handle the exceptions program terminates normally, If not it leads to abnormal termination.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.(out of range)
InputMismatchException	If we are giving input is not matched for storing input.
ClassCastException	If the conversion is Invalid.
IllegalArgumentException	Illegal argument used to invoke a method.
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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.

Examples for UnChecked Exceptions:

i) ArrayIndexOutOfBoundException:

```
int a[]={10,20,30}
System.out.println(a[3]); // Array index out of bound exception
```

ii) NumberFormatException:

```
String s="ten"
int i=Integer.parseInt(s);
System.out.println(i); // Number Format Exception
```

iii) Arithmetic Exception:

```
System.out.println(100/0);
```

iv)NullPointerException:

```
String s=null;
System.out.println(s); // Null Pointer Exception
```

v) IllegalArgumentException

```
public class Ex1
{
    public static void main(String[] args)
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("Hello");
            try {
                Thread.sleep(-10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}
```

Checked Exceptions

- The Exceptions which are checked by the compiler at compilation time for the proper execution of the program at runtime is called Checked Exceptions.

- Also called compile time exceptions.

- External resources (like files , printers , scanners) may connected to the programs.

- Exception handling is mandatory for this type of Exceptions . If not handled even **.class** file wont generate.

Exception	Description
ClassNotFoundException	If the loaded class is not available
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	If the requested method is not available.

Exception handling?

- Exception handling means it is not repairing an exception we are providing alternative way to continue rest of the program normally.

- The program must be graceful termination.

- There are two types of Exception handling techniques.
  - Default Exception Handling
  - Customized Exception Handling

Default Exception Handling

- When ever an exception raised in the method in which it is raised is responsible for the preparation of exception object by including the following information.
  - Name of Exception.
  - Description.
  - Location of Exception.

- After preparation of Exception Object, The method handovers the object to the JVM.

- JVM will check for Exception handling code in that method

- If the method doesn’t contain any exception handling code then JVM terminates that method abnormally and removes corresponding entry from the stack.

- This process will be continued until **main()** method.

- If the main method also doesn’t contain any exception handling code then JVM terminates main method abnormally.

- Just before terminating the program JVM handovers the responsibilities of exception handling to default exception handler.

- Default exception handler prints the error in the following format.
  - Name of Exception
  - Description stackTrace

Ex:

```
class Test
{
    public static void main(String[] args)
    {
        doStuff();
    }
    public static void doStuff()
    {
        doMoreStuff();
    }
    public static void doMoreStuff()
    {
        System.out.println(10/0);
    }
}
```

O/P:-

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo.doMoreStuff(ExceptionDemo.java:30)
        at ExceptionDemo.doStuff(ExceptionDemo.java:25)
        at ExceptionDemo.main(ExceptionDemo.java:21)
```

Exception Handling:

- Providing alternative way to execute rest of the program normally.
- Exception Handling is normal Execution of the program or graceful termination of the program at runtime.
- Exception class present in **java.lang** package.
- We can handle the exceptions in two ways.
  - By using try-catch blocks
  - By using throws keyword.

In Java we have 5 key words to handle the Exceptions:-

- try
- catch
- finally
- throw
- throws

Exception handling by using try-catch block:

- In Exception Handling **try** block contains **risky code** of the program and **catch** block contains handling code of the program.
- **Catch** block code is a alternative code for Exceptional code. If the exception is raised the alternative code is executed fine then rest of the code is executed normally.

Syntax:

```
try
{
    Risky code;
}
Catch(ExceptionName reference_variable)
{
    Alternative code if Exception raised;
}
```

Without try-catch

```
class ExceptionDemo
{
    public static void main(String[] args) {
        System.out.println("statement-1");
        System.out.println("statement-2");
        System.out.println("statement-3");
        System.out.println(10/0);
        System.out.println("statement-4");
        System.out.println("statement-5");
        }
}
```

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
statement-1
statement-2
statement-3
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ExceptionDemo.main(ExceptionDemo.java:7)
```

With try-catch

```
class ExceptionDemo
{
    public static void main(String[] args) {
        System.out.println("statement-1");
        System.out.println("statement-2");
        System.out.println("statement-3");
        try {
            System.out.println(10/0);
        }
        catch (ArithmeticException e)
        {
           System.out.println("Divide / Zero exception");
        }
        System.out.println("statement-4");
        System.out.println("statement-5");
        }
}
```

output:

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
statement-1
statement-2
statement-3
Divide / Zero exception
statement-4
statement-5
```

Multiple catch() blocks:

- A try block can be followed by one or more catch blocks.
- Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions.
- 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.

Syntax:

```
try{
}
catch(Exception1 e)
{
}
catch(Exception2 e)
{
}
.
.
etc
```

finally:

- It is never recommended to write clean up code in try block. Because try block may execute or may not execute.

- And never recommended to use catch block for clean up code , because if there is no exception catch block wont execute.

- The finally keyword is used in association with a try/catch block and guarantees that a section of code will be executed, even if an exception is thrown.

- The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

- **Important:** The finally block is a key tool for preventing resource leaks

Possible combinations (try-catch()-finally())

- Try-catch() ➔ Allowed
- Try-finally() ➔ Allowed
- Try-catch-finally() ➔ Allowed
- Try-finally()-catch() ➔ Not allowed
- Catch()-finally() ➔ Not allowed

throw:

- The main purpose of the **throw** keyword is to creation of **Exception object explicitly** either for predefined or user defined .

- Throw keyword works like a try block. The difference is try block is automatically find the situation and creates a Exception object implicitly. Whereas throw keyword creates a Exception object explicitly.

- **throw** keyword must call within the method.

- By using throw keyword we can throw **only one Exception object** at a time.

Example:

```
if (withdrawal>balance)
        {
            throw new InsufficientFunds("No funds..");
        }
```

Creating user defined Exceptions

- In Exception Handling user can defined their own Exceptions.

- To create user defined Exceptions we need to create a child class to **RuntimeException** class or **Exception** class.

- Each and every Exception contains two constructors
  - default constructor
  - parameterized constructor

- **Naming convention**: Every user defined exception name must be suffix of **Exception**

Ex:

InsufficientFundsException

Example:

```
public static void checkBalance(double withdrawal)
  {
     double balance=1234567.50;
     if (withdrawal>balance)
     {
        throw new InsufficientFunds("No funds..");
     }
     else
     {
       balance= balance-withdrawal;
       System.out.println("You're A/C balance : "+balance);
     }
  }
}
class InsufficientFunds extends RuntimeException
{
  InsufficientFunds(String str)
  {
    super(str);
  }
}
```

throws:

- If any Checked Exception raised in a program that must be handle by **try-catch** or **throws** keyword.

Ex:

```
class ExceptionDemo
{
    public static void main(String[] args)
    {
        Thread.sleep(100);    // Checked Exception
    }
}
```

```
ExceptionDemo.java:5: error: unreported exception InterruptedException; must be caught or declared to be thrown
        Thread.sleep(100);
                     ^
1 error
```

Handling Exception with try-catch:

Ex:

```
public static void main(String[] args)
{
    try
    {
        Thread.sleep(100);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

Handling Exception with throws keyword:

Ex:

```
public static void main(String[] args) throws InterruptedException
{
    Thread.sleep(100);
}
```

- The main objective of the **throws** keyword is to delegate responsibilities to the caller method about Exception handling.
- **throws** keyword bypass the Exception but it doesn't prevent abnormal termination.

- By using throws keyword we can throw multiple Exceptions at a time.
- We can use throws keyword in Method declaration.
- **throws** keyword is applicable only Throwable objects but not Normal objects.
- throws keyword is required only for **checked exception** and usage of throws keyword for unchecked exception is meaningless.

Differences b/w throw and throws

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.