❖ Exception-handling fundamentals:

Exception 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 exception handling is managed via five keywords: **try, catch, throw, throws and finally.**

try statement: program statements that you want to monitor for exceptions are contained within try block.
Catch statement: If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System generated exceptions are automatically thrown by the Java run-time system.

throw statement: To manually throw an exception, use the keyword **throw.**

throws statement: Any exception that is thrown out of a method must be specified as such by a throws clause.

finally statement: any code that absolutely must be executed before a method returns is put in a finally block.

This is general form of an exception-handling block;

```
try
{
        //block of code to monitors for errors
}
catch(ExceptionType1 exob)
{
        //exception handler for ExceptionType1
}
catch(ExceptionType2 exob)
{
        //exception handler for ExceptionType2
}
finally
{
        //block of code to be executed before try block ends.
}
```

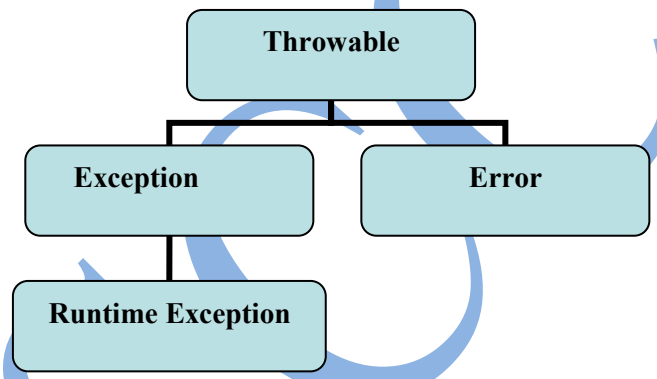
❖ Exception Types:

All exception types are subclasses of the built-in class **Throwable. Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception.**

Exception: this is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception** is called **RuntimeException.**

The other branch is topped by **Error,** which defines exceptions that are not expected to be caught under normal circumstances by your program. **Exception** of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

So the hierarchical structure of Exception may be generating as per the following



❖ Uncaught Exceptions

This small program includes an expression that intentionally causes a **divide-by-zero** error.

```
Class exc1
{
        public static void main(String args[])
        {
                int d=0;
                int  a=42/d;
        }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of exc1 to stop, because once an exception has been thrown, it must be caught by an exception handler and

dealt with immediately. In this example, we haven't supplied any exception handler of our own, so the exception is caught by the default handler provided by the Java run-time system.

The default handler displays a string describing the exception, prints a **stack trace** from the point at which the exception occurred and terminates the program.

Here is the output generated when this example is executed:

Java.lang.ArithmeticException: / by zero.
at exc1.main (Exc1.java:4)

here

1. **exc1=class name**
2. **main=method name**
3. **exc1.java=file name**
4. **4:= line number**
5. **ArithmeticException= types of exception thrown is a subclass of Exception.**

The stack trace will always show the sequence of method invocations that led up to the error.

```
Class exc1
Static void subroutine()
{
        int d=0;
        int a=10/d;
}
public static void main(String args[])
{
    exc1.subroutine();
}
```

the resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
Java.lang.ArithmeticException: / by zero
At exc1.subroutine(exc1.java:4)
At exc1.main(exc1.java:7)
```

- **Using try and catch statement:**

The benefits of the default exception handler provided by Java run time systems are as follow.

1. **It allows you to fix the error.**
2. **It prevents the program from stopped terminating.**

To guard against and handle run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the try block, include a **catch** clause that specifies the exception type that you wish to **catch.** The following program includes a try block and a catch clause which process the **ArithmeticException** generated by the division-by-zero error;

```
Class exc2
{
        public static void main(String args[])
        {
                int d, a;
                try
                {
                        d=0;
                        a=42/d;
                        System.out.println("This will not be executed");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Division by zero:");
                }
                System.out.println("After catch statement");
        }
}
```

This program generates the following output:

Division by zero.
After catch statement.

Notice that the call to **println()** inside the try is never executed. Once an exception is thrown, program control transfer out of the **try block** into the **catch block.**

Once an exception is created the **catch block** will be executed and control never **"returns"** to the try block from a **catch.** Therefore the line **"This will not be executed"** is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

➢ Displaying a Description of an Exception

Throwable, overrides the **toString()** method so, that it returns a string containing a description of the exception. You can display this description in a **println()** statement by simply passing the exception as an argument.

```
Catch(ArithmeticException e)
{
        System.out.println("Exception:" +e);
        A=0;            //set a to zero and continue
}
```

❖ Multiple catch clauses

In some cases, more than one exception could be raised by single piece of code. To handle this type of situation, you can specify two or more **catch clause,** each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.

After one catch statement executes, the others are bypassed, and execution continues after the **try/catch** block.

The **following example** traps two different exception types:

```
Class multicatch
{
        public static void main(String args[])
        {
                try
                {
                        int a=args.length;
                        System.out.println("a=" +a);
                        int b=42/a;
                        int c[]={10};
                        c[5]=100;
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Divide by 0:" +e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array index oob:" +e);
                }
                System.out.println("After try/catch blocks:");
        }
}
```

When you use **multiple catch** statements, it is important to remember that exception subclasses must come before any of their subclasses. This is because a catch statement that uses a **superclass** will catch exception of that type plus any of its subclasses. Thus a subclass would never be reached if it came after its **superclass.**

For example,

```
/* This program contains an error.

A subclass must come before its superclass in a series of catch statement. If not, unreachable
code will be created and a compile-time error will result.
*/

class supercatch
{
        public static void main(String args[])
        {
                try
                {
                        int a=0;
                        int b=42/a;

                }
                catch(Exception e)
                {
                        System.out.println("Generic Exception catch…");
                }
/*
        This catch is never reached because ArithmeticException is a subclass of Exception.
*/
                catch(ArithmeticException e)    //error-unreachable code
                {
                        System.out.println("This is never reached.");
                }
        }
}
```

- **Nested try statements**

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler of a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.

This continues until one of the catch statements succeeds or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run time system will handle the exception.

For example

```
Class nesttry
{
        public static void main(String arg[])
        {
                try
                {
                        int a=arg.length;
                        System.out.println("A=" +a);
```

```
                try
                {
                        if(a==1)
                        a=a/(a-a);
                        if(a==2)
                        {
                                int c[]={1};
                                c[3]=99;
                        }
                }
                catch(ArrayIndexOutOfBoundsException e)
                {System.out.println("Array-Index-Out…");}
                catch(ArithmeticException e)
                {System.out.println("Divide By Zero:" +e);}
        }
        catch(ArithmeticException e)
        {System.out.println("Divide By Zero:" +e);}
    }
}
```

- **throw statement**

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows.

```
throw new ThrowableInstance;
```

here ThrowableInstance must be caught an object of type **Throwable** or a subclass of **Throwable.** Simple types such as **int** or **char, String** and **Object** cannot be used as exceptions.

There are two ways you can obtain a **Throwable** object:

1. By using a parameter into **catch** clause
2. Creating one with the **new** operator.

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 the 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.

For example,

Define an exception called "myexception" that is thrown when a string is not equal to "Hello". Write a java program that uses this exception.

```
class myexception extends Exception
{
        myexception(String message)
        {super(message);}
}
class testmyexception
{
        public static void main(String args[])
        {
                try
                {
                        String s1="Hello";
                        String s2="HELLO";
                        if (s1.equals(s2))
                        System.out.println("String is equal");
                        else
                        {throw new myexception("String is not Equal");}
                }
                catch(myexception e)
                {
                        System.out.println("Caught my exception");
                        System.out.println(e.getMessage());
                }
                finally
                {System.out.println("BYE-BYE");}
        }
}
```

➢ throws statement

Sometimes in program exception may be generate inside the method. So we must specify some mechanism to protect the caller method against exception. You do so by declaring the keyword **throws** in java. A **throws** clause lists the types of exceptions that a method might a throw. This is necessary for all exceptions, except those of type **Error or RuntimeException** or any of their subclasses.

All the exceptions that a method can throw must be declared in the **throws** clause. If it does not, then the compile-time error will result.

The general form of a method declaration that includes a throws clause:

```
Type method-name(parameter-list) throws exception list
{
        //body of method
}
```

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

Example 1:

```
class throw1
{
	void demoproc() throws ArrayIndexOutOfBoundsException
	{
		int a[]={99};
		a[5]=100;
	}
}
class throwdemo2
{
	public static void main(String arg[])
	{
		try
		{
			throw1 th1=new throw1();
			th1.demoproc();
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("Inside Main");
			System.out.println(e.getMessage());
		}
	}
}
```

- **Finally statement:**

Finally creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. The finally clause is **optional.** However, each try statement requires at least one **catch** or **finally** clause.

Any time a method is **return** to the caller from inside **try/catch** block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the resources that might have been allocated at the beginning of method with the intent of disposing of them before returning.

Example 1:
File: finallydemo.java

```
class finallydemo
{
	//through an exception out of the method.
	static void procA()
	{	try
		{
		System.out.println("Inside procA");
```

```
                throw new NumberFormatException("demo");
                }
            finally
            { System.out.println("procA Finally");}
    }
        static void procB()
        {
                try
                {
                System.out.println("Inside procB");
                return;
                }finally
            { System.out.println("procB Finally");}
        }
        static void procC()
        {
                try
                {
                System.out.println("Inside procC");
                }finally
               { System.out.println("procC Finally");}
        }
        public static void main(String arg[])
        {
                try
                {
                        procA();
                }catch(Exception e)
            { System.out.println("Exception Caught");}
                procB();
                procC();
        }
}
```

Here is the output generated by the preceding program.

```
Inside procA()
procA finally
Exception caught
Inside procB()
procB finally
Inside procC()
procC finally
```

➢ Java's Built-in Exception:

Java defines several exception classes inside the standard java package **java.lang.** the most general of these exceptions are subclasses of the standard type RuntimeException. But java.lang is implicitly imported into all java programs, most exceptions derived from RuntimeException are automatically available.

Checked Exception: Checked Exceptions are the exceptions that are checked at compile time. 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.**

Exception	Meaning
ClassNotFoundException	Class not found
IllegalAccessException	Access to a class is denied
InterruptedException	Attempt to create an object of an abstract class or interface.

Java's checked Exceptions defined in java.lang

Unchecked Exception: Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions

.

In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked. These are called **unchecked exception.**

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
NegativeArraySizeException	Array created with a negative size
NumberFormatException	Invalid conversion of a string to a numeric format.

Java's unchecked RuntimeException subclasses

==========

==========

❖ How to create your own Exception subclasses?

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. This is quite easy to do: just define a subclass of **Exception. Exception** is a subclass of **Throwable** class.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable.** Therefore, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are listed in table.

Method	Description
String getMessage()	Returns a description of the exception
String toString()	Returns string value

The methods defined by Throwable

Example 1:

```
class myexception extends Exception
{
	private int detail;
	myexception(int a)
	{detail=a;}
	public String toString()
	{
		return "MyException[" +detail +"]";
	}
}

class exception1
{
	void compute(int a) throws myexception
	{
		System.out.println("Called compute(" +a +")");
		if(a>10)
		throw new myexception(a);
		System.out.println("Normal Exit");
	}
}
class exceptiondemo
{
	public static void main(String arg[])
	{
		try
		{
			exception1 e1=new exception1();
			e1.compute(10);
			e1.compute(20);
		}
		catch(myexception e)
		{
			System.out.println("Caught=" +e);
		}
	}
}
```

output:

```
called comute(10)
Normal exit
Called compute(20)
Caught MyException[20]
```