Exception

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

Exception Types:

1. **Checked exception:** The Java compiler is able to check the exceptions at compile time, these are called checked exceptions. E.g. java.io.IOException .
2. **Unchecked exception:** These are also called RuntimeExceptions, Java compiler is not able to check these exceptions. E. g. Divide by zero exceptions and ArrayIndexOutOfBoundsException.

All exception types are subclasses of the built-in class **Throwable**. Below **Throwable** are two subclasses that partition exceptions into two distinct branches.

1. **Exception:**
   - This class is used for exceptional conditions that user programs should catch.
   - This is also the class that you will subclass to create your own custom exception types.
   - There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as *division by zero* and *invalid array indexing*.
2. **Error:**
   - which defines exceptions that are not expected to be caught under normal circumstances by your program.
   - Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions:

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

```
class Exc {
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 **Exc** 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 handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Dileep

- ❖ Any exception that is not caught by your program will ultimately be processed by the default handler. 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 by the standard Java JDK run-time interpreter:

```
java.lang.ArithmeticException: / by zero
at Exc.main(Exc.java:4)
```

Notice how the class name, **Exc**; the method name, **main**; the filename, **Exc.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of the exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

Exception handling Mechanisms:

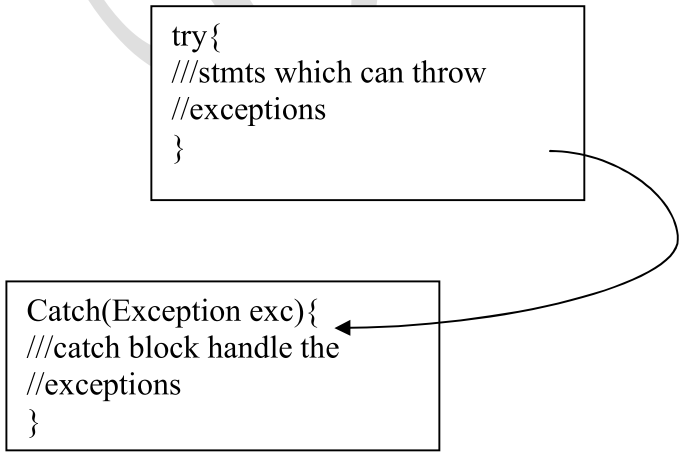
Java provides following Exception handling mechanisms:

1. try…..catch.
2. throws clause.
3. finally.
4. throw clause

try...catch:

- ❖ Program statements that you want to monitor for exceptions are contained within a **try** block.
- ❖ 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.
- ❖ To guard against and handle a 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.

Dileeps



```
try{
///stmts which can throw
//exceptions
}
```

```
Catch(Exception exc){
///catch block handle the
//exceptions
}
```

General form of Exception handling block:

```
try {
// block of code to monitor 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
}
```

To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zeroerror
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** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed.

Displaying a Description of an Exception:

- **Throwable** overrides the **toString( )** method (defined by **Object**) 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.

For example, the **catch** block in the preceding program can be rewritten like this:

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

When this version is substituted in the program, and the program is run under the standard Java JDK interpreter, each divide-by-zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

2. Multiple catch Clauses:

- ❖ While more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, 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:

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} 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.");
}}
```

This program will cause a division-by-zero exception if it is started with no command-line parameters, since **a** will equal zero. It will survive the division if you provide a commandline argument, setting **a** to something larger than zero.

But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.
```

3. Nested try Statements:

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

Dileep

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

4. throw cluase:

System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw** The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

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

- ❖ using a parameter into a **catch** clause,
- ❖ or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

throws cluase:

- ➢ A **throws** clause lists the types of exceptions that a method might throw.
- ➢ This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- ➢ All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is 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.
The corrected example is shown here:

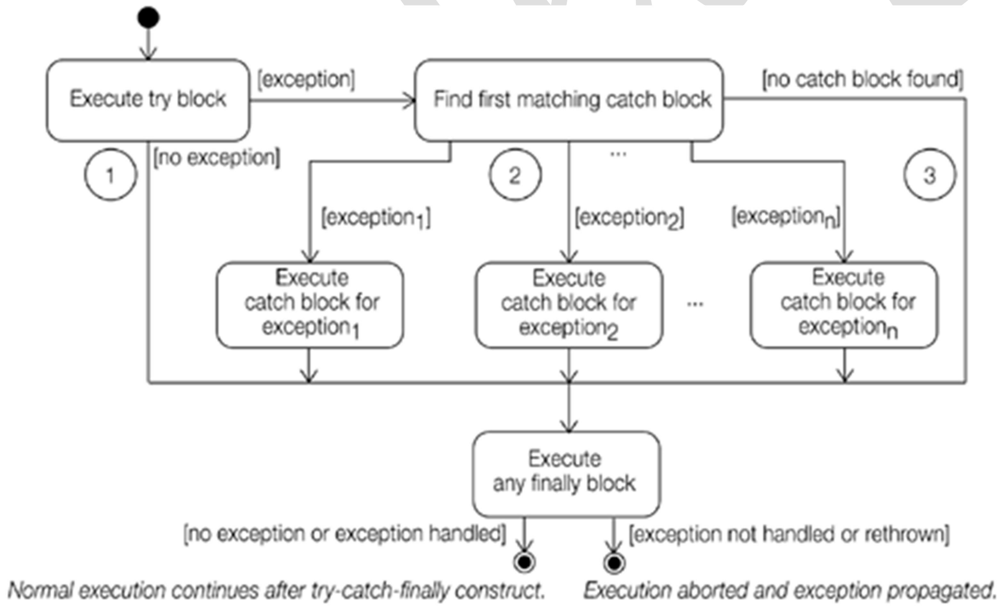
```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}}}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally:

- **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.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.



Execute try block
[exception]
Find first matching catch block
[no catch block found]
[no exception]
1
2
3
...
[exception1]
[exception2]
[exceptionn]
Execute catch block for exception1
Execute catch block for exception2
...
Execute catch block for exceptionn
Execute any finally block
[no exception or exception handled]
[exception not handled or rethrown]
Normal execution continues after try-catch-finally construct.
Execution aborted and exception propagated.

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

```
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {

System.out.println("procB's finally");
}}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}}
```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement.

The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Note If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.

Dileep

Exception Classes:

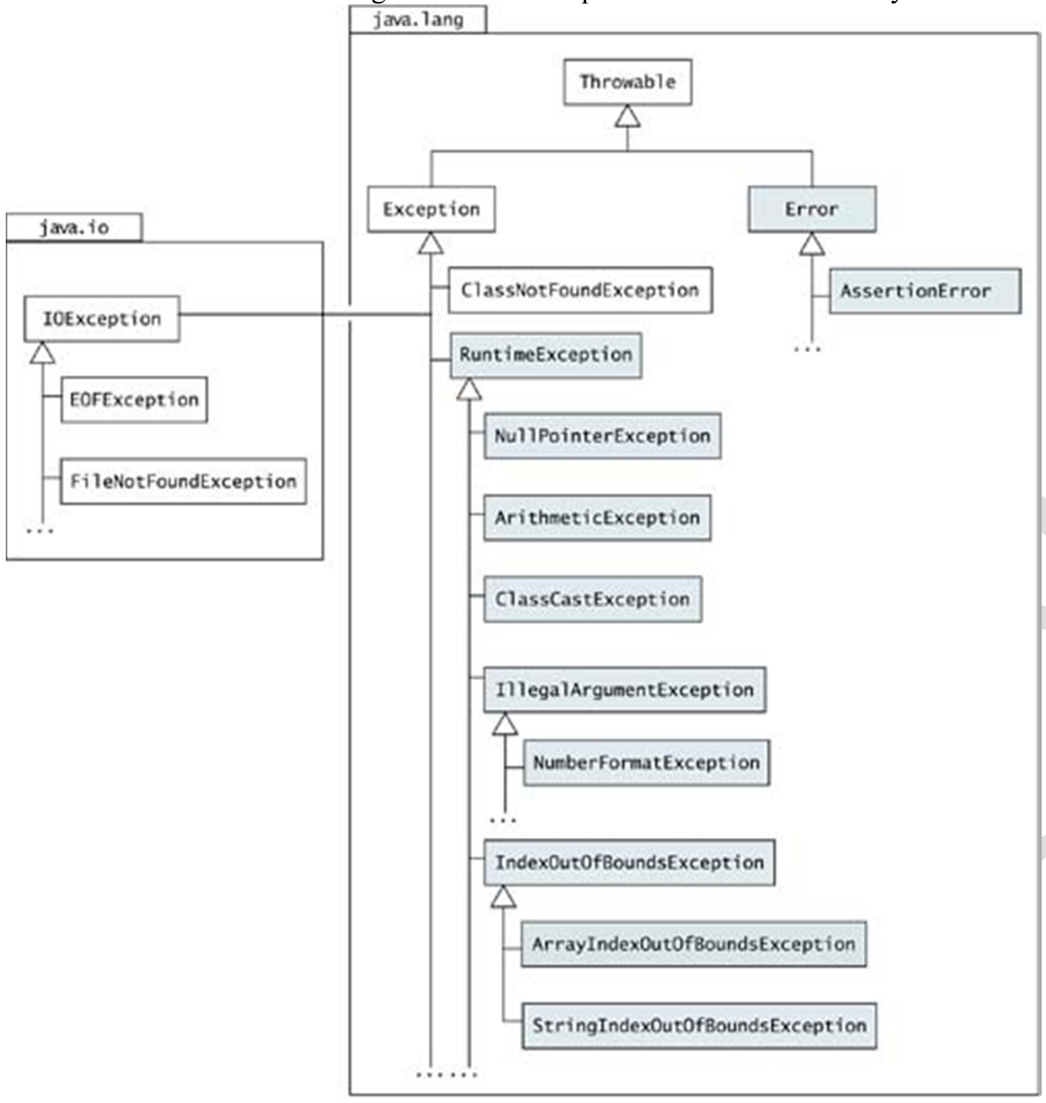
Exceptions in Java are objects. All exceptions are derived from the java.lang. Throwable class. Figure shows a partial hierarchy of classes derived from the Throwable class. The two main subclasses Exception and Error constitute the main categories of *throwables*, the term used to refer to both exceptions and errors. Figure also shows that not all exception classes are found in the same package.

The Throwable class provides a String variable that can be set by the subclasses to provide a *detail message*. The purpose of the detail message is to provide more information about the actual exception. All classes of throwables define a one-parameter constructor that takes a string as the detail message.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them You may also wish to override one or more of these methods in exception classes that you create. Some methods are shown below:

1. String getMessage ( ) ---Returns a description of the exception.
2. void printStackTrace( ) --- Displays the stack trace.
3. String toString ( ) --- Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

Figure: Partial Exception Inheritance Hierarchy



Classes that are shaded (and their subclasses) represent unchecked exceptions.

- Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.
- They 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.
- 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.*
- Java defines several other types of exceptions that relate to its various class libraries.

Creating own Exception subclasses:

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString( )** method, allowing the description of the exception to be displayed using **println( )**.

```
// This program creates a custom exception type.
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
class ExceptionDemo {
static void getAge(int a) throws MyException {
System.out.println("Called getAge(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
getAge(1);
getAge(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}}
}
```

Using Exceptions

1. Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
2. It is important to think of **try**, **throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in your program's logic.

Chained Exceptions

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.

For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an **ArithmeticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error.

To allow chained exceptions, Java 2, version 1.4 added two constructors and two methods to **Throwable**. The constructors are shown here.

1. **Throwable(Throwable *causeExc*):** --*causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred.

2. **Throwable(String *msg*, Throwable *causeExc*):---** allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

The chained exception methods added to **Throwable** are **getCause( )** and **initCause( )**. These methods are shown above.

- ✓ **Throwable getCause( )** :-- The **getCause( )** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. Added by Java 2, version 1.4.
- ✓ **Throwable initCause(Throwable *causeExc*):** --The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. Added by Java 2, version 1.4.

However, the cause exception can be set only once. Thus, you can call **initCause( )** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause( )**. In general, **initCause( )** is used to set a cause for legacy exception classes.

```
Example: // Demonstrate exception chaining.
class ChainExcDemo {
static void demoproc() {
// create an exception
NullPointerException e =
new NullPointerException("top layer");
// add a cause
e.initCause(new ArithmeticException("cause"));
throw e;
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
// display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " +
e.getCause());
}
}}
```

Dileep

The output from the program is shown here.

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

Where,

- ✓ The top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmeticException**. When the exception is thrown out of **demoproc( )**, it is caught by **main( )**.
- ✓ There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause( )**.

Note:

1. Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.
2. Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Advantages of using Exception Handling

- ➢ Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
- ➢ It is important to think of **try**, **throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in your program's logic.
- ➢ When a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

Dileeps