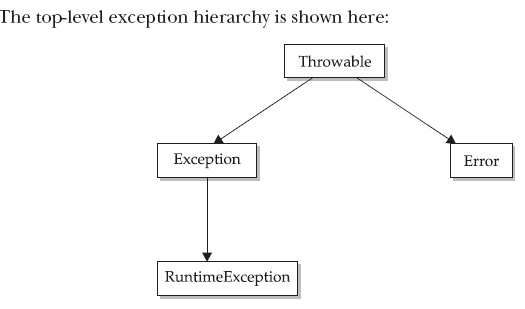
**Q1. What is the Throwable class? Differentiate between Exception and Error class**

Ans: The **Throwable** class supports Java’s exception-handling system and is the class from which all exception classes are derived.

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.



**Different between Exception and Error class**

**Exception**::

This class is used for exceptional conditions that user programs should catch. This is also the class that we will subclass to create our own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that we write and include things such as division by zero and invalid array indexing.

**Error**::

It 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.

**Q2. What is the default behaviour if a RuntimeException occurs without using try and catch? Give an example**.

Ans: 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.

**Example::**

Here is the small program includes an expression that

intentionally causes a divide-by-zero error::

class Exc0 {

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 **Exc0** 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.

Q3. Can we catch two or more exceptions occurring within the same try block?

Ans: Yes, 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 MultipleCatches {

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.");

}

Q4. What do you understand by nested try statements?

Ans: 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 for 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. Here is an example that uses nested **try** statements:

// An example of nested try statements.

class NestTry {

public static void main(String args[]) {

try {

int a = args.length;

/\* If no command-line args are present, the following statement will generate a divide-by-zero exception. \*/

int b = 42 / a;

System.out.println("a = " + a);

try { // nested try block

/\* If one command-line arg is used, then a divide-by-zero exception will be generated by the following code. \*/

if(a==1) a = a/(a-a); // division by zero

/\* If two command-line args are used, then generate an out-of-bounds exception. \*/

if(a==2) {

int c[] = { 1 };

c[42] = 99; // generate an out-of-bounds exception

}

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Array index out-of-bounds: " + e);

}

} catch(ArithmeticException e) {

System.out.println("Divide by 0: " + e);

}

}

}

Q5. Differentiate between **throw** and **throws** keyword.

Ans:

**Throws::**

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method’s declaration. A **throws** clause lists the types of exceptions that a method might throw. 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.

Q6. What is the significance of the finally block?

Ans: When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency. **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. 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.

Here is an example program that shows three methods that exit in various ways, none

without executing their **finally** clauses:

// Demonstrate finally.

class FinallyDemo {

// Throw 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.