MODULE 3

Exception Handling

INTRODUCTION

- An *exception is an* Abnormal condition that arises in a code sequence at runtime. In other words, an exception is a run-time error.

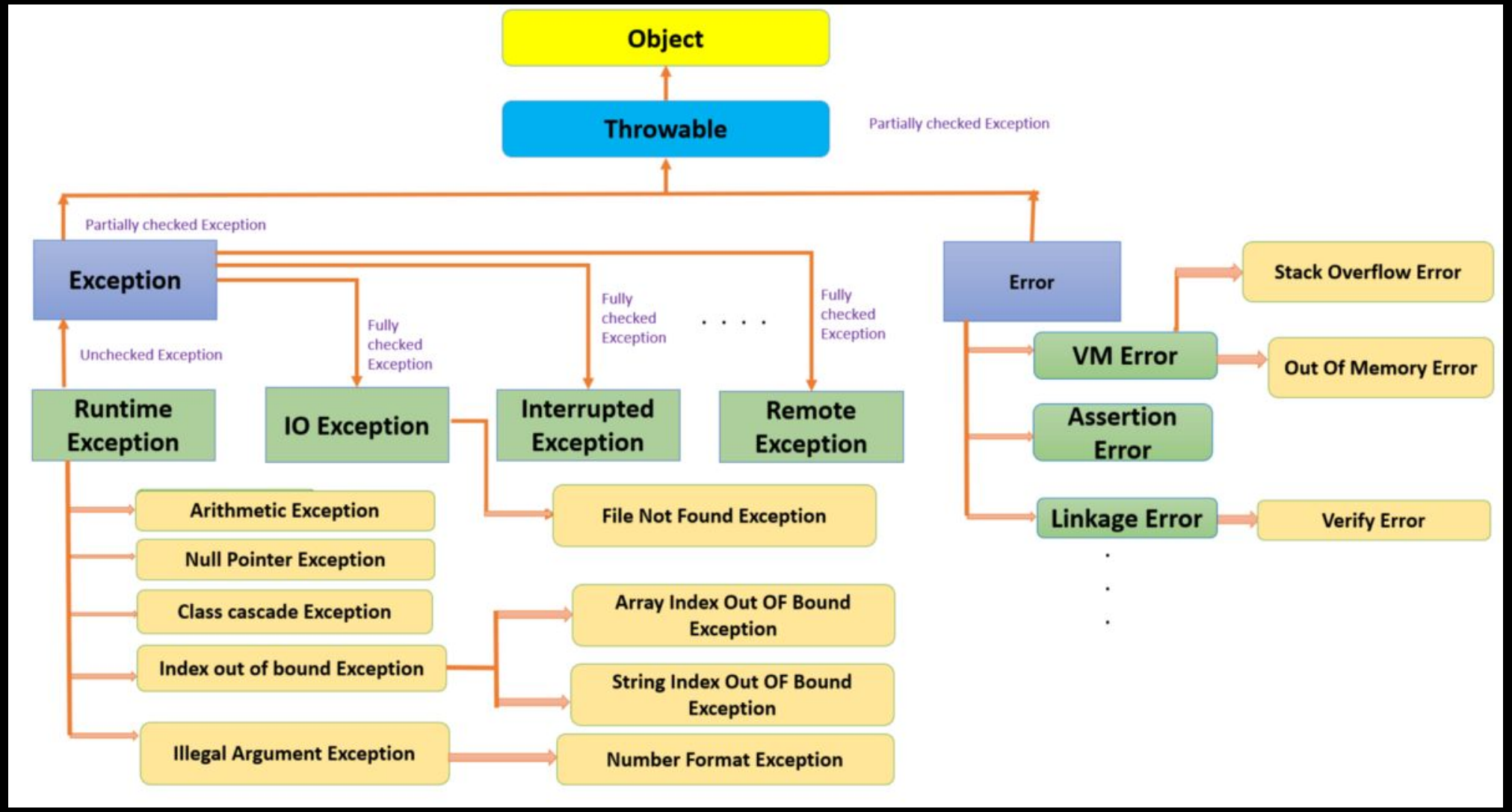
- When an exceptional condition arises, an object representing that exception is created and *thrown in the method* that caused the error.

- The 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.

INTRODUCTION

- All exception types are sub classes of the built- in class **Throwable.**



INTRODUCTION

- Exceptions are further classified as Checked and Unchecked Exception.

- **Checked Exception:** When JVM is already expecting an exception then, JVM want to include Exception causing statement in exception handling block. Ex: IOException, SQLException etc..

- **Unchecked Exception:** If unexpected exception is raised which is not written in program that is called Unchecked Exception. Ex: RunTimeException, ArithmaticException etc.

- **Error:** Error class exceptions are always part of Unchecked Exceptions, as they are always raised by system not by program.

INTRODUCTION

```
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 after try block ends
}
```

INTRODUCTION

- Once an exception is thrown, program control transfers out of the try block into the catch block. catch is not “called,” so execution never “returns” to the try block from a catch.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.
- The statements that are protected by try must be surrounded by curly braces.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).

MULTIPLE CATCH BLOCKS

- 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 by passed, and execution continues after the try/catch block.
- When you use multiple catch statements, it is important to remember that exception sub classes must come before any of their super classes.
- Article: https://www.javatpoint.com/multiple-catch-block-in-java to understand Multiple catch with examples.

NESTED TRY BLOCK

- 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.
- EX:

THROW

- Throw is used to throw an exception explicitly.
- General Form: throw *ThrowableInstance;*
- Here, *ThrowableInstance must be an object of type Throwable or a subclass of Throwable*
- Primitive 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 in 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.
- The nearest enclosing try block is inspected to see if it has catch statement that matches the type of 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. EX:

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.
- 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.* *Ex:*

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.

```
try{   //risk code;     }
catch(Exception x)  {       //Risk handled;          }
  finally    {   //Clean Up Activities;          }
```

- 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.
- If a try block contains return statement, then finally block will be executed before terminating the return.
- Ex: