Java Exception Handling

- What is an Exception?
- What happens when an Exception occurs?
- Benefits of Exception Handling framework
- Catching exceptions with try-catch
- Catching exceptions with finally
- Throwing exceptions
- Rules in exception handling
- Exception class hierarchy
- Checked exception and unchecked exception
- Creating your own exception class

What is an Exception?

- An exception is an unexpected event that occurs during the execution of a program that disrupts the normal flow of instructions.

Examples:

- Divide by zero errors
- Accessing the elements of an array beyond its range
- Invalid input
- Hard disk crash
- Opening a non-existent file
- Heap memory exhausted

- **Exception Example:**

```
class Division {
        public static void main(String args[]) {
                int result = division(3,0);
                System.out.println(”result:”+result);
        }
        public static int division (int a, int b) {
            return a/b;
        }
}
```

- Displays this error message

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at Division.main(Division.java:7)
```

Default exception handler:

– Provided by Java runtime

– Prints out exception description

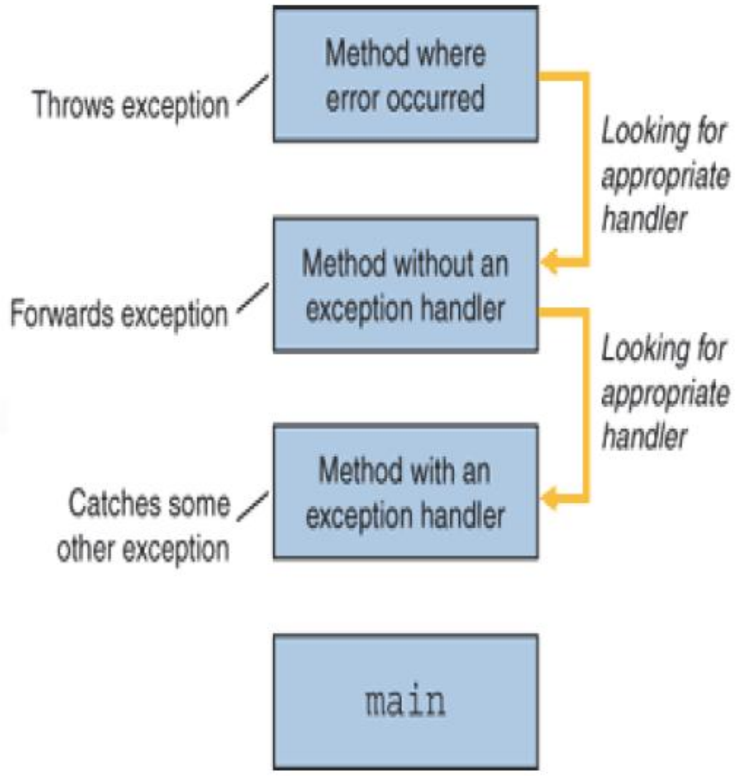
– Prints the stack trace

- **What Happens When an Exception Occurs?**

  When an exception occurs within a method, the method creates an exception object and hands it off to the runtime system.

1. Creating an exception object and handing it to the runtime system is called “throwing an exception”
2. Exception object contains information about the error, including its type and the state of the program when the error occurred

Searching the Call Stack for an Exception Handler


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

- **Benefits of Java Exception Handling Framework**

1. Separating Error-Handling code from “regular” business logic code
2. Propagating errors up the call stack
3. Grouping and differentiating error types

- **Catching Exceptions:**

try-catch **Statements:**

Syntax:

```
try {
            <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
            <handler if ExceptionType1 occurs>
}
```

Example:

```
public class Division {
            public static void main(String args[]) {
                        int result = 0;
                        try {
                            result = division(3, 0);
                        } catch (ArithmeticException e) {
                             System.err.println(e);
                        }
                        System.out.println("result:" + result);
            }
            public static int division(int a, int b) {
                        return a / b;
            }
}
```

Try with multiple catch:

Syntax:

```
try {
  // code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

Finally block:

- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

try-finally block:

```
try {

  // business logic

} finally {

 // resource cleanup code

}
```

try-catch-finally block:

```
try {
  // business code

} catch(Exception e) {

  // exception code

} finally {

  // resource cleanup code

}
```

• *finally* Keyword

Block of code is always executed despite of different scenarios:

– Forced exit occurs using a *return*, a *continue* or a *break* statement

– Normal completion

– Caught exception thrown

Output of given program:

```
public class ArthExample {
Public static int add(int a, int b) {
    try {
                   return (a+b);
       } catch(Exception e) {
                   return 6;
        } finally {
                   return 10;
        }
  }

public static void main(String args[]) {
    int result = Invovation: ArthExample .add(6,4);
    System.out.println(result);
  }
}
```

Throws/Throw Keywords

throws is used when the programmer does not want to handle the exception and throw it out of a method.
Then method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

throw is used when the programmer wants to throw an exception explicitly and wants to handle it using catch block.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

```
Public class Account {
    public int withdraw(String accountNum,double amount) throws
CustomerException {
    if(amount < 0) {
        throw new CustomerException(“Enter valid amount”);
    }
    try {
          // Fetch balance
    }catch(Exception e) {
      throw new CustomerException(“Error occurred while fetching balance”);
    }
  }
}
```

- **Rules on Exceptions**

1.A method is required to either catch or list all exceptions it might throw

Except for *Error* or *RuntimeException*, or their subclasses

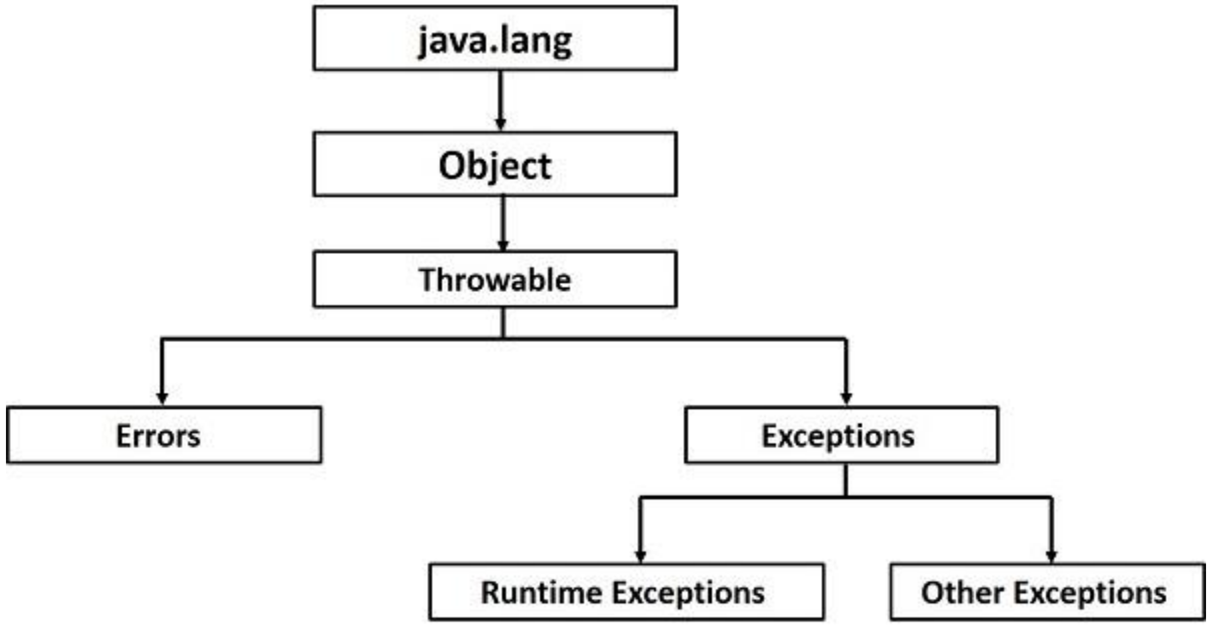
2. If a method may cause an exception to occur but does not catch it, then it must say so using the *throws* keyword.

Applies to checked exceptions only

Syntax:

```
<type> <methodName> (<parameterList>) throws <exceptionList> {
        <methodBody>
}
```

- **Exception Class Hierarchy:**



Throwable class

– Root class of exception classes

– Immediate subclasses

● Error

● Exception

Exception class

– Conditions that user programs can reasonably deal with

– Usually the result of some flaws in the user program code

– Examples

● Division by zero error

● Array out-of-bounds error

Error class

– Used by the Java run-time system to handle errors occurring in the run-time environment

– Generally beyond the control of user programs

– Examples

● Out of memory errors

● Hard disk crash

Checked and Unchecked Exceptions

Checked exception:

- checked exceptions are forced by compiler and used to indicate exceptional conditions that are out of the control of the program.
- Example: I/O Errors (IOExceptions) , Connection Broken (SQLExceptions)
- Java verifies checked exceptions at compile-time.
- if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword. otherwise the program will give a compilation error.
- The Exception class is the superclass of checked exceptions. Therefore, we can create a custom checked exception by extending Exception:

Unchecked exceptions:

- if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction.
- Example: ArithmeticException,NullPointerException,ArrayIndexOutOfBoundsException
- Not subject to compile-time checking for exception handling
- Built-in unchecked exception classes
  - RuntimeException
  - Their subclasses

Creating Your Own Exception Class:

Steps to follow

– Create a class that extends the RuntimeException or the Exception class

– Customize the class Members and constructors may be added to the class

Example:

```
public class CustomerException extends RuntimeException {
   /* some code */
}
```

```
Public class CustomerException extends Exception {
   public CustomerException(String exceptionMessage) {
         super(exceptionMessage);
    }
    public CustomerException(Throwable cause) {
         super(cause);
    }

}
```

Sr.No.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.