Lesson 10

Exception in Java

By the end of this lesson you will learn:

- What is Exception-Handling in Java
- Exception Types
- Exception-Handling Mechanism in Java
- User Defined Exception

What is Exception-Handling in Java

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

A Java exception is an **object** that describes an exceptional 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**.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the ***call stack.***

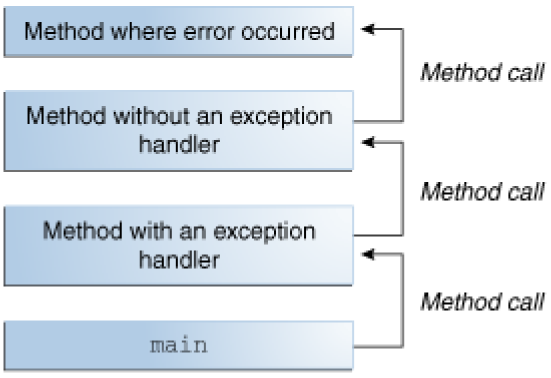
Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

Fig: The Call Stack

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler**. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler. The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the below figure, the runtime system (and, consequently, the program) terminates.

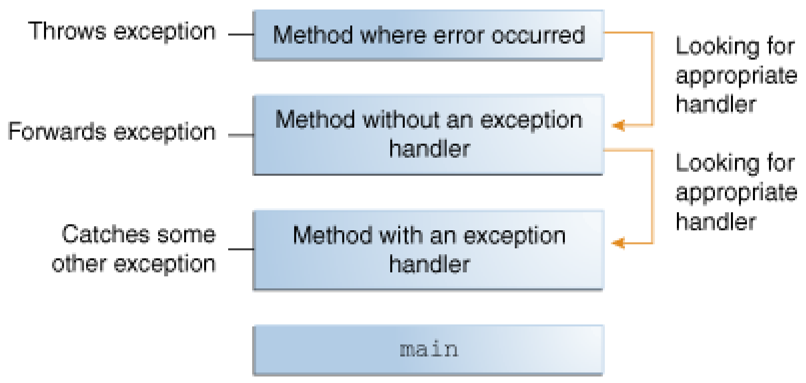
Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Fig: Searching the call stack for the exception handler.

Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception Types

All exception types are subclasses of the built-in class **Throwable.** Immediately below Throwable are two subclasses.

- ❖ **Exception**
- ❖ **Error**

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.

Error

- 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.

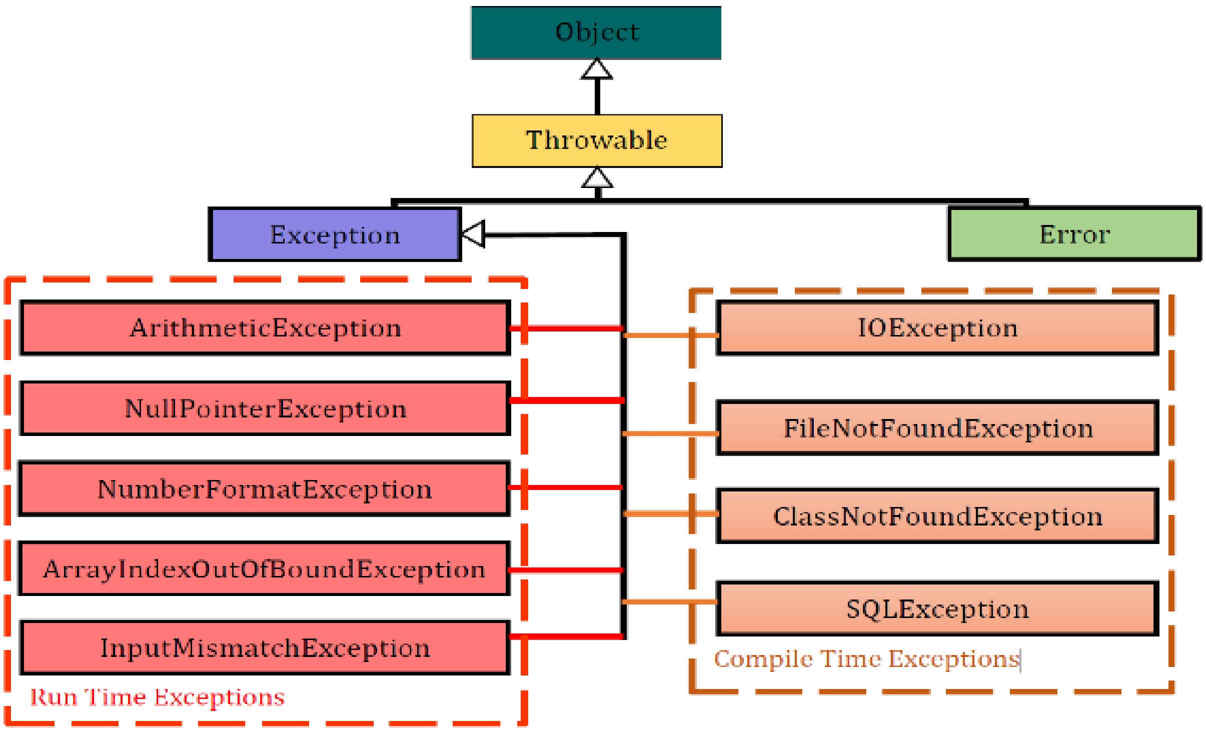

Fig: Exception Hierarchy in Java

There are two types of exceptions:

1. **Checked Exception (Compile-time Exception)**
2. **Unchecked Exception (Run-time Exception)**

Checked Exception

- A checked exception is an exception that is checked (notified) by the compiler at compilation time, these are also called as **compile time exceptions**.
- These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for

java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the ***FileReader*** object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws ***java.io.FileNotFoundException***.

- A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.
- Checked exceptions are subject to the Catch or Specify Requirement.
- All exceptions are checked exceptions, except for those indicated by **Error**, **RuntimeException**, and their subclasses.
- Some checked exceptions are FileNotFoundException, IOException, ClassNotFoundException, SQLException.

Unchecked Exception

- An unchecked exception is an exception that occurs at the time of execution. These are called as Runtime Exceptions.
- These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.
- These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for ***FileReader***. If a logic error causes a **null** to be passed to the constructor, the constructor will throw ***NullPointerException***. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.
- Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by **RuntimeException** and its subclasses.
- **Errors** and **Runtime Exceptions** are collectively known as unchecked exceptions.
- Some unchecked exceptions are ArithmaticException, NumberFormatException, InputMismatchException, ArrayIndexOutOfBoundException, NullPointerException.

Exception-Handling Mechanism in Java

Java has 5 keywords for exception handling:

- **try** - Program statements that you want to monitor for exceptions are contained within a try block.
- **catch** - 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.
- **finally** - Any code that absolutely must be executed after a try block completes is put in a finally block.
- **throw** - To manually throw an exception, use the keyword throw
- **throws** - Any exception that is thrown out of a method must be specified as such by a throws clause

Using try and catch

1) To guard against and handle a run-time error, simply enclose the code that you want

to monitor inside a try block.

2) Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

```
class Exc2 {
     public static void main(String args[]) {
          int d, a;
          try {
               d = 0;
               a = 42 / d;
               System.out.println("This will not be printed.");
          }
          catch (ArithmeticException e) {
               System.out.println("Division by zero.");
          }
          System.out.println("After catch statement.");
     }
}
```

Output

```
Division by zero.
After catch statement.
```

Let us examine the above example

- Once an exception is thrown, program control transfers out of the try block into the catch block, that is why **println**( ) inside the try block is never executed.
- 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.
- Once the catch statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

Multiple catch Clauses

In some cases, 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.

```
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.");
     }
}
```

This program will cause a **division-by-zero** exception if it is started with no commandline arguments, since a will equal zero. It will survive the division if you provide a command-line 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]**.

When you use multiple catch statements, it is important to remember that exception **subclasses** must come before any of their **superclasses**. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
class SuperSubCatch {
     public static void main(String args[]) {
          try {
               int a = 0;
               int b = 42 / a;
          }
          catch(Exception e) {
               System.out.println("Generic Exception catch.");
          }
          catch(ArithmeticException e) { // ERROR
               System.out.println("This is never reached.");
          }
     }
}
```

Nested try Statements

```
class NestTry {
     public static void main(String args[]) {
          try {
               int a = args.length;
               int b = 42 / a;
               System.out.println("a = " + a);
               try { // nested try block
                    if(a==1) a = a/(a-a);
I                   if(a==2) {
                         int c[] = { 1 };
                         c[42] = 99;
                    }
               }
               catch(ArrayIndexOutOfBoundsException e) {
                    System.out.println("Array index out-of-
               bounds: " + e);
               }
          }
          catch(ArithmeticException e) {
               System.out.println("Divide by 0: " + e);
          }
     }
}
```

The program works as follows. When you execute the program with no command-line arguments, a divide-byzero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

Using throw

- So far, you have only been catching exceptions that are **thrown** by the **Java run-time system**.
- However, it is possible for your program to **throw an exception explicitly**, using the **throw** statement.
- The general form of throw is shown here:

throw ThrowableInstance;

- Here, **ThrowableInstance** must be an object of type Throwable or a subclass of Throwable.
- 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 a 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.

```
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.

Using throws

- If a method can cause an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- It is achieved by including a **throws** clause in the method's declaration.
- A throws clause **lists the types of exceptions** that a method might throw.

See the following example.

```
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);
          }
     }
}
```

Suppose **throwOne** throw two exceptions, then the syntax will be the following.

```
static void throwOne() throws IllegalAccessException,
IndexOutOfBoundsException {
     .......................
}
```

Using finally

The ***finally*** block *always* executes when the ***try*** block exits. This ensures that the ***finally*** block is executed even if an unexpected exception occurs. But ***finally*** is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a ***return***, ***continue***, or ***break***. Putting cleanup code in a ***finally*** block is always a good practice, even when no exceptions are anticipated.

```java
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();
     }
}
```

```
Output
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

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.

User Defined Exception

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** (which is, of course, a subclass of **Throwable**). Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

See the following example, we have an Account class, which is representing a bank account where you can deposit and withdraw money, but what will happen if you want to withdraw money which exceeds your bank balance? You will not be allowed, and this is where user defined exception comes into the picture. We have created a custom exception called ***NotSufficientFundException*** to handle this scenario. This will help you to show a more meaningful message to user and programmer.

```
class NotSufficientFundException extends Exception {
     private String message;
     public NotSufficientFundException(String message) {
          this.message = message;
     }
     public String getMessage() { return message; }
}

public class Account {
     private int balance = 1000;

     public int balance() {
          return balance;
     }

     public void withdraw(int amount) throws
     NotSufficientFundException {

          if (amount > balance) {
               throw new
NotSufficientFundException(String.format("Current balance %d is
less than requested amount %d", balance, amount));
          }
          balance = balance - amount;
     }

     public void deposit(int amount) {
          if (amount <= 0) {
               throw new
IllegalArgumentException(String.format("Invalid deposit amount
%s", amount));
          }
     }
}
```

In the above example, we have a class ***NotSufficientFundException*** which is extending the **Exception** class and we are throwing the exception from **Account** class.

Exercise

1. What is difference between throw and throws keyword in Java.
2. What is difference between Checked and Unchecked Exception in Java.
3. What is difference between final, finally.
4. What happens when exception is thrown by main method?
5. Create a class registration with the following attributes:
    a. Id
    b. Name

c. Age

Create a custom exception AgeLimitException class to validate the age attribute. If age is less than 18, it will throw an error.