KLFSPRO
Learn & Grow

Core Java

Exception Handling

Lesson Objective

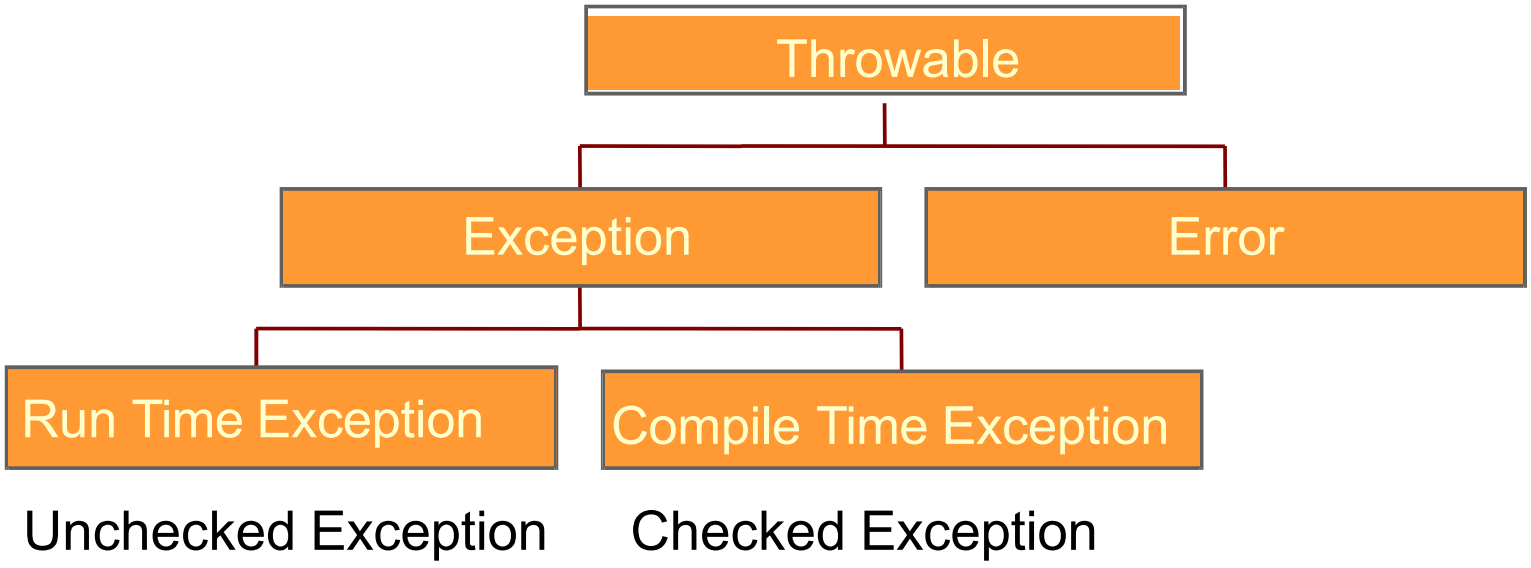
- What is Exception?
- Why Exception Handling?
- Java Exception Class Hierarchy
- Handle Exception in Java
  - Using try and catch
  - Multiple catch
  - Finally Clause
  - Throwing an Exception
  - Throws Clause
- Create your own Exceptions

What is Exception?

- Are abnormal events that might occur during program execution
- They terminate program execution abruptly
- Such abnormal events have to be handled to prevent the execution of the program from being terminated abruptly
- Examples:
  - Hard disk crash;
  - Out of bounds array access;
  - Divide by zero, and so on

Use of Exception Handling?

- No matter how well-designed a program is, there is always a chance that some kind of error will arise during its execution, for example:
  - Attempting to divide by 0
  - Attempting to read from a file which does not exist
  - Referring to non-existing item in array
- Programmer should always be prepared for the worst.
- The preferred way of handling such conditions is to use exception handling, an approach that separates a program's normal code from its error-handling code.


Throwable
Exception
Error
Run Time Exception
Compile Time Exception
Unchecked Exception
Checked Exception

Exception

- The **Exception** class and its subclasses are a form of **Throwables.** They indicate conditions, which a reasonable application may want to catch.

Exception Types

- **Checked Exception**
  - They are checked by the compiler at the time of compilation.
  - They must be handled in your code, or passed to parent classes for handling.
  - Some examples of **Checked exceptions** include:
    - **IOException**, **SQLException**, **ClassNotFoundException**

- **UnChecked Exception**
  - It is called **unchecked exception** because the compiler does not check to see if a method handles or throws these exceptions.
  - Example : ArithmeticException, ArrayIndexOutOfBoundsException

Handling Exception

- Using try and catch
- Multiple catch
- Finally Clause
- Throwing an Exception
- Throws Clause

Keywords for Exception Handling

- try
  - This marks the start of a block associated with a set of exception handlers.
- catch
  - The control moves here if an exceptions is generated.
- finally
  - This is called irrespective of whether an exception has occurred or not.
- throws
  - This describes the exceptions which can be raised by a method.
- throw
  - This raises an exception to the first available handler in the call stack, unwinding the stack along the wayBehavior of an object determines how an object reacts to other objects

Try and Catch

- The **try** structure has three parts:
  - The **try** block
    - Code in which exceptions are thrown
  - One or more **catch** blocks
    - To respond to various types of Exceptions
  - An optional **finally** block
    - Code to be executed last under any circumstances
- The **catch** Block:
  - If a line in the **try** block causes an exception, program flow jumps to the **catch** blocks.
  - If any **catch** block matches the exception that occurred that block is executed.

Using Try and Catch

```
try {
        // code in which exceptions may be thrown
  } catch (ExceptionType1   identifier) {
        // code executes if an ExceptionType1 occurs
  } catch (ExceptionType2 identifier) {
        // code executes if an ExceptionType2 occurs
  } finally  {
        // code executed last in any case
  }
```

Using Try and Catch

```
class DefaultDemo {
    public static void main(String a[]) {
      String str = null;
      try {
            str.equals("Hello");
        } catch(NullPointerException ne) {
                str = new String("Hello");
                System.out.println(str.equals("Hello"));
              }
       System.out.println("Continuing in the program…..."); }
}
```

Multiple Catch Blocks

- If you include multiple **catch** blocks, the order is important.
- You must catch subclasses before their ancestors.

```
public void divide(int x,int y)
{
   int ans=0;
   try{
          ans=x/y;
   }catch(Exception e) { //handle }
   catch(ArithmeticException f) {//handle}   //error
```

Nested Try Catch Block

```
try {
  int a = arg.length;  int b = 10 / a;
  System.out.println("a = " + a);
try {
  if(a==1)
  a = a/(a-a);
  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); }
```

Finally Clause

- The **finally** block is optional.
- It is executed whether or not exception occurs.

```
public void divide(int x,int y)
{
   int ans;
   try{
   ans=x/y;
   }catch(Exception e) {  ans=0;  }
   finally{
System.out.println("Task Completed");   //  always executed
}
}
```

Throwing an Exception

- You can throw your own runtime errors:
  - To enforce restrictions on use of a method
  - To "disable" an inherited method
  - To indicate a specific runtime problem
- To throw an error, use the **throw** Statement
  - throw ThrowableInstance
- **ThrowableInstance** is any **Throwable** Object

Throwing an Exception

```
class ThrowDemo {
void proc() {
try {
   throw new ArithmeticException(“From Exception”);
} catch(ArithmeticException e) {
   System.out.println("Caught inside demoproc.");
   throw e; // rethrow the exception
} }
public static void main(String args[]) {
ThrowDemo t=new ThrowDemo();
try {
   t.proc();
} catch(ArithmeticException e) {
System.out.println("Recaught: " + e); } } }
```

- If a method might throw an exception, you may declare the method as “throws” that exception and avoid handling the exception yourself.

```
class ThrowsDemo {
public static void main(String args[]) {
try {
doWork();
} catch (ArithmeticException e) {
System.out.println("Exception: " + e.getMessage());
} }
static void doWork() throws ArithmeticException {
int array[] = new int[100];
array[100] = 100;
} }
```

User Defined Exceptions

- Write a class that extends(indirectly) Throwable.
- What Superclass to extend?
  - For unchecked exceptions: RuntimeException
  - For checked exceptions:
    - Any other Exception subclass or the Exception itself

```
class AgeException extends Exception{  private
public AgeException(String msg) {
        super(msg);
}
}
```

User Defined Exceptions-Other way

- Write a class that extends(indirectly) Throwable.
- What Superclass to extend?
  - For unchecked exceptions: RuntimeException
  - For checked exceptions:
    - Any other Exception subclass or the Exception itself

```
class AgeException extends Exception {
private int age;
AgeException(int a) {
age = a;
}
public String toString() {
return age+" is an invalid age"; } }
```