JAVA PROGRAMMING 1

Summer 2018 - Christian Hur

# Unit 11 Lecture - Exception Handling

Reading: Chapter 12

# Objectives:

* Learn about exceptions
* Try code and catch exceptions
* Throw and catch multiple exceptions
* Use the finally block
* Understand the advantage of exception handling
* Specify the exceptions that a method can throw
* Trace exceptions through the call stack (stack trace)
* Create your own Exception classes
* Use an assertion

# Introduction

Computer programs are prone to produce errors which could disrupt the normal flow of the program. An error can occur for various reasons including:

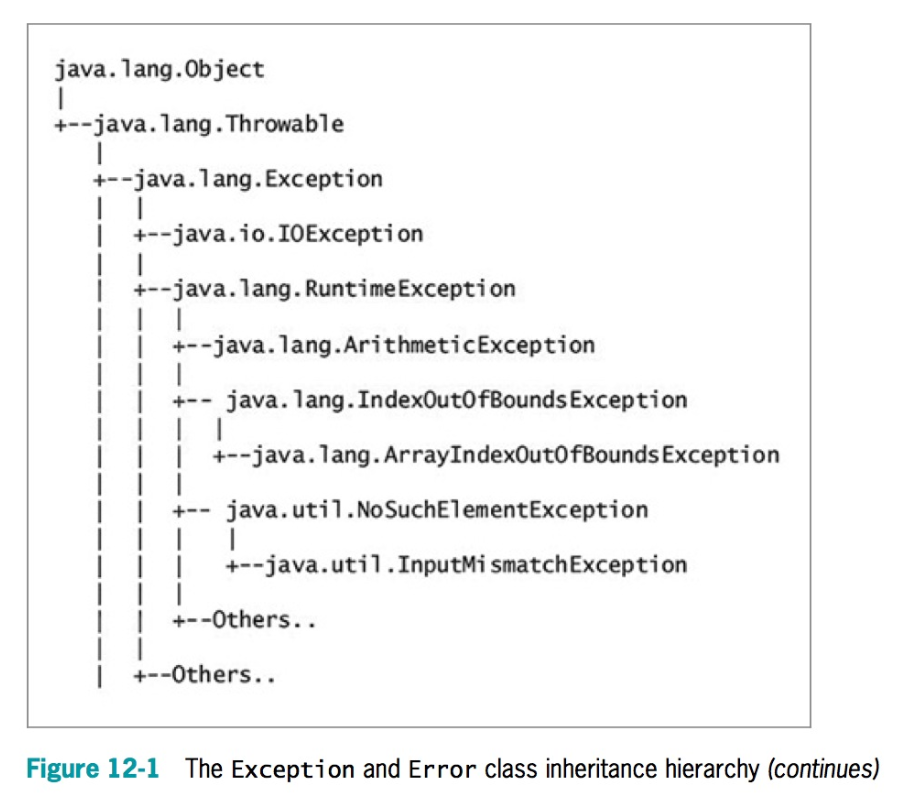
* Invalid input data
* A file not found
* Failure to write data to a disk
* Network connection lost
* An illegal mathemtical operation (e.g. divide by zero)
* Computer resources are exhausted

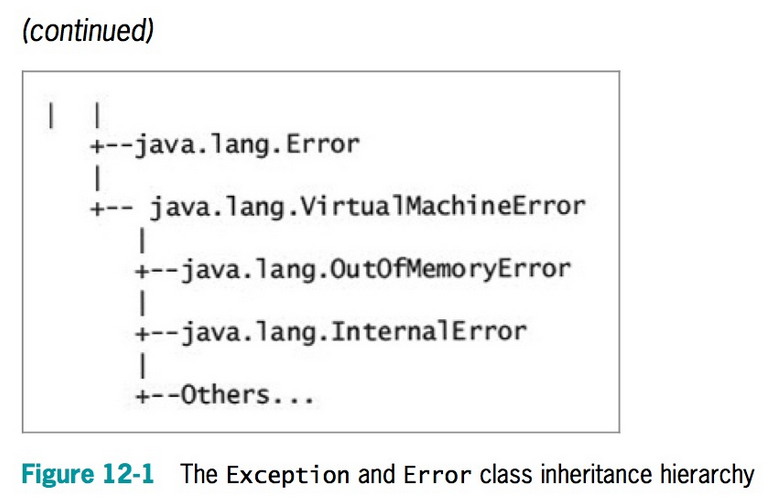
Errors such as these are called an “**exceptions**” (or exception events) because they are not usual occurrences; they are “exceptional.” They can arise during the execution of a program.

**Exception handling** is the name for the object-oriented techniques that manage or resolve such errors. Unplanned exceptions that occur during a program’s execution are also called **runtime exceptions**, in contrast with syntax errors that are discovered during program compilation.

## Error and Exception

Java includes two basic classes of errors: Error and Exception. Both of these classes descend from the Throwable class, as shown in Figure 12-1. Like all other classes in Java, Error and Exception originally descend from the Object class, which is defined in the automatically imported java.lang package.



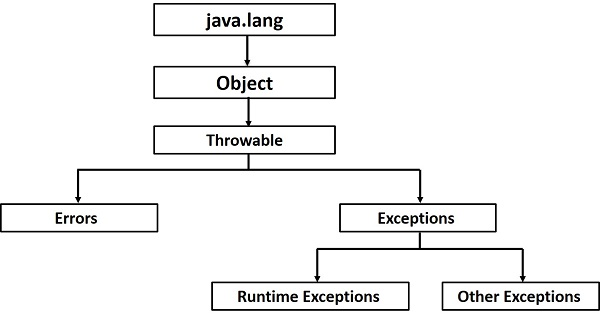


## 

## 

## 

## **Exception Hierarchy**



The **Error class represents more serious errors** from which your program usually cannot recover. For example, there might be insufficient memory to execute a program. Usually, you do not use or implement Error objects in your programs. A program cannot recover from Error conditions on its own.

The **Exception class comprises less serious errors that represent unusual conditions** that arise while a program is running and from which the program can recover. For example, one type of Exception class error occurs if a program uses an invalid array subscript value, and the program could recover by assigning a valid value to the subscript variable.

## Three Types of Exceptions

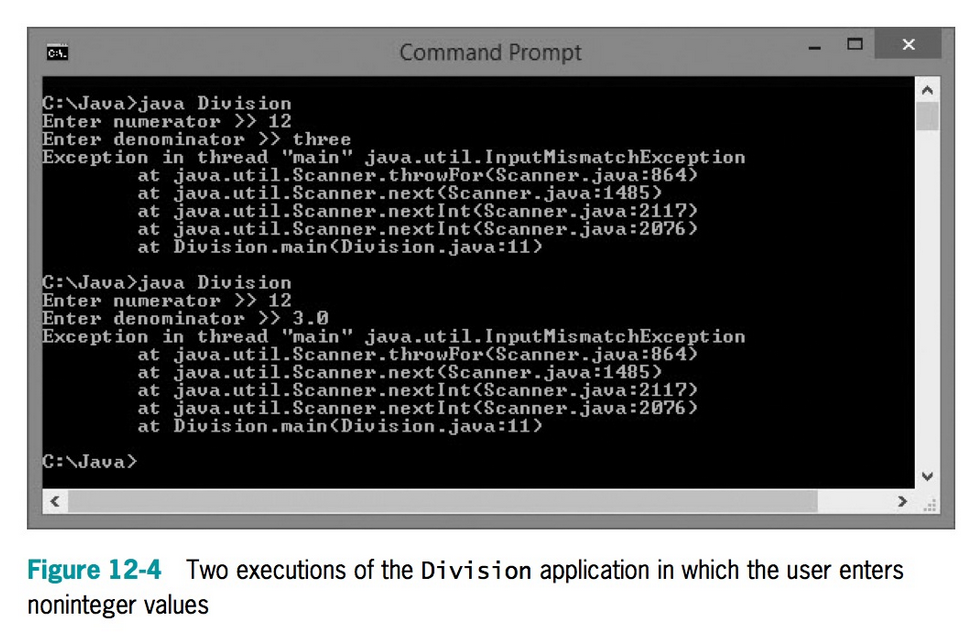
**Checked exceptions** − A checked exception is an exception that occurs at the compile time, these are also called as ***compile time exceptions***. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as ***Runtime Exceptions***. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

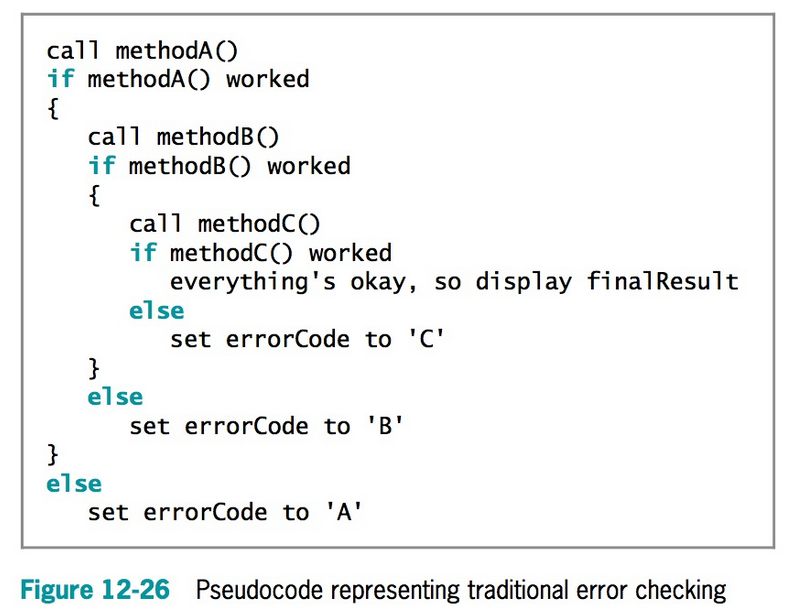
**Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

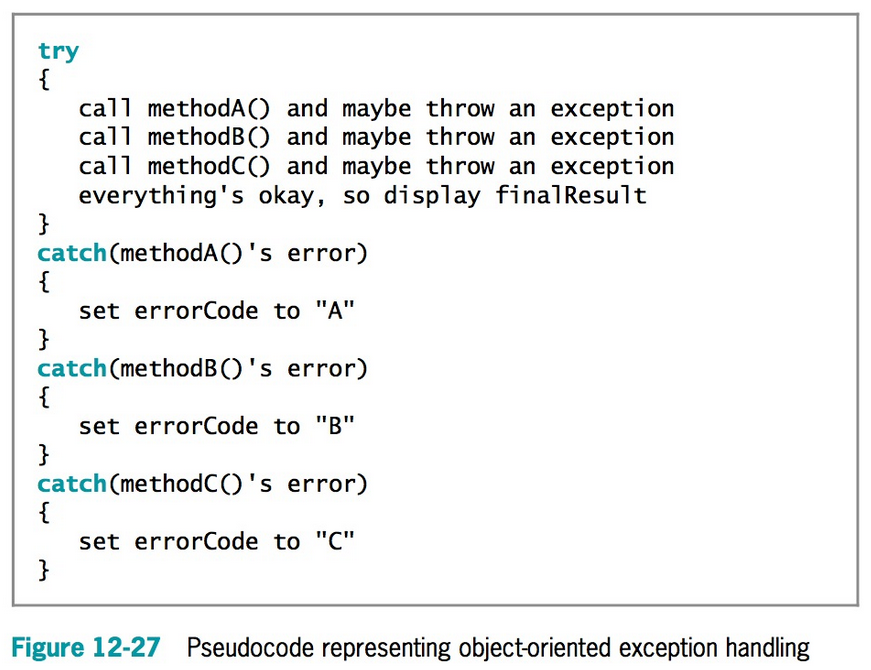
## Stack Trace

Stack trace (or stack backtrace, stack traceback) is a list of error messages after each attempted execution of a Java program. The list shows each method that was called.



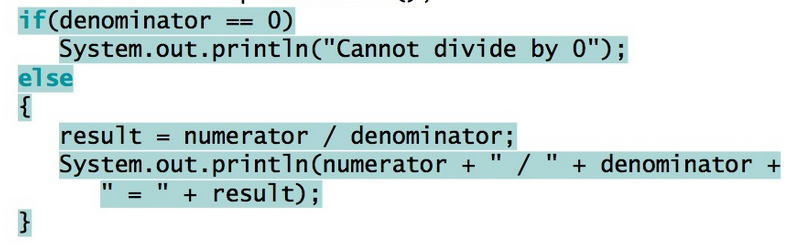
Traditionally, you can write your own custom codes to handle errors. Or you can let Java handle them for you. With some custom codes you can write better codes to handle errors more gracefully.





## Division By Zero Example

Handle the error using traditional method.



## Trying Code and Catching Exceptions

In object-oriented terminology, you “try” a procedure that might cause an error. A method that detects an error condition “throws an exception,” and if you write a block of code that processes the error, that block is said to “catch the exception.”

When you create a segment of code in which something might go wrong, you place the code in a **try block**, which is a block of code you attempt to execute while acknowledging that an exception might occur.

To handle a thrown exception, you can code one or more catch blocks immediately following a try block. A **catch block** is a segment of code that can handle an exception that might be thrown by the try block that precedes it. The exception might be one that is thrown automatically, or you might explicitly write a throw statement.

A **throw statement** is one that sends an Exception object out of a block or a method so that it can be handled elsewhere. A thrown Exception can be caught by a catch block. Each catch block can “catch” one type of exception—that is, one object that is an object of type Exception or one of its child classes.

try

{

//statement(s) that might generate an exception

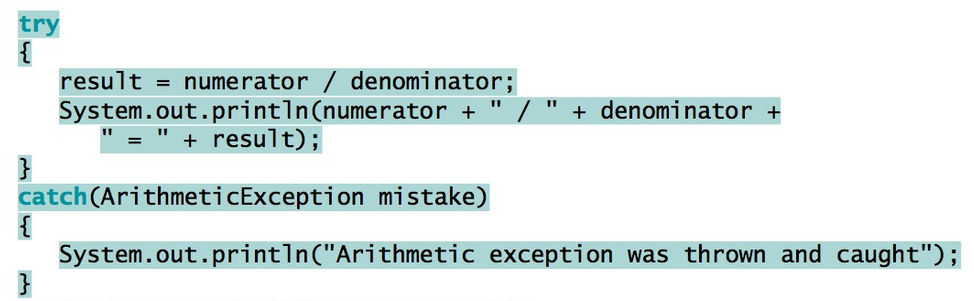
}

catch (Exception e)

{

//actions to take if exception occurs

}



### Accessing 3rd element and throwing an exception

try {  
 int a[] = new int[2];  
 System.out.println("Access element three :" + a[3]);  
}catch(ArrayIndexOutOfBoundsException e) {  
 System.out.println("Exception thrown :" + e);  
}

## Multiple Catch Blocks

You can place as many statements as you need within a try block, and you can catch as many exceptions as you want. If you try more than one statement, only the first error-generating statement throws an exception. As soon as the exception occurs, the logic transfers to the catch block, which leaves the rest of the statements in the try block unexecuted.

When a program contains multiple catch blocks, ***they are examined in sequence*** until a match is found for the type of exception that occurred. Then, the matching catch block executes, and each remaining catch block is bypassed.

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

Scanner input = new Scanner(System.in);

int numerator, denominator, result;

try {

System.out.print("Enter numerator >> ");

System.out.print("Enter denominator >> ");

System.out.println(numerator + " / " + denominator + " = " + result);

} catch(ArithmeticException mistake) {

System.out.println(mistake.getMessage());

} catch(InputMismatchException mistake) {

System.out.println("Wrong data type");

}

System.out.println("End of program");

## Catching Multiple Type of Exceptions

You can also handle multiple types of exceptions in a single catch block.

try {  
 // Protected code  
}catch(ExceptionType1|ExceptionType2|ExceptionType3 e) {  
 // Catch block  
}

## The **finally** block

When you have actions you must perform at the end of a **try...catch** sequence, you can use a **finally block**. The code within a finally block executes regardless of whether the preceding try block identifies an exception. Usually, you use a finally block to perform cleanup tasks that must happen regardless of whether any exceptions occurred and whether any exceptions that occurred were caught.

try

{

// statements to try

} catch(Exception e) {

// actions that occur if exception was thrown

}

finally

{

//actions that occur whether catch block executes or not

}

Pseudocode that tries reading a file and handles an IOException

try

{

// Open the file

// Read the file

// Place the file data in an array

// Calculate an average from the data

// Display the average

} catch(IOException e) {

// Issue an error message

// System exit

}

finally

{

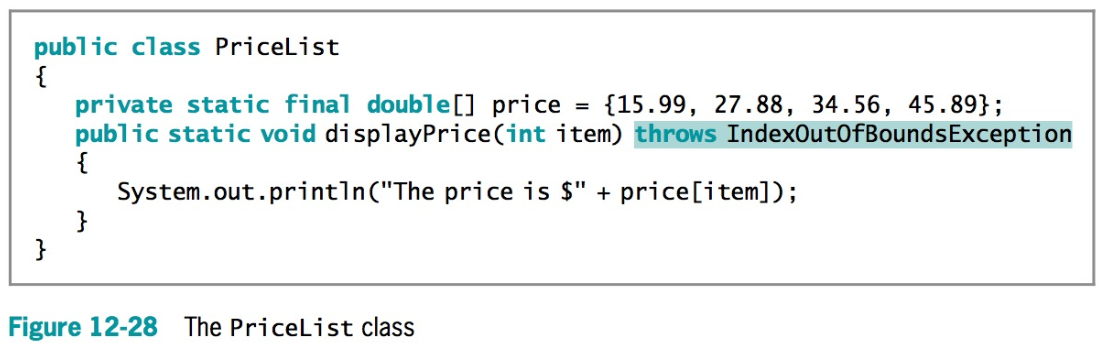
// If the file is open, close it

}

## Throws Clause

If a method throws an exception that it will not catch but that will be caught by a different method, you **must** create a **throws** clause by using the keyword throws followed by an Exception type in the method header. This practice is known as **exception specification**.

For example, Figure 12-28 shows a PriceList class used by a company to hold a list of prices for items it sells. For simplicity, there are only four prices and a single method that displays the price of a single item. The displayPrice() method accepts a parameter to use as the array subscript, but because the subscript could be out of bounds, the method contains a shaded throws clause, acknowledging it could throw an exception.

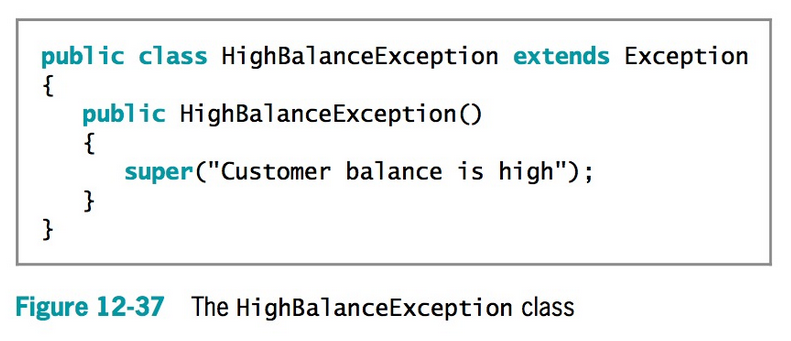


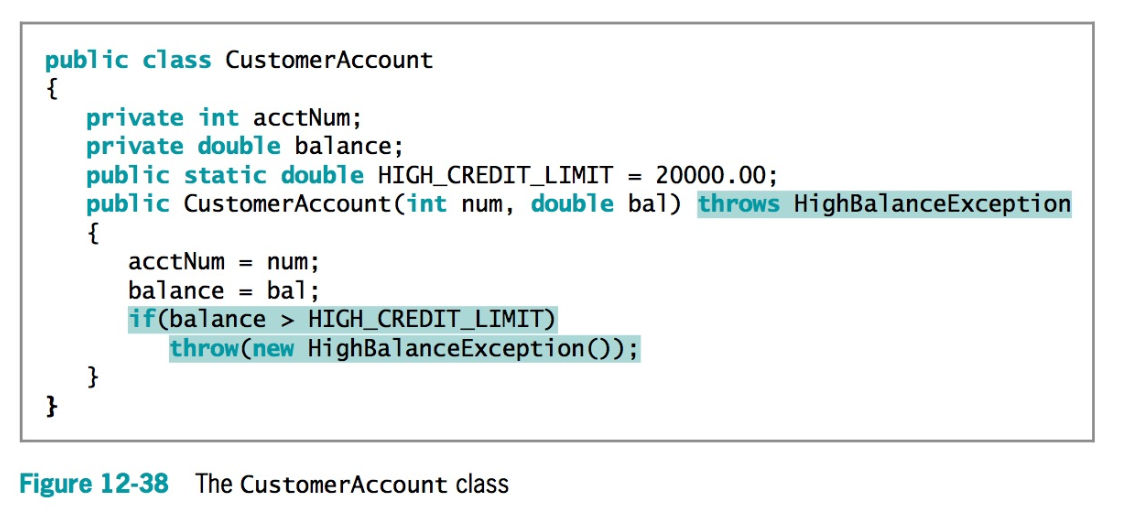
## Creating Your Own Exception Classes

To create your own throwable Exception class, you must extend a subclass of Throwable. Recall from Figure 12-1 that Throwable has two subclasses, Exception and Error, which are used to distinguish between recoverable and nonrecoverable errors. Because you always want to create your own exceptions for recoverable errors, your classes should extend the Exception class. You can extend any existing Exception subclass, such as ArithmeticException or NullPointerException, but usually you want to inherit directly from Exception. It is conventional to end each Exception subclass name with Exception.

The **Exception** class contains four constructors as follows: Exception()—Constructs a new

* **Exception()** object with null as its detail message
* **Exception(String message)**—Constructs a new Exception object with the specified detail message
* **Exception(String message, Throwable cause)**—Constructs a new Exception object with the specified detail message and cause
* **Exception(Throwable cause)**—Constructs a new Exception object with the specified cause and a detail message of cause.toString(), which typically contains the class and the detail message of cause, or null if the cause argument is null





Try{

Do sothing

} catch (IOExecption e) {

..

}

}catch (HighBalanceException e) {

System.out.print(e.getMessage());

}