Object-Oriented Programming

Chapter Five: Exception Handling in Java

Chapter Outline

- Introduction
- Types of Exceptions
- Exception handling
- Java's Exception Hierarchy
- Coding Exceptions
  - Try – catch mechanism
  - Passing exception
- Some common exceptions
- Declaring your own Exception
  - Example

Introduction

- An exception is a problem that arises during the execution of a program. It is a representation of an error condition or a situation that is not the expected result of a method/program.
- An exception can occur for many different reasons, including the following:
  - attempting to divide by zero (arithmetic exception)
  - reading a decimal value when an integer is expected (number format exception)
  - attempting to write to a file that doesn't exist (I/O exception)
  - or referring to a nonexistent character in a string (index out of bounds exception).
  - 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.

Introduction Cont.

- No matter how well-designed a program is, there is always the chance that some kind of error will arise during its execution.
- Raising an exception halts normal execution abruptly and alternative statements are sought to be executed.
- A well-designed program should include code to guard against errors and other exceptional conditions when they arise.
- This code should be incorporated into the program from the very first stages of its development.
- That way it can help identify problems during development.
- In Java, the preferred way of handling such conditions is to use **exception handling** - a divide-and-conquer approach that separates a program's normal code from its error-handling code.

Categories of Exception

- To understand how exception handling works in Java, you need to understand the three categories of exceptions:
  - **Checked exceptions:**
    - A checked exception is one that can be analyzed (can't be ignored) by the Java compiler.
    - That is when the compiler encounters one of these exceptions it checks whether the program either handles or declares the exception.
    - A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer.
    - For example, if a file is to be opened, but the file cannot be found, an exception occurs.

Categories of Exception…

- **Runtime exceptions:**
  - Runtime exception is an exception that occurs that probably could have been avoided by the programmer.
  - As opposed to checked exceptions, 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.

Exception Handling

- Exception handling is the technique of catching the exceptions that might be thrown some time in the future during runtime.
- Exceptions can be handled in traditional way of handling errors within a program with Java's default exception-handling mechanism or using Exception class defined in Java API.
- **Traditional way of Handling Errors**
  - Consider the following example

```
public double avgFirstN(int N) {
    int sum = 0;
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;       // What if N is 0?
}
```

Exception Handling…

```
public double avgFirstN(int N) {
    int sum = 0;
    if (N <= 0) {
      System.out.println("ERROR avgFirstN: N <= 0. Program terminating.");
      System.exit(0);
    }
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;              // What if N is 0?
} // avgFirstN()
```

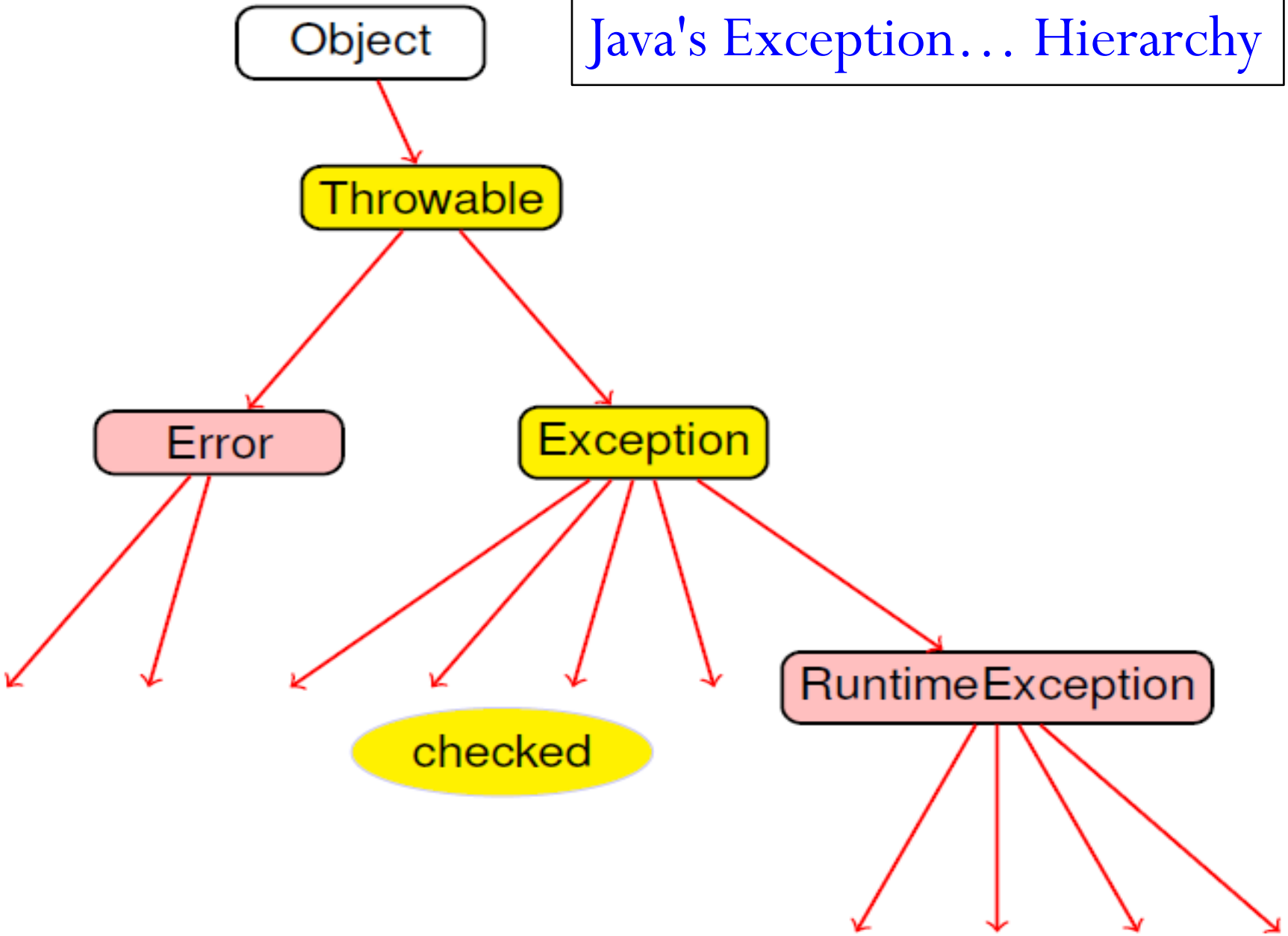
The error-handling code is built right into the algorithm

- This method has several problems
  - programmer must remember to always check the return value and take appropriate action. This requires much code (methods are harder to read) and something may get overlooked.
  - Poor modular decomposition
  - Hard to test such programs

Java's Exception Hierarchy

- The Java class library contains a number of predefined exceptions.
- All exception classes are subtypes of the java.lang.Exception class.
- The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
- Errors are not normally trapped from the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs.
- Errors are generated to indicate errors generated by the runtime environment.
  - Example : JVM is out of Memory. Normally programs cannot recover from errors.

Java's Exception… Hierarchy



Exception handling…

- How do you handle exceptions?
  - Exception handling is accomplished through the "try – catch" mechanism, or by a "throws or throw" clause in the method declaration.
  - For any code that throws a checked exception, you can decide to handle the exception yourself, or pass the exception "up the chain" (to a parent class).
  - To handle the exception, you write a "try-catch" block. To pass the exception "up the chain", you declare a throws clause in your method or class declaration.
  - If the method contains code that may cause a checked exception, you MUST handle the exception OR pass the exception to the parent class (remember, every class has Object as the ultimate parent)

Coding Exceptions

- Try-Catch Mechanism
  - Wherever your code may trigger an exception, the normal code logic is placed inside a block of code starting with the "try" keyword:
  - After the try block, the code to handle the exception should it arise is placed in a block of code starting with the "catch" keyword.
  - You may also write an optional "finally" block. This block contains code that is ALWAYS executed, either after the "try" block code, or after the "catch" block code.
  - Finally blocks can be used for operations that must happen no matter what (i.e. cleanup operations such as closing a file)
  - Generally, the try statement contains and guards a block of statements.

Coding Exception…

- Syntax of **try catch**

```
try {
    codes that may throw exception(s)
}
catch (exception_type identifier){
    //how do you want to deal with this exception
}
catch (exception_type identifier){
    //how do you want to deal with this exception
}
// you can use multiple catches to handle different exceptions
finally {
    // code that must be executed  under successful or unsuccessful
       conditions
}
```

Coding Exception…

- Example

```
1. int x = (int)(Math.random() * 5);
2. int y = (int)(Math.random() * 10);
3. int [] z = new int[5];
4. try {
5.   System.out.println("y/x gives " + (y/x));
6.   System.out.println("y is "
7.    + y + " z[y] is " + z[y]);
8. }
9. catch (ArithmeticException e) {
10.   System.out.println("Arithmetic problem " + e);
11. }
12. catch (ArrayIndexOutOfBoundsException e) {
13.   System.out.println("Subscript problem " + e);
14. }
```

Coding Exception…

- A catch block will catch exceptions of the class specified, including any exceptions that are subclasses of the one specified.
- A more specific catch block must precede a more general one in the source. Failure to meet this ordering requirement causes a compiler error.
- Only one catch block, that is the first applicable one, will be executed.
- The finally block always executed regardless an exception or not except the following conditions:
  - the death of the thread
  - the use of System.exit( )
  - turning off the power to the CPU
  - an exception arising in the finally block itself

Coding Exception…

- **Passing the exception:** In any method that might throw an exception, you may declare the method as "throws" that exception, and thus avoid handling the exception yourself
  - Example
    - public void myMethod throws IOException {
      … normal code with some I/O
      }
  - Examples:

```
public void openFile(String fileName) throws
        FielNotExistException, IOException{
        open the file using the input file name;
        if (can not open file)  throw new FileNotExistException ();
              read the file,
        if (any error occurs)throw new IOException();
    }
```

throws vs throw

Coding Exception…

- In general, any method that contains an expression that might throw a checked exception must declare the exception. (declaring exception). Example

```
import java.io.*;
public class Example {
    BufferedReader input = new BufferedReader (new
InputStreamReader(System.in));
    public void doRead() throws IOException {
        // May throw IOException
        String inputString = input.readLine();
    }
    public static void main(String argv[]) throws IOException
{
        Example ex = new Example();
        ex.doRead();
    }
}
```

```
public class CalcAverage {
  public double avgFirstN(int N) {
    int sum = 0;
    if (N <= 0)  throw new IllegalArgumentException("ERROR: Illegal argument");
    for (int k = 1; k <= N; k++)
      sum += k;
    return sum/N;// What if N is 0?
  } // avgFirstN()
} // CalcAverage class
```

If the CalcAverage.avgFirstN() method has a zero or negative argument, it will throw an exception:

```
TheException ex = new TheException();
throw ex;
```

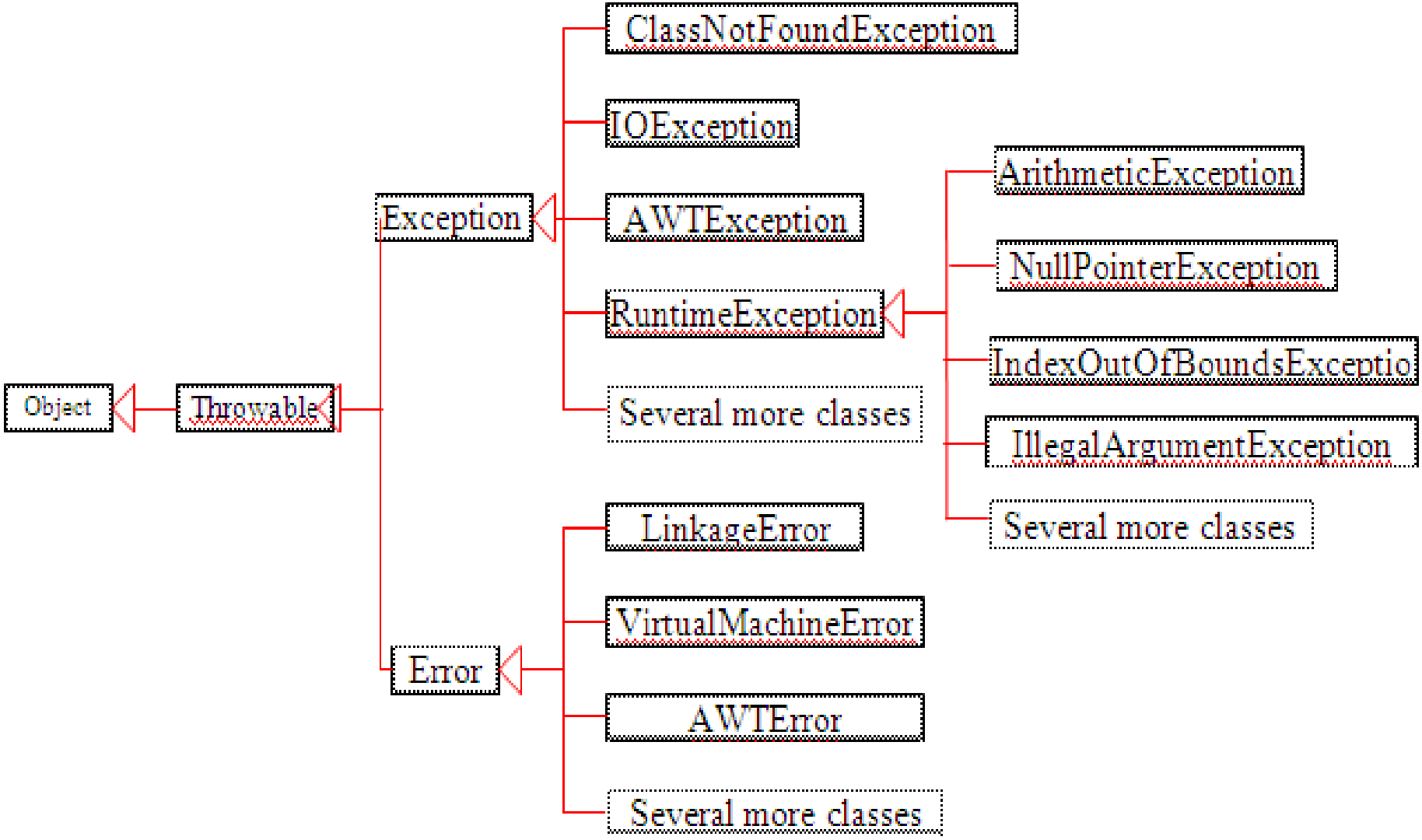
```
public class CalcAvgTest {
  public static void main(String args[]) {
    try {
      CalcAverage ca = new CalcAverage();
      System.out.println( "AVG + " + ca.avgFirstN(0));
    }
    catch (IllegalArgumentException e) { // Exception Handler
      System.out.println(e.getMessage());
      e.printStackTrace();
      System.exit(0);
    }
  } // main()
} // CalcAvgTest class
```

→the catch clause immediately follows the try block

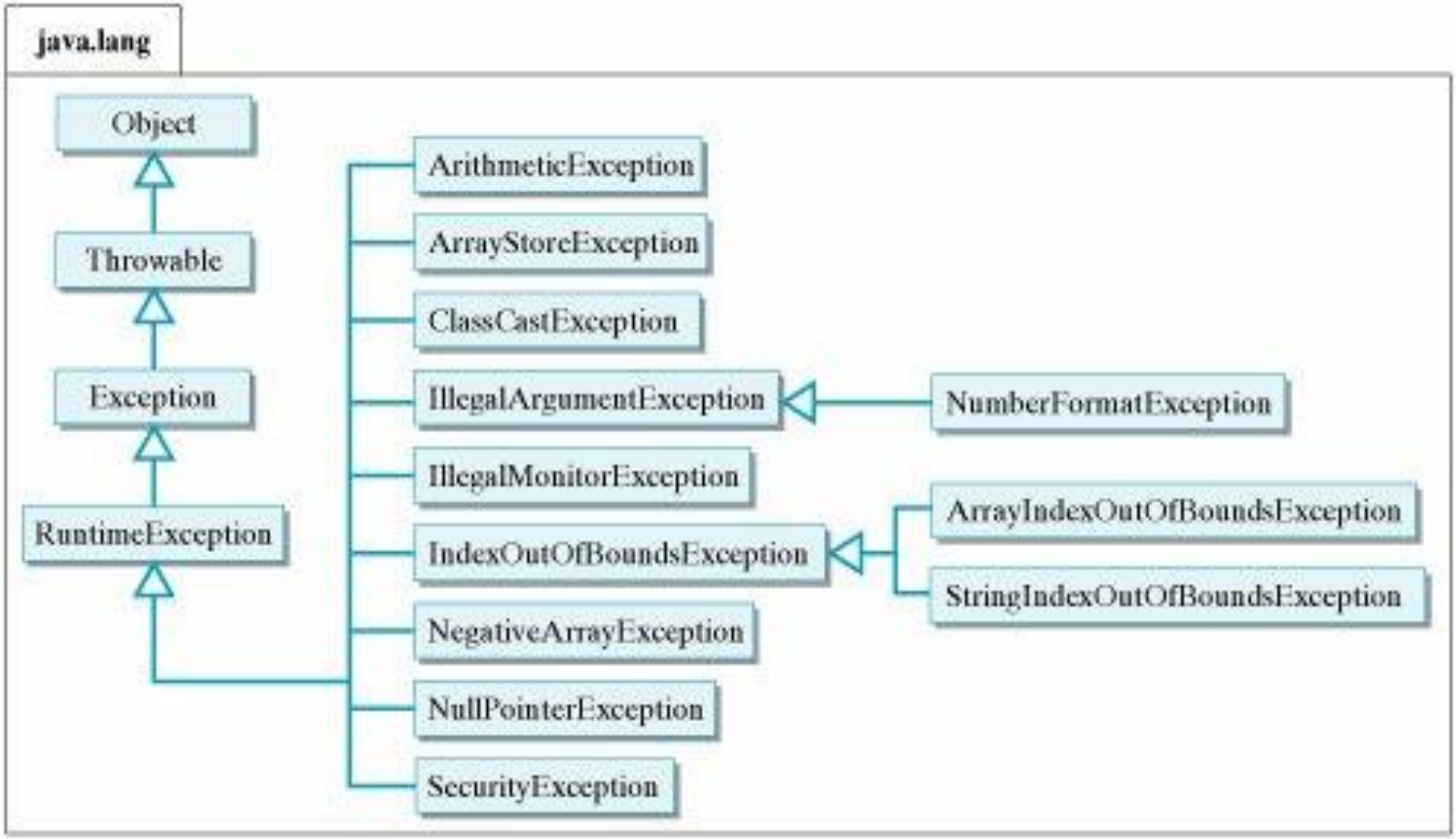
Some Common Exceptions

Class	Description
`ArithmeticException`	Division by zero or some other arithmetic problem
`ArrayIndexOutOfBoundsException`	An array index is less than zero or greater than or equal to the array's length
`FileNotFoundException`	Reference to a file that cannot be found
`IllegalArgumentException`	Calling a method with an improper argument
`IndexOutOfBoundsException`	An array or string index is out of bounds
`NullPointerException`	Reference to an object that has not been instantiated
`NumberFormatException`	Use of an illegal number format, as when calling a method
`StringIndexOutOfBoundsException`	A `String` index is less than zero or greater than or equal to the `String`'s length

Some Common Exceptions…



Some Common Exceptions…



Declaring you own Exception

- You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:
  - All exceptions must be a child of Throwable.
  - If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
  - If you want to write a runtime exception, you need to extend the RuntimeException class.
  - You can define our own Exception class as below:
    - class MyException extends Exception{ …}
  - You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions.
  - An exception class is like any other class, containing useful fields and methods.

Declaring you own Exception…

- The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception.

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends
Exception{
    private double amount;
    public InsufficientFundsException(double amount){
        this.amount = amount;
    }
    public double getAmount(){
        return amount;
    }
}
```

Declaring you own Exception…

- To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;
public class CheckingAccount{
   private double balance;
   private int number;
   public CheckingAccount(int number) {  this.number = number;  }
   public void deposit(double amount) {  balance += amount;   }
   public double getBalance() { return balance;   }
   public int getNumber() { return number;     }
```

Declaring you own Exception…

```
public void withdraw(double amount) throws
   InsufficientFundsException {
     if(amount <= balance) {
       balance-= amount;
     }else{
       double needs = amount - balance;
       throw new InsufficientFundsException(needs);
     }
   }
}
```

- The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

Declaring you own Exception…

```
public class BankDemo{
  public static void main(String [] args)   {
     CheckingAccount c = new CheckingAccount(101);
     System.out.println("Depositing $500…");
     c.deposit(500.00);
     try {
            System.out.println("\nWithdrawing $100…");
            c.withdraw(100.00);
            System.out.println("\nWithdrawing $600…");
            c.withdraw(600.00);
     }catch(InsufficientFundsException e)  {
            System.out.println("Sorry, but you are short $“  + e.getAmount());
            e.printStackTrace();
     }
   }
}
```

Declaring you own Exception…

- Compile all the above three files and run BankDemo, this would produce following result.

```
Depositing $500…
Withdrawing $100…
Withdrawing $600…
Sorry, but you are short $200.0
InsufficientFundsException
    at  CheckingAccount.withdraw(CheckingAccount.java:25)
    at BankDemo.main(BankDemo.java:13)
```

Creating a better error message for debugging: e.printStackTrace()

- Instead of using Java's e.getMessage() method to print errors during the debugging process, you can get more information about the error process if you print a stack trace from the exception.
- The snippet of source code shown below shows how to print the stack t.

```
try {
    // try to open the non-existent file
} catch (IOException e) {
    // you handle the exception here
    e.printStackTrace();
}
```

- Using this code snippet if you try to open a non-existent file , you'll get this output message:

Creating a better error message for debugging: e.printStackTrace()…

Java.io.FileNotFoundException: file name

at java.io.FileInputStream.<init>(FileInputStream.java)

at java.io.FileInputStream.<init>(FileInputStream.java)

at ExTest.readMyFile(ExTest.java:19) at ExTest.main(ExTest.java:7)

- As a final point - don't forget the e.getMessage() method - because the error message is not automatically printed to the screen.

Next Class : Multithreading