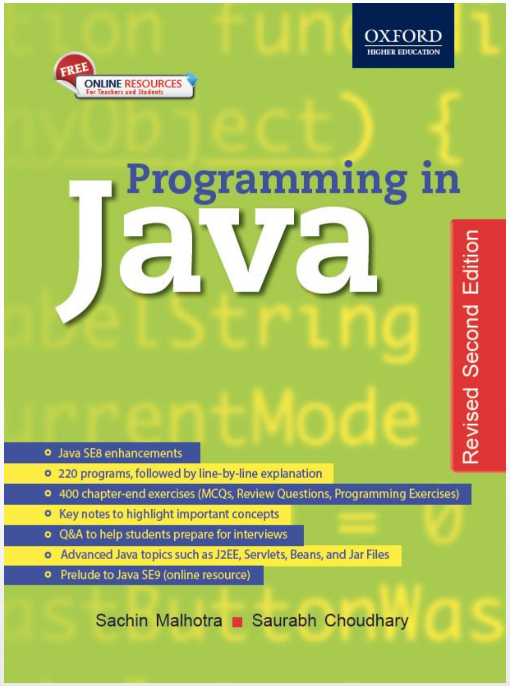
OXFORD
UNIVERSITY PRESS

# Programming in Java

Revised 2nd Edition

Sachin Malhotra & Saurabh Choudhary



OXFORD HIGHER EDUCATION

FREE ONLINE RESOURCES For Teachers and Students

Programming in Java

Revised Second Edition

- Java SE8 enhancements
- 220 programs, followed by line-by-line explanation
- 400 chapter-end exercises (MCQs, Review Questions, Programming Exercises)
- Key notes to highlight important concepts
- Q&A to help students prepare for interviews
- Advanced Java topics such as J2EE, Servlets, Beans, and Jar Files
- Prelude to Java SE9 (online resource)

Sachin Malhotra ■ Saurabh Choudhary

# Chapter 7

## Exception Assertion and Logging

# Objectives

- Understand the concept and application of exception handling
- Understand all the keywords used for exception handling
- Create user-defined exceptions
- Know what assertions are and how to use them
- Know the basics of logging

# Exception

- Exceptions in real life are rare.
- Are usually used to denote something unusual that does not conform to the standard rules.
- In programming, exceptions are events that arise due to the occurrence of unexpected behaviour in certain statements, disrupting the normal execution of a program.

# Causes of Exception

- Exceptions can arise due to a number of situations. For example,
  - ❑ Trying to access the 11th element of an array when the array contains of only 10 element (*ArrayIndexOutOfBoundsException*)
  - ❑ Division by zero (*ArithmeticException*)
  - ❑ Accessing a file which is not present (*FileNotFoundException*)
  - ❑ Failure of I/O operations (*IOException*)
  - ❑ Illegal use of null (*NullPointerException*)

# Exception Classes

- Top class in exception hierarchy is *throwable.*
- This class has two siblings: Error and Exception.
- All the classes representing exceptional conditions are subclasses of the Exception class.

# What Happens When an Execution Occurs

- Runtime environment identifies the type of Exception and throws the object of it.
- If the method does not employ any exception handling mechanism
  - the exception is passed to the caller method, and so on
- If no exception handling mechanism is employed in any of the Call Stack methods
  - the runtime environment passes the exception object to the default exception handler available with itself
  - the default handler prints the name of the exception along with an explanatory message followed by stack trace at the time the exception was thrown and the program is terminated

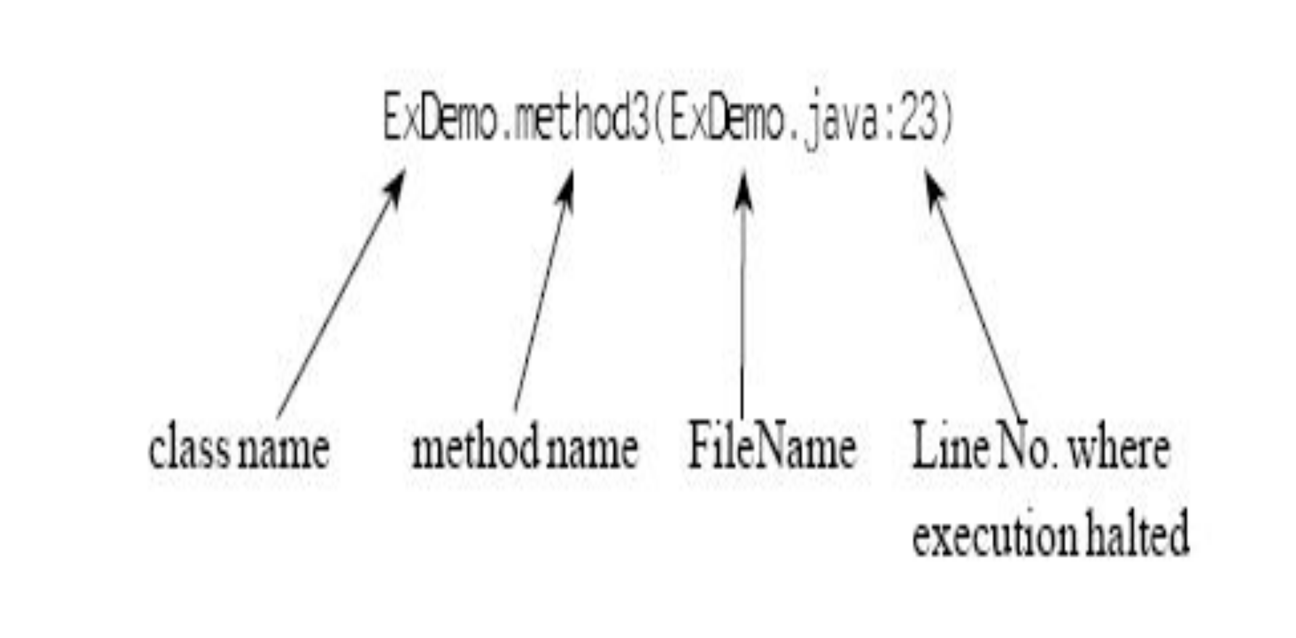
# Exception Example

```
class ExDemo {
public static void main(String args[]){
method1();     }
static void method1(){
    System.out.println(“IN Method 1, Calling Method 2”);
    method2();
    System.out.println(“Returned from method 2”);
}
static void method2(){
    System.out.println(“IN Method 2, Calling Method 3”);
    method3();
    System.out.println(“Returned from method 3”);
}
static void method3(){
System.out.println(“IN Method 3”);
int a=20,b=0;
int c=a/b;
System.out.println(“Method 3 exits”); }}
```

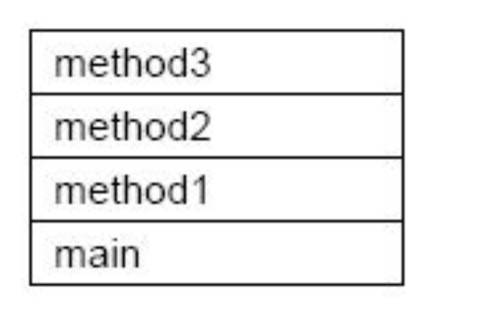
# The Output

```
IN Method 1, Calling Method 2
IN Method 2, Calling Method 3
IN Method 3
Exception in thread “main”
java.lang.ArithmeticException: / by zero
at ExDemo.method3(ExDemo.java:23)
at ExDemo.method2(ExDemo.java:16)
at ExDemo.method1(ExDemo.java:10)
at ExDemo.main(ExDemo.java:5)
```

# Stack Trace

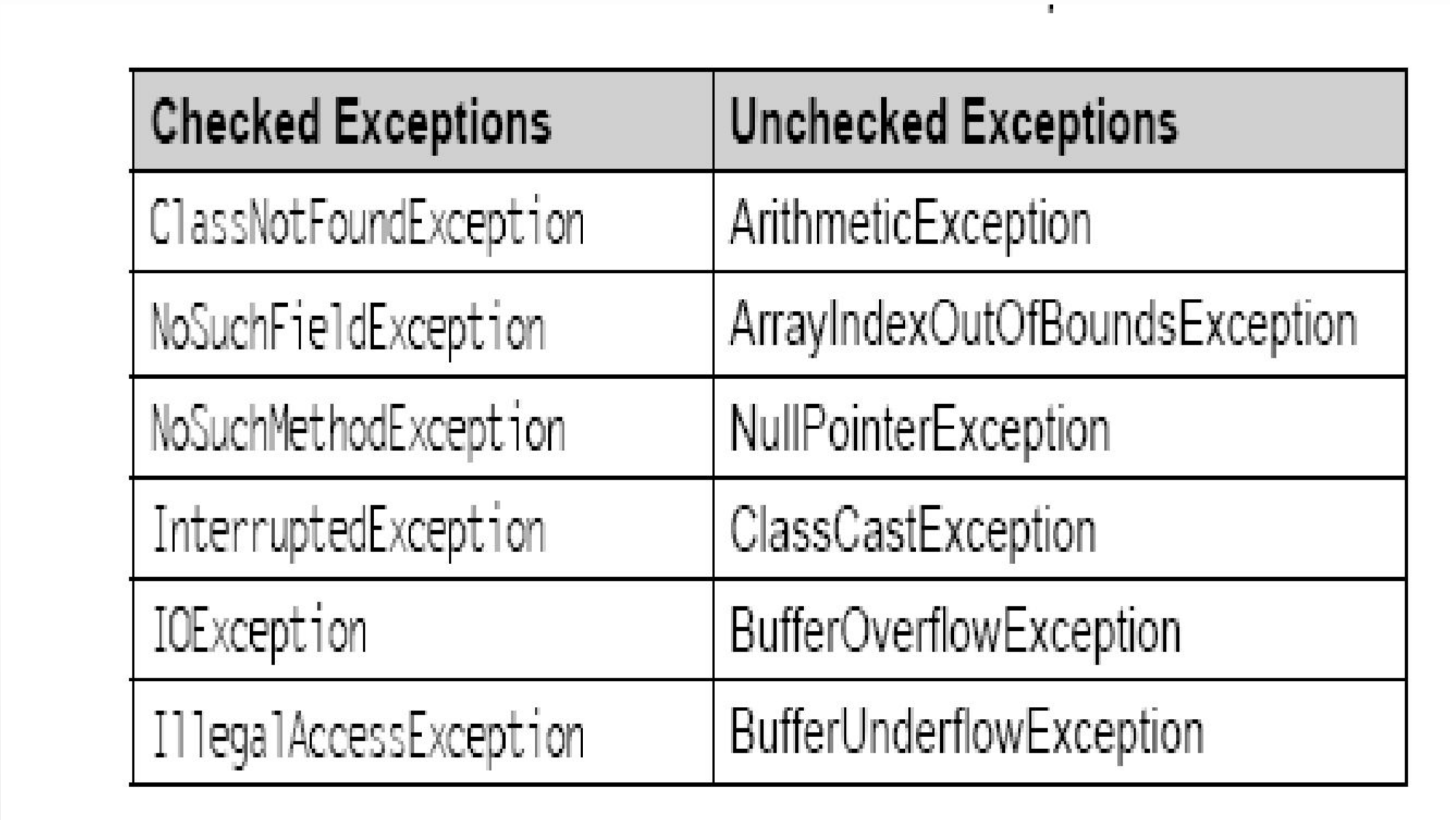


```
ExDemo.method3(ExDemo.java:23)
```



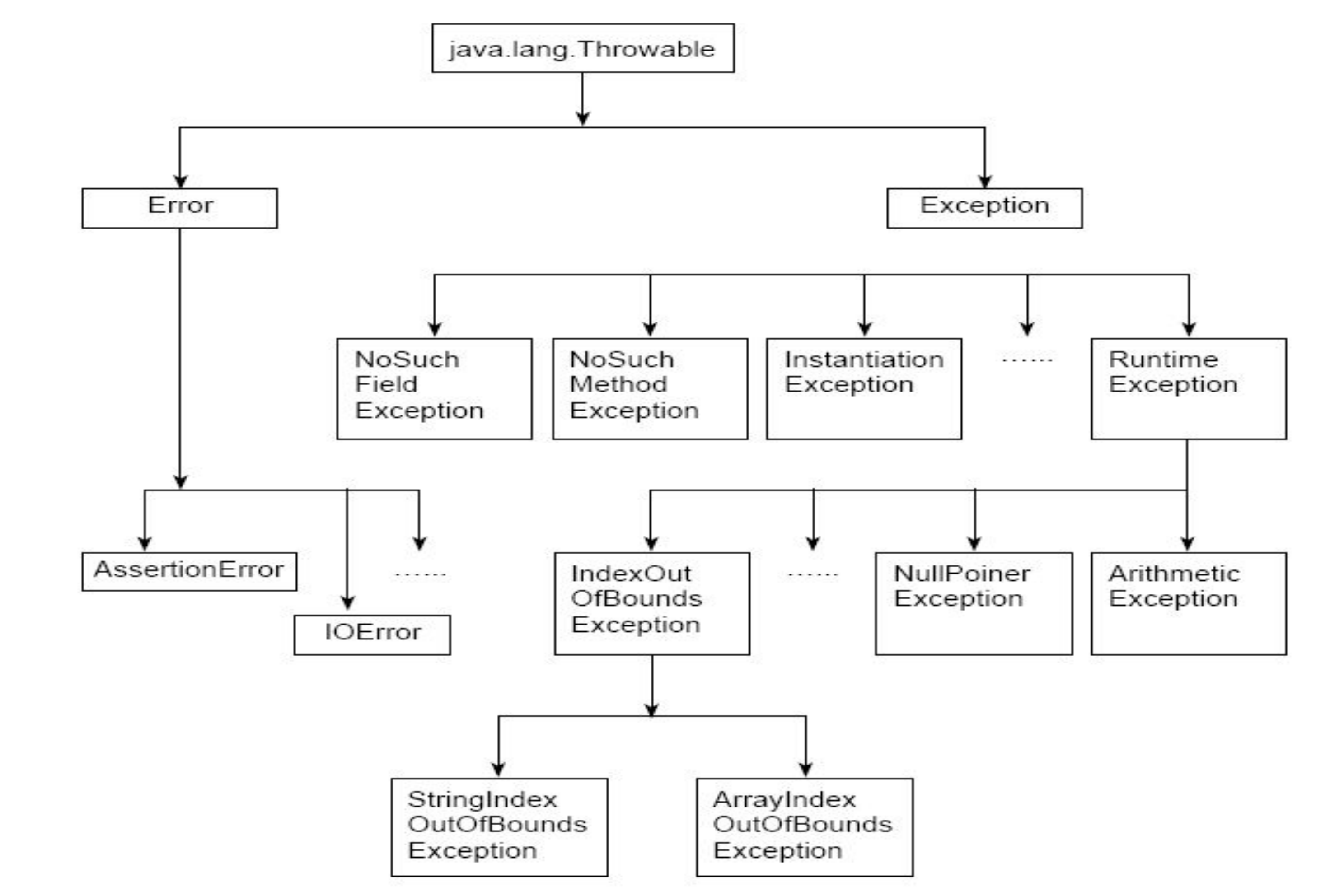
| |
|---|
| method3 |
| method2 |
| method1 |
| main |

# Types of Exceptions



| Checked Exceptions | Unchecked Exceptions |
|---|---|
| ClassNotFoundException | ArithmeticException |
| NoSuchFieldException | ArrayIndexOutOfBoundsException |
| NoSuchMethodException | NullPointerException |
| InterruptedException | ClassCastException |
| IOException | BufferOverflowException |
| IllegalAccessException | BufferUnderflowException |

# Exception Hierarchy



# Exception Hierarchy

- try…catch
- throw
- throws
- finally

# try…catch

- try/catch block can be placed within any method that you feel can throw exceptions.
- All the statements to be tried for exceptions are put in a try block.
- catch block is used to catch any exception raised from the try block.
- If exception occurs in any statement in the try block, control immediately passes to the corresponding catch block.

# Example

```
static void method2()
{
   System.out.println(“IN Method 2, Calling Method 3”);
   try{
         method3();
   }
   catch(Exception e)
   {
         System.out.println(“Exception Handled”);
   }
System.out.println(“Returned from method 3”);
}
```

# Multiple Catch Clauses

```
static void method2()
{
    System.out.println(“IN Method 2, Calling Method 3”);
    try{
        method3(); }
    catch(ArithmeticException ae)
    {
        System.out.println (“Arithmetic Exception Handled: “ +ae);
    }
    catch(Exception e)
    {
        System.out.println(“Exception Handled”);
    }
System.out.println(“Returned from method 3”);
}
```

**Note:** catch having super class types should be defined later than the catch clauses with subclass types. The order is important.

# Nested try...catch

```
try{ …..//statements
try{  …..//statements
}
catch(ArithmeticException ae){ . . . .}
…// statements
try{…//statements}
catch(ArrayIndexOutOfBoundsException ie){}
}
catch(Exception e){…..}
```

# throw Keyword

- Used to explicitly throw an exception
- Useful when we want to throw a user-defined exception
- The syntax for *throw* keyword is as follows:

  - throw new ThrowableInstance

  For example

  - throw new NullPointerException();

# throw Keyword

- Is added to the method signature to let the caller know about what exception the called method can throw.
- Responsibility of the caller to either handle the exception (using try…catch mechanism) or it can also pass the exception (by specifying throws clause in its method declaration).
- If all the methods in a program pass the exception to their callers (including main( )), then ultimately the exception passes to the default exception handler.

# throws Syntax

- A method can throw more than one exception; the exception list is specified as separated by commas.
- The syntax for the *throws* keyword is shown below:

```
public void divide(int a, int b) throws ArithmeticException,
 IllegalArgumentException
```

# Your turn

- What is Exception?
- What precaution to be taken while using multiple catch clauses?
- Can statements be placed after throw clause?
- What is the difference between throw and throws?

# finally

- **finally** block is executed in all circumstances
  - if the exception occurs or
  - it is normal return (using return keyword) from methods
- Mandatory to execute statements like related to release of resources, etc. can be put in a **finally** block.
- The syntax of the **finally** keyword is as follows:

```
try {……}
catch(Throwable e){……}
finally {……..}
```

# finally Example

```
class FinallyDemo{
    public static void main(String args[])
    {
        method1();
        System.out.println(“Result : “+method2 (24,0));
    }
    static void method1(){
    try{
        System.out.println(“IN Method 1”);
        throw new NullPointerException(); }
    catch(Exception e) {
        System.out.println(“Exception Handled: “ + e);      }
    finally {
        System.out.println(“In method 1 finally”);} }
    static int method2(int a,int b){
    try{
        System.out.println(“IN Method 2”);
        return a/b; }
    finally {
        System.out.println(“In method 2 finally”);} } }
```

# The Output

**When a=24 and b=4**

```
IN Method 1
Exception Handled:java.lang.NullPointerException
In method 1 finally
IN Method 2
In method 2 finally
Result : 6
```

**When a=24 and b=0**

```
IN Method 1
Exception Handled: java.lang.NullPointerException
In method 1 finally
IN Method 2
In method 2 finally
Exception in thread “main” java.lang.ArithmeticException: / by zero
at FinallyDemo.method2(FinallyDemo.java:24)
at FinallyDemo.main(FinallyDemo.java:6)
```

# try-with-resources Statement

- Java 7 added a new enhancement i.e., automatic resource management with a try-with-resources statement.
- Applications use many resources during their lifetime by creating their objects,
  - e.g., creating a database connection for accessing/updating databases
  - creating file objects for working with files
  - creating sockets for transmission/ receiving of data, etc.

# try-with-resources Statement

- A common mistake committed by programmers is that they often do not close/release the resources occupied by the programs, after their task is complete.
- This leads to many orphaned instances, inefficient memory allocation, and garbage collection.
- Hence the need for automatic resource management arises.

# try-with-resources Statement

- To address this problem AutoCloseable, a new interface, has been created in the java.lang package.
- The resources that want to be closed automatically after use must implement this interface.
- This interface has just one method,

  - public void close() throws Exception

# Syntax of try-with-resources Statement

- For example

  - try (abc a=new abc(); pqr p=new pqr())

  - { // statements within the block }

- More than one AutoCloseable resource can be used in try-with-resources statement separated by semicolon.

- abc and pqr objects should implement the AutoCloseable interface.

# Multi-catch

- Java 7 introduced the multi-catch statement to catch multiple exceptions using a single catch

```
try   {
            // statements
            }
            catch (Exception1 | Exception2 | Exception3 e)
            {     // statements      }
```

- Exception1, Exception2, and Exception3, belonging to different hierarchies, are handled in a single catch block.

# Benefits of Multi-catch

- Results in more efficient byte code as you have just one catch block (instead of more as in the above case).
- Moreover same treatment can be applied to exceptions of different hierarchies.
- A way of applying different treatment while using multi catch syntax is by using instanceof operator.

# Example

```
catch(ArithmeticException |
ArrayIndexOutOfBoundsException |
NumberFormatException e)
{
if(e instanceof ArithmeticException)
System.out.println("Arithmetic Exception Handled: " +e);
else if(e instanceof NumberFormatException)
System.out.println("Exception Handled: " +e);
else
System.out.println(e);   }
```

# Improved Exception Handling in Java 7

- A method can specify only those exceptions in the throws clause that have been specified in the catch clause while re-throwing exceptions from within catch block.
- But Java 7 onwards, throws can specify more refined exceptions to be rethrown.
- Suppose there are two user-defined exceptions - Exception1 and Exception2 - which can be rethrown from within the catch block of a method.

# Improved Exception Handling

```
class DemoException  {
void throwException(int a, int b) throws Exception1, Exception2 {
try {
if (a<b)
throw new Exception1();
else
throw new Exception2();
} catch (Exception e) {
throw e;     }}
public static void main(String args[]) throws
Exception1,Exception2 {
new DemoException().throwException(4,0);  }}
```

# Improved Exception Handling

- Prior to Java 7 only the exceptions specified in the catch block can be mentioned as argument to the throws keyword.

# User Defined Exception

```
class ExcepDemo extends Exception
{
ExcepDemo(String msg){
super(msg);    }
public String toString()
{
  return “Exception in thread \”main\”
        ExcepDemo Exception: “+getMessage();
} }
```

# Exception Encapsulation

- Also known as chaining.
- Wrapping a caught exception in a different exception and throwing the wrapped exception.
- If you pass all your exceptions, your top level method might have to deal with a lot of exceptions and declaring or handling exceptions in all the methods back is a tedious task.
- Wrapping is also used to abstract the details of implementation. You might not want your working details (including the exception that are thrown) to be known to others.

# Exception Encapsulation

```
try{
throw new InstantiationException();
} catch(InstantiationException t)
{  // wrapping InstantiationException in
        //ExcepDemo
throw new ExcepDemo(“Wrapped Instantiation Exception”,t);
}
```

# Problems and Solution

- It leads to long stack traces; one for each exception in the wrapping hierarchy.
- Secondly, due to wrapping, it becomes difficult to figure out the problem that led to exceptions.
- Solution: Exception Enrichment.

# Exception Enrichment

- You do not wrap exceptions but add information to the already thrown exception and rethrow it, which leads to a single stack trace.

# Example

```
class ExcepDemo extends Exception{
    String message;
    ExcepDemo(String msg)           {
    message=msg;                    }
    public String toString(){
        return “Exception in thread \”main\” ExcepDemo Exception: “+message};
    public void addInformation(String msg) {
        message+=msg;               }}
    class ExceptionEnrichmentDemo{
    static void testException() throws ExcepDemo{
        try{
                throw new ExcepDemo(“Testing User Defined Exception”);
        }
    catch(ExcepDemo e)   {
        e.addInformation(“\nexception was successfully enriched and re-thrown from catch”);
        throw e; }          }
    public static void main(String args[])      {
        try {
                testException();   }
    catch(ExcepDemo e){
        System.out.println(e);      }}}
```

# The Output

```
Exception in thread “main” ExcepDemo Exception: Testing User
Defined Exception.

Exception was successfully enriched and re-thrown from catch.
```

# Assertions

- To create reliable programs that are correct and robust.
- Assertions are boolean expressions that are used to test/validate the code.
- They are basically used during testing and development phases.
- Used by the programmers to be doubly sure about a particular condition, which they feel to be true.
- If you expect a number to be positive, negative, array/reference is not null, then you can check these conditions by asserting them.

# assert Example

- Assertions in Java are declared with the help of *assert* keyword as shown below:
  - assert expression1; // assert x > 0;

    Or
  - assert expression1: expression2; // assert x < 0: “Value Ok”;
- Assertions have to be enabled explicitly; they are disabled by default.
- Options of Java can be used to enable and disable assertions.
  - -ea enable assertions
  - -da disable assertions

# Assertion Example

```
class AssertDemo
{
static void check(int i)
{
   assert i>0: “Value must be positive”;
   System.out.println(“value fine “+i);
}
public static void main(String args[])
{
   check(Integer.parseInt(args[0]));
}}
```

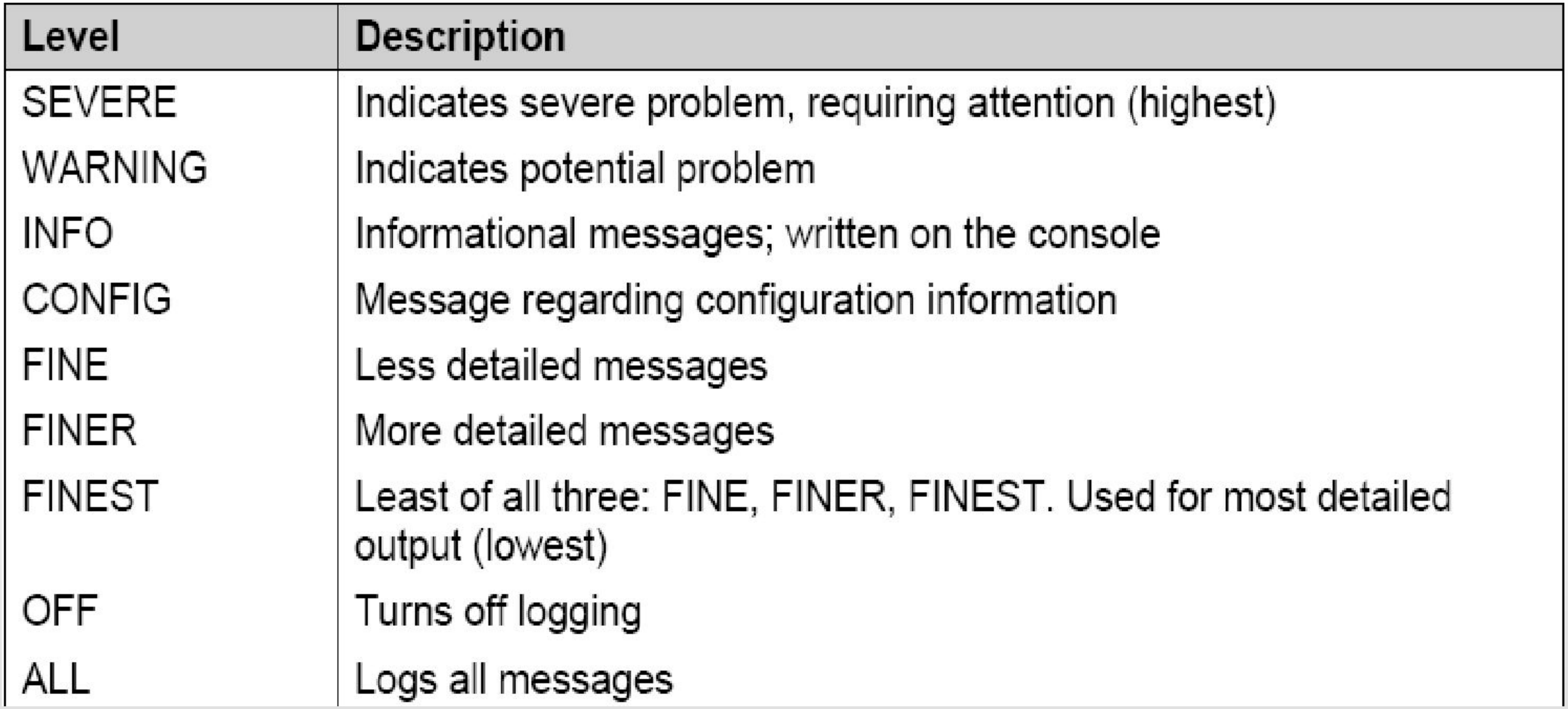
# The Output

- When i = 1
    - C:\javabook\programs\chap 7>java -ea AssertDemo 1
    - value fine 1
- When i = –1
    - C:\javabook\programs\chap 7>java -ea AssertDemo -1
    - Exception in thread “main” java.lang.AssertionError: Value must be positive
    - at AssertDemo.check(AssertDemo.java:5)
    - at AssertDemo.main(AssertDemo.java:10)
- Without enabling Assertion
    - C:\javabook\programs\chap 7>java AssertDemo -1
    - value fine -1

# Logging

- Added in the *java.util.logging* package.
- Logs are basically used to report messages regarding the functioning of the application to the programmer.
- Logs are created with the help of a *Logger* class.
- These messages are passed to handler objects which pass these messages to console, log files, etc.
- Logging has nine levels in Java to indicate the severity of logged messages.
- These levels are final and static fields of *Level* class (*util.logging* package).

# Levels of Logging



| Level | Description |
|---|---|
| SEVERE | Indicates severe problem, requiring attention (highest) |
| WARNING | Indicates potential problem |
| INFO | Informational messages; written on the console |
| CONFIG | Message regarding configuration information |
| FINE | Less detailed messages |
| FINER | More detailed messages |
| FINEST | Least of all three: FINE, FINER, FINEST. Used for most detailed output (lowest) |
| OFF | Turns off logging |
| ALL | Logs all messages |

# Methods of Logger Class

- The Logger class provides methods similar to the names of the levels for logging messages.
    - public void severe(String msg)
    - public void warning(String msg)
    - public void config(String msg)
    - public void info(String msg)
    - public void finest(String msg)
    - public void finer(String msg)
    - public void fine(String msg)
- It also provides a method which sets the level  as well as prints the message on the console.
    - public void log(Level l,String msg)

# Example

```
import java.util.logging.*;
class LoggingDemo {
static Logger l = Logger.getLogger(“LoggingDemo”);
void demo() {
l.log(Level.SEVERE,”Shows Severe level of the Logger “);
}
public static void main(String[] args) {
LoggingDemo d = new LoggingDemo();
d.demo();
}}
```

# The Output

- C:\javabook\programs\chap 7>java LoggingDemo
  - 22 Feb, 2009 11:18:49 AM LoggingDemo demo
  - SEVERE: Shows Severe level of the Logger

# Summary

- This chapter focuses on how to handle unusual conditions /situations in Java.
- Two types of exceptions have been defined: Checked and Unchecked.
- Five keywords in exception handling, namely try, catch, throw, throws, and finally.
- Apart from using the predefined exceptions, you can code your own exceptions according to your own requirements.

# Summary (contd.)

- Improvements in Java 7 are automatic resources management, multi-catch, etc.
- Assertions (introduced in JDK 1.4) are helpful in assuring the programmer about a particular condition using assert keyword.
- They help in increasing the reliability of a Java program.
- Logging features (part of java.util.logging package introduced in JDK 1.4) help the user to debug his program and can exactly pinpoint the errors in his/her program.