

Java Programming I

CHAPTER 09

Exceptions Part 2

Contents

- ◆ Exception Handling in Java
- ◆ Catching Exceptions
- ◆ The throws Clause
- ◆ Statement 'throw'
- ◆ Exception Chaining
- ◆ Exception Handling
- ◆ User Defined Exception Types
- ◆ RuntimeException

Exception Handling in Java

- ◆ Exception is the standard way of Java for error handling.
- ◆ Whenever there is a problem in a method, it throws an exception.

Exception Handling in Java

◆ Some Examples:

- **FileInputStream**(File file) throws FileNotFoundException
- int **read** () throws IOException
- final Object **readObject** () throws OptionalDataException, ClassNotFoundException, IOException
- static Connection **getConnection**(String url) throws SQLException
- protected **UnicastRemoteObject** () throws RemoteException

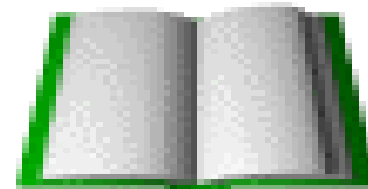
Catching Exceptions

- ◆ If an exception is to be thrown, something has to catch it.
- ◆ It may be caught either by the immediate caller, or somewhere up the calling stack.
- ◆ If the exception is not caught anywhere, the application terminates with a corresponding message.

Catching Exceptions

You would have the try block, followed by the catch block (actually, you can have more than one catch block, one for every possible exception).

```
try {  
    ...  
} catch (ExceptionType1 e1) {  
    ...  
} catch (ExceptionType2 e2) {  
    ...  
}
```



One of the obvious advantages of this clause, is the isolation of the code and the error handling code.

Catching Exceptions

To extend the usage a bit, you can define a special block, which would be finally called in any case, after the exception had been caught. The finally block, allows the programmer to define what will happen in any case (such as releasing resources, closing files).

```
try {  
    ...  
} catch (ExceptionType1 e1) {  
    ...  
} catch (ExceptionType2 e2) {  
    ...  
} finally {  
    ...  
}
```

Catching Exceptions

- ◆ In the catch block we may:
 - Show a warning message and return
 - Exit the program
 - Re-throw the exception
 - Throw other exception
 - Try to solve the problem

Catching Exceptions

◆ Example:

```
1. void readFile (String fileName) throws FileEmpty, FormatException {
2.     Vector v = new Vector ( ) ;
3.     try {
4.         RecordsFile rf = new RecordsFile (fileName) ;
5.         Header h = rf.readHeader ( ) ;
6.         while ( !rf.eof ( ) ) {
7.             Record rec = rf.readNextRecord ( ) ;
8.             v.add(rec) ;
9.         }
10.    } catch (FileEmpty e) {
11.        rf.close ( ) ; // error handling
12.        throw e; // re-throw
13.    } catch (UnexpectedCharacter e) {
14.        // catch exception and throw another exception
15.        throw new FormatException (fileName + " Format error") ;
16.    }
17. }
```

Catching Exceptions

- ◆ Line 10-11: Method `readFile` catches the exception and handles it partially. Then it is thrown to the caller
- ◆ Line 15: The method catches an exception and throws another one

Catching Exceptions

```
1.  try {
2.      // trying to load driver
3.      Class.forName ("yuhu.drivers.CoolDriver");
4.  } catch (ClassNotFoundException ex) {
5.      try {
6.          // trying another driver
7.          Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver") ;
8.      } catch (ClassNotFoundException e) {
9.          // bad luck, quit
10.         System.err.println ("Failed to load driver: " + e) ;
11.         System.exit (-1) ;
12.     }
13. }
```

◆ Line 5: A try block inside a catch block

The throws Clause

- ◆ The checked exceptions that a method throws are as important as the type of value it returns. Both must be declared.
- ◆ If you invoke a method that lists a checked exception in its throws clause, you have three choices:
 - Catch the exception and handle it.
 - Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own throws clause.
 - Declare the exception in your throws clause and let the exception pass through your method (although you might have a finally clause that cleans up first) .
- ◆ Throws clauses and Method Overriding: An overriding or implementing method is not allowed to declare more checked exceptions in the throws clause than the inherited method does.

The throws Clause

- ◆ You would need to use the keyword **throws** when declaring your method so as to specify that your method may throw an exception and its type

```
ReturnType myMethod (. . . ) throws  
    ExceptionType { . . . }
```

- ◆ When throwing an exception, you must decide which type of exception to throw

Example (1 of 2)

```
1 import java.io.*;
2 public class AgeTest {
3     String line;
4     static InputStreamReader isr = new InputStreamReader(System.in);
5     static BufferedReader in = new BufferedReader(isr);
6     public void tryMethod()throws IOException {
7         int age=0;
8         boolean ok = false;
9         while (!ok) {
10             try {
11                 System.out.print("Enter your age : ");
12                 line = in.readLine();
13                 age = Integer.parseInt(line);
14                 ok = true;
15             }
16             catch(NumberFormatException nfe){
17                 System.err.println("Integer value required");
18             }
19         }
20         System.out.printf("Your age is: %d \n", age);
21     }
```

Example (2 of 2)

```
22 public void useThrowsMethod() throws IOException, NumberFormatException {
23     int height =0;
24     System.out.print("Enter your height : ");
25     line = in.readLine();
26     height = Integer.parseInt(line);
27     System.out.printf("Your height is: %d \n", height);
28 }
29 public static void main(String [] args) throws IOException,NumberFormatException{
30     AgeTest obj= new AgeTest();
31     try {
32         obj.tryMethod();
33         obj.useThrowsMethod();
34     } catch(NumberFormatException nfe){
35         System.err.println("Integer value required");
36     } finally {
37         in.close();
38     }
39 }
40 }
```

Comments on the example

- ◆ If the exception occurs inside the **tryMethod()** method (by lines 12 and 13), it will be caught and handled inside the method itself (lines 16-18).
- ◆ If the exception occurs inside the **useThrowsMethod()** method (by lines 25 and 26), since there is no catcher and handler in this method, it will be **thrown** to the caller method (the **main()** method) and will be caught and handled there (lines 34-35).
- ◆ The **finally** clause will be implemented in any case and will close the opened reading streams (we study streams in IO chapter)
- ◆ You can simulate the **NumberFormatException** by trying to input a non integer value (for example try to input a character such as 'A', 'B', etc.)

How to Throw Exceptions

- ◆ Before we can catch an exception, some code somewhere must throw one. Regardless of what throws the exception, it's always thrown with the throw statement.
- ◆ The throw Statement
throw someThrowableObject;

```
public Object pop () {  
    Object obj;  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
    obj = objectAt(size - 1);  
    setObjectAt (size - 1, null);  
    size--;  
    return obj;  
}
```

If the stack is empty, pop instantiates a new EmptyStackException object and throws it.

Statement 'throw'

- ◆ You can throw exceptions by yourself as part of your code using the keyword **throw**

```
if (someCondition)  
    throw new ExceptionType(. . . ) ;
```

Statement 'throw'

Throw a **user created** exception

◆ Throw Statement *throw expression;*

```
public class  
    NoSuchAttributeException  
    extends Exception  
{  
    public final String attrName;  
  
    public NoSuchAttributeException  
        (String name) {  
        super("No attribute named \" +  
            name + "\" found");  
        attrName = name;  
    }  
}
```

```
public void replaceValue(String name,  
    Object newValue)  
    throws NoSuchAttributeException  
{  
    Attribute attr = find(name);  
    if (attr == null)  
        throw new  
            NoSuchAttributeException(name);  
    attr.setValue(newValue);  
}
```

Statement 'throw'

```
class ThrowDemo {  
  
    public static void main(String args[])  
    {  
        try {  
            System.out.println("Before a");  
            a();  
            System.out.println("After a");  
        }  
        catch (ArithmeticException e) {  
            System.out.println("main: " + e);  
        }  
        finally {  
            System.out.println("main: finally");  
        }  
    }  
  
    public static void a() {  
        try {  
            System.out.println("Before b");  
            b();  
            System.out.println("After b");  
        }  
        catch (ArithmeticException e) {  
            System.out.println("a: " + e);  
        }  
        finally {  
            System.out.println("a: finally");  
        }  
    }  
}
```

```
    public static void b() {  
        try {  
            System.out.println("Before c");  
            c();  
            System.out.println("After c");  
        }  
        catch (ArithmeticException e) {  
            System.out.println("b: " + e);  
        }  
        finally {  
            System.out.println("b: finally");  
        }  
    }  
  
    public static void c() {  
        try {  
            System.out.println("Before d");  
            d();  
            System.out.println("After d");  
        }  
        catch (ArithmeticException e) {  
            System.out.println("c: " + e);  
            throw e;  
        }  
        finally {  
            System.out.println("c: finally");  
        }  
    }  
}
```

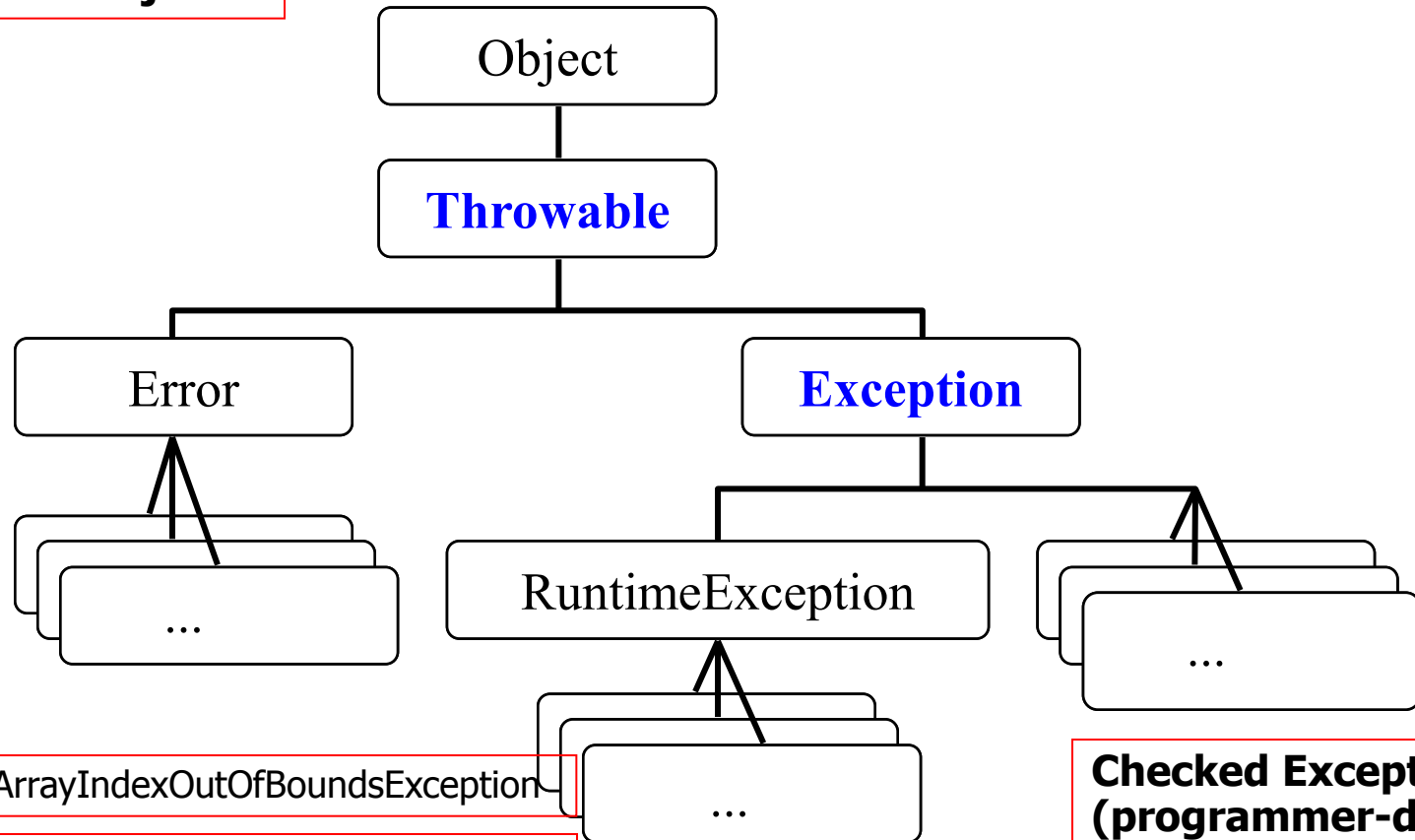
```
    public static void d() {  
        try {  
            int i = 1;  
            int j = 0;  
            System.out.println("Before division");  
            System.out.println(i / j);  
            System.out.println("After division");  
        }  
        catch (ArithmeticException e) {  
            System.out.println("d: " + e);  
            throw e;  
        }  
        finally {  
            System.out.println("d: finally");  
        }  
    }  
}
```

Result :

Before a
Before b
Before c
Before d
Before division
d: java.lang.ArithmeticException: / by zero
d: finally
c: java.lang.ArithmeticException: / by zero
c: finally
b: java.lang.ArithmeticException: / by zero
b: finally
After b
a: finally
After a
main: finally

Throwable Class and Its Subclasses

Exceptions are objects.



`(ex)ArrayIndexOutOfBoundsException`

**Unchecked Runtime Exceptions
(system-defined exception):**

Reflect errors in program's logic

- Error and RuntimeException

**Checked Exceptions
(programmer-defined exceptions):**

The compiler checks

Exception Chaining

Exceptions caused by other exceptions

```
public double[] getDataSet(String setName) throws BadDataSetException
```

```
{
    String file = setName + ".dset";
    FileInputStream in = null;
    try {
        in = new FileInputStream(file);
        return readDataSet(in);
    } catch (IOException e) {
        throw new BadDataSetException();
    } finally {
        try {
            if ( in != null ) in.close();
        } catch (IOException e) {
            ; // ignore: we either read the data OK
            // or we're throwing BadDataSetException
        }
    }
}
// ... definition of readDataSet
```

For the notion of one exception being caused by another exception

```
catch (IOException e) {
    BadDataSetException bdse =
        new BadDataSetException();
    bdse.initCause(e);
    throw bdse;
} finally {
    // .....
}
```

initCause is used to remember the exception that made the data bad. Later, the **getCause** method is used to retrieve the exception.

Exception Chaining

Another Way: to define new exception class

```
class BadDatasetException extends Exception {  
    public BadDatasetException() {}
```

```
    public BadDatasetException(String details) {  
        super(details);  
    }
```

```
    public BadDatasetException(Throwable cause) {  
        super(cause);  
    }
```

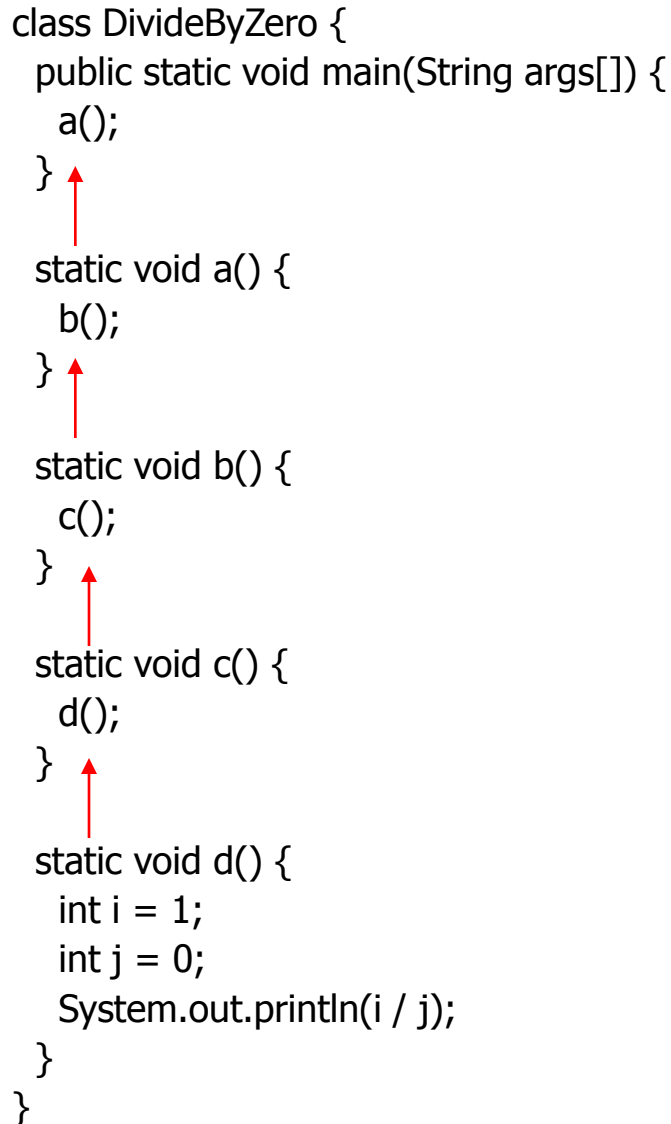
```
    public BadDatasetException(String details,  
        Throwable cause) {  
        super(details, cause);  
    }  
}
```

```
    } catch (IOException e) {  
        throw  
            new BadDatasetException(e);  
    } finally {  
        // ...  
    }
```

Now you can write like this!

Exception Handling

```
class DivideByZero {  
    public static void main(String args[]) {  
        a();  
    }  
    static void a() {  
        b();  
    }  
    static void b() {  
        c();  
    }  
    static void c() {  
        d();  
    }  
    static void d() {  
        int i = 1;  
        int j = 0;  
        System.out.println(i / j);  
    }  
}
```



Result :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at DivideByZero.d(DivideByZero.java:22)  
    at DivideByZero.c(DivideByZero.java:16)  
    at DivideByZero.b(DivideByZero.java:12)  
    at DivideByZero.a(DivideByZero.java:8)  
    at DivideByZero.main(DivideByZero.java:4)
```


Exception Handling

```
class Divider {  
    public static void main(String args[]) {  
        try {  
            System.out.println("Before Division");  
            int i = Integer.parseInt(args[0]);  
            int j = Integer.parseInt(args[1]);  
            System.out.println(i / j);  
            System.out.println("After Division");  
        }  
        catch (ArithmeticException e) {  
            System.out.println("ArithmeticException");  
        }  
        catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("ArrayIndex" +  
                "OutOfBoundsException");  
        }  
        catch (NumberFormatException e) {  
            System.out.println("NumberFormatException");  
        }  
        finally {  
            System.out.println("Finally block");  
        }  
    }  
}
```

No arguments, so no args[0], args[1]

Result: java Divider

Before Division

ArrayIndexOutOfBoundsException

Finally block

Result: java Divider 1 0

Before Division

ArithmeticException

Finally block

Exception Handling

```
class CatchSearch {
{
    public static void main(String args[])
    {
        try {
            System.out.println("Before a");
            a();
            System.out.println("After a");
        }
        catch (Exception e) {
            System.out.println("main: " + e);
        }
        finally {
            System.out.println("main: finally");
        }
    }

    public static void a() {
        try {
            System.out.println("Before b");
            b();
            System.out.println("After b");
        }
        catch (ArithmeticException e) {
            System.out.println("a: " + e);
        }
        finally {
            System.out.println("a: finally");
        }
    }
}
```

After handle the exception, it is reached here.

```
public static void b() {
    try {
        System.out.println("Before c");
        c();
        System.out.println("After c");
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("b: " + e);
    }
    finally {
        System.out.println("b: finally");
    }
}

public static void c() {
    try {
        System.out.println("Before d");
        d();
        System.out.println("After d");
    }
    catch (NumberFormatException e) {
        System.out.println("c: " + e);
    }
    finally {
        System.out.println("c: finally");
    }
}
```

```
public static void d() {
    try {
        int array[] = new int[4];
        array[10] = 10;
    }
    catch (ClassCastException e) {
        System.out.println("d: " + e);
    }
    finally {
        System.out.println("d: finally");
    }
}
```

Result :

Before a
Before b
Before c
Before d
d: finally
c: finally
b: java.lang.ArrayIndexOutOfBoundsException: 10
b: finally
After b
a: finally
After a
main: finally

Choosing a Super or Sub class

- ◆ We cannot put a superclass catch clause before a catch of one of its subclasses.

```
Class SuperException extends Exception {}
Class SubException extends SuperException {}
Class BadCatch {
    public void goodTry() {
        /* This is an INVALID catch ordering */
        try {
            throw new SubException();
        } catch (SuperException superRef) {
            // Catches both SuperException and SubException
        } catch (SubException subRef) {
            // This would never be reached
        }
    }
}
```

User Defined Exception Types

◆ Why Create New Exception Type:

- Adding useful data
- Type of the exception is important part of the exception data.

<Example>

```
public class
    NoSuchAttributeException
    extends Exception
{
    public final String attrName;

    public NoSuchAttributeException
        (String name) {
        super("No attribute named \" +
            name + "\" found");
        attrName = name;
    }
}
```

Another Example Using Constructors

```
public class BadDataSetException
    extends Exception
{
    public BadDataSetException() {}

    public BadDataSetException(String s)
        { super(s);}

    public
        BadDataSetException(Throwable
            cause) { super(cause);}

    public BadDataSetException(String s,
        Throwable cause) { super(s,
            cause);}
}
```

User Defined Exception Types

Subclass of Exception

```
import java.util.*;

class ExceptionSubclass {

    public static void main(String args[]) {
        a();
    }

    static void a() {
        try {
            b();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void b() throws ExceptionA {
        try {
            c();
        }
        catch (ExceptionB e) {
            e.printStackTrace();
        }
    }
}
```

In the "b" method, the ExceptionB is to be handled, but pass the ExceptionA, which will be caught in the method a().

```
static void c() throws ExceptionA, ExceptionB {
    Random random = new Random();
    int i = random.nextInt();
    if (i % 2 == 0) {
        throw new ExceptionA("We have a problem");
    }
    else {
        throw new ExceptionB("We have a big problem");
    }
}

class ExceptionA extends Exception {
    public ExceptionA(String message) {
        super(message);
    }
}

class ExceptionB extends Exception {
    public ExceptionB(String message) {
        super(message);
    }
}
```

Execution:

ExceptionA: We have a problem

at ExceptionSubclass.c(ExceptionSubclass.java:31)
at ExceptionSubclass.b(ExceptionSubclass.java:20)
at ExceptionSubclass.a(ExceptionSubclass.java:11)
at ExceptionSubclass.main(ExceptionSubclass.java:6)

Exception Vs. Error

- ◆ An **Error** indicates a serious problem that a reasonable application should not try to catch.
- ◆ Examples:
 - `ClassFormatError`,
 - `InstantiationError`,
 - `InternalError`,
 - `NoSuchMethodError`,
 - `OutOfMemoryError`,
 - `StackOverflowError`,
 - `VirtualMachineError`

RuntimeException

- ◆ **RuntimeException** is the superclass of those exceptions that can be thrown during the normal operation of the JVM.
- ◆ Examples:
 - NullPointerException,
 - ArrayIndexOutOfBoundsException,
 - NegativeArraySizeException,
 - ClassCastException,
 - NumberFormatException,
 - SecurityException.

RuntimeException

- ◆ The only exception which the Java language does not require to be caught, when thrown, are those of type **RuntimeException**.
- ◆ Exception of type **RuntimeException** are meant to be dealt with by the Java virtual machine. Therefore, although tempting, you must not throw exceptions of type **RuntimeException** unless there is a good reason to do so.

Summary

- ◆ Error Handling is a safe and convenient way to deal with problems in a program.
- ◆ Exceptions are Java's tool for error handling.
- ◆ Exceptions allows us to separate the normal flow of the program from the error handling code.
- ◆ The compiler checks that all exceptions are caught.
- ◆ If the exception is not caught anywhere, the application terminates with a corresponding message.