ITI 1121. Introduction to Computing II

Error handling in Java

by

Marcel Turcotte

Version February 24, 2020

Summary

Error handling in Java

Modern programming languages offer mechanisms for error handling. In Java, we will see that an error situation is modeled using an object. We will see how to signify an error and understand the consequences on the flow of control. Finally, we will look at two ways to handle errors.

General objective :

- Following this lesson, you will be able to report and handle errors in Java.

Learning objectives

- **Name** some types of Java exceptions.
- **Trace** the execution of a program following the execution of a throw statement.
- **Explain** the following statement: Exceptions are either checked or unchecked.
- **Modify** an application to signify errors using exceptions.
- **Modify** an application to handle error situations.
- **Create** new types of exceptions.

Readings:

- Pages 29–36, 559, 608–619 of E. Koffman and P. Wolfgang.

Plan

Error handling

Theory : Error handling

Objectifs d'apprentissage :

- **Distinguish** compilation errors from runtime errors
- **Develop** preconditions for a class method
- **Develop** preconditions for an instance method

Error handling

Introduction

Computer programming disasters

- **Give** examples of programming errors that have led to disasters.

Self-driving cars

Source: Grendelkhan

Compilation errors and execution errors

There are **two** types of errors: **compilation** errors and **execution** errors.

Compilation :

- **Syntax** errors
- Java being a **strongly typed** language, the compiler also checks the type of each expression, which allows the detection of some errors as soon as possible, **before the execution of the program**. Type checking ensures that operations on a value are valid for the type of the value.

Compilation errors don't affect the users!

Discussion : Runtime errors

Give examples of run-time errors!

Discussion : Sources of runtime errors

Name the sources of the run-time errors!

*As a consequence, we will see that Java also offers us two ways to deal with error situations.

Error handling

Preconditions

Preconditions

- A **precondition** is a **condition that should be met** before a method is executed.
- In object-oriented programming, we need to validate not only the **parameter values**, but also the **state of the object**.

A **method** must begin with the **validation** of the **preconditions**!

Error handling : What to do?

Desired features

Detecting and handling error situations helps make programs more **robust**.

- **Indicate** the source of the error precisely.
    - What method? What statement? What is the nature of the error?
- **Force** the software to take action to correct the situation. (impossible to ignore errors)

Exception

Exception

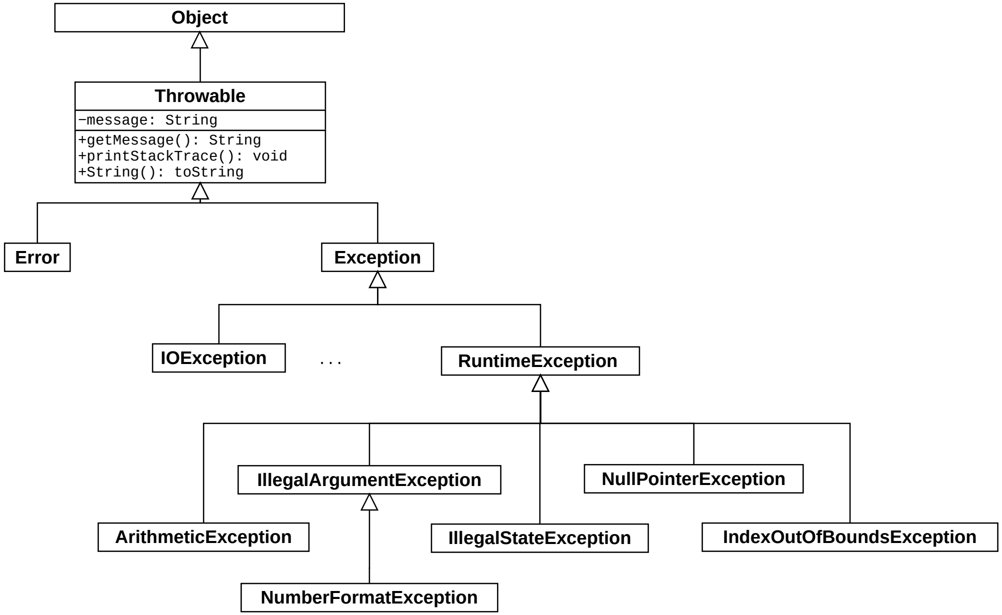
Learning objectives :

- **Summarize** the role of the class **Exception**

Exception

The class Exception

Exception is a class!



Exception is a class!

- Exceptions are **objects!**
- An error situation is modeled using an object of the class **Throwable**, or one of its subclasses.
    - The object encapsulates an error message.
- Among others, the class **Throwable** declares the methods **String getMessage()** and **void printStackTrace()**.

Exception

Variable of type Exception

Declaring a variable of type Exception

Declaring a reference of type **Exception** is not exceptional.

```
Exception e;
```

Exception

Object from the class Exception

Creating an object of the class Exception

Likewise, there's nothing exceptional about creating an object of the class **Exception**.

```
e = new Exception("Houston, we've had a problem!");
```

The line "Houston, we've had a problem" became famous after the movie "Apollo 13".

Throw

Signaling an error situation

Learning objectives:

- **Modifying** an application to signify errors using exceptions.
- **Trace** the execution of a program following the execution of a throw statement.

Readings:

- Pages 559, 608–619 of E. Koffman and P. Wolfgang.

Signaling an error situation

- Let’s consider the implementation of a stack using linked elements and its method **pop()**.
- Removing an element when the stack is empty is an error situation.
- Formulate a test to validate the precondition of the **pop** method.

Throw

Throw statement

Throw statement

In Java, the statement “**throw**” alters the normal flow of control. Its argument is a reference to an object of the class **Throwable** or one of its subclasses.

Throw statement

Throw

Transfer of control

Transfer of control

When an exceptional situation is reported,

- The statement or expression **ends abruptly**;
- If the exception is not processed, **the stack of method calls will be completely unwound**, i.e. each method call on the execution stack will end abruptly, and the execution of the program will end with the printing of the stack at the time of the error (“stack trace”);
- **No statements** and **no parts of the expression** after the expression that caused the error **will be executed**;
- **following an exception**, the next statements executed are those in a **catch** or **finally** block.

Throw

Examples

```java
class Test {
    public static void main( String[] args ) {
        System.out.println( "Label 1" );
        if (args.length == 0) {
            throw new RuntimeException("Houston...");
        }
        System.out.println( "Label 2" );
    }
}
```

Houston, we’ve had a problem!

- Following the statement **throw**, the method **terminates** abruptly.

```
> java Test
Label 1
Exception in thread "main" java.lang.RuntimeException:
Houston, we’ve had a problem!
        at Test.main(Test.java:5)
```

- Thus, the string “Label 2” **will not be displayed** on the console.

```java
public class Test {
    public static boolean error(int v) {
        if (v == 0) {
            throw new RuntimeException("Oops! I'm sorry.");
        }
        return true;
    }
    public static boolean display() {
        System.out.println("Label 2");
        return true;
    }
    public static void main( String[] args ) {
        System.out.println("Label 1" );
        if (error(0) || display()) {
            System.out.println("Label 3");
        }
        System.out.println("Label 4");
    }
}
```

- Compile and run the above program.

Oops! I’m sorry

- Following the **throw** statement, the method **error terminates** abruptly, likewise, the method **main terminates** abruptly.

```
> java Test
Label 1
Exception in thread "main" java.lang.RuntimeException:
Oops! I’m sorry.
        at Test.error(Test.java:5)
        at Test.main(Test.java:16)
```

- Thus, the strings “Label 2”, “Label 3” and “Label 4” **will not be displayed** on the console.

```java
public class Test {
    public static void c() {
        System.out.println("c: Label 1" );
        if (true) {
            throw new RuntimeException("dessus de la pile");
        }
        System.out.println("c: Label 2");
    }
    public static void b() {
        System.out.println("b: Label 1");
        c();
        System.out.println("b: Label 2");
    }
    public static void a() {
        System.out.println("a: Label 1");
        b();
        System.out.println("a: Label 2");
    }
    public static void main(String[] args) {
        System.out.println("main: Label 1");
        a();
        System.out.println("main: Label 2");
    }
}
```

Unwind the call stack

```
> java Test
main: Label 1
a: Label 1
b: Label 1
c: Label 1
Exception in thread "main" java.lang.RuntimeException:
dessus de la pile des appels
        at Test.c(Test.java:6)
        at Test.b(Test.java:11)
        at Test.a(Test.java:16)
        at Test.main(Test.java:21)
```

```
public E pop() {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    E saved;
    saved = elems[--top];
    elems[top] == null;
    return saved;
}
```

- **If** the **precondition** is not satisfied, the method signals an error situation.
- **Otherwise**, the method **removes** the first item in the stack, and **returns** its value.

Try-Catch

Theory

Learning objectives :

- **Modifying** an application to handle error situations.
- **Tracing** the execution of a program following the execution of a throw statement, but in the presence of try-catch statements.

Readings:

- Pages 559, 608–619 of E. Koffman and P. Wolfgang.

Try-Catch

Syntax

Handling Exceptions

The bloc **try/catch** is used to regain control when an error situation has occurred.

```
try {
    // ...
} catch (ExceptionType1 id1) {
    // statements;
} catch (ExceptionType2 id2) {
    // statements;
} finally {
    // statements;
}
```

If no exceptions are thrown, only the statements in the **try** block and the **finally** block will be executed.

Try-Catch

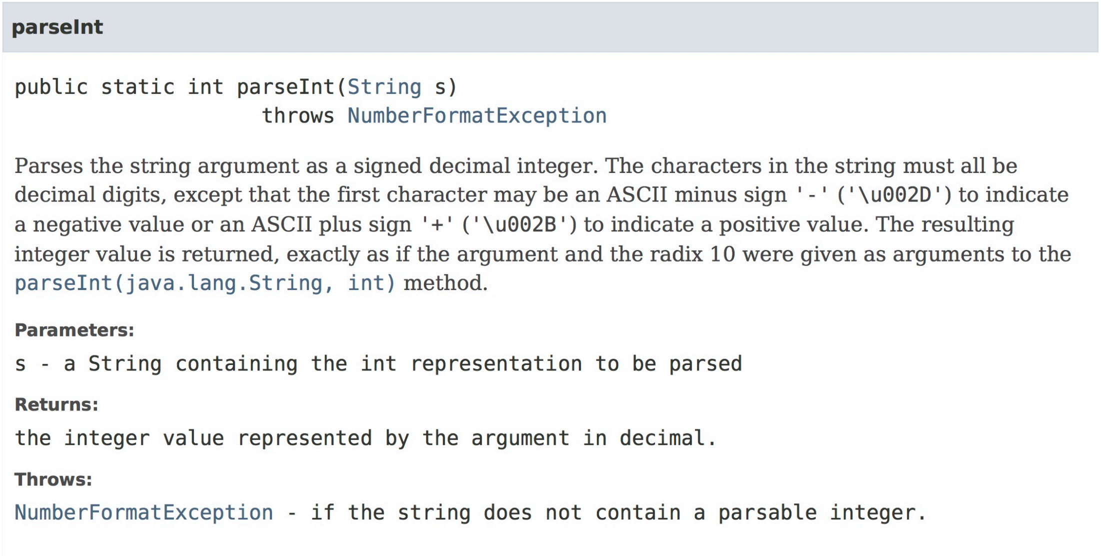
Example

```java
public class Grill {
    private Burner burner = new Burner();
    public void cooking() {
        try {
            burner.on();
            addSteak();
            addSaltAndPepper();
            boolean done = false;
            while (!done) {
                done = checkSteak();
            }
        } catch (OutOfGazException e1) {
            callRetailer();
        } catch (FireException e2) {
            extinguishFire();
        } finally {
            burner.off();
        }
    }
}
```

Example: try/catch

```
int DEFAULT_VALUE = 0;
int value;

try {

    value = Integer.parseInt("100");

} catch (NumberFormatException e) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

How do we know what kind of exception might be thrown?



parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a `String` containing the `int` representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Example: try/catch

```
int DEFAULT_VALUE = 0;
int value;

try {

    value = Integer.parseInt("cent");

} catch (NumberFormatException e) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

Try-Catch

Flow of control

Flow of control

- When an exception is thrown, the execution of the statements of the block **try** ends (abruptly) and continues with the statements of the first block **catch** whose parameter is of the same type as that of the object modeling the error situation, or of a more general type, followed by the execution of the statements of the block **finally**, if present.
- No other blocks will be executed.
- If no **catch** block is appropriate, then the exception percolates.
- Statements in the **finally** block are always executed: with or without error.
  - Finally blocks are used to close open files, for example, or in general, to deal with post conditions.

Try-Catch

Comprehensive example

```java
public class Test {
    public static void c() {
        System.out.println( "c() :: about to throw exception" );
        throw new RuntimeException( "from c()" );
    }
    public static void b() {
        System.out.println( "b() :: pre-" );
        c();
        System.out.println( "b() :: post-" );
    }
    public static void a() {
        System.out.println( "a() :: pre-" );
        try {
            b();
        } catch ( RuntimeException e ) {
            System.out.println( "a() :: caught exception" );
        }
        System.out.println( "a() :: calling b, no try block" );
        b();
        System.out.println( "a() :: post-" );
    }
    public static void main( String[] args ) {
        System.out.println( "main( ... ) :: pre-" );
        a();
        System.out.println( "main( ... ) :: post-" );
    }
}
```

Try-Catch

The reference of type Exception

About the parameter of the block catch

```
int DEFAULT_VALUE = 0;
int value;

try {
    value = Integer.parseInt("douze");
} catch (NumberFormatException e) {
    System.out.println("warning: " + e.getMessage());
    value = DEFAULT_VALUE;
}
```

- The parameter **e** is a reference designating the object passed to the statement **throw**, as for any other object, the dot notation is used to access the methods of the object.

Throws

Theory

Learning objectives:

- **Explain** the following statement: Exceptions are either checked or un-checked.

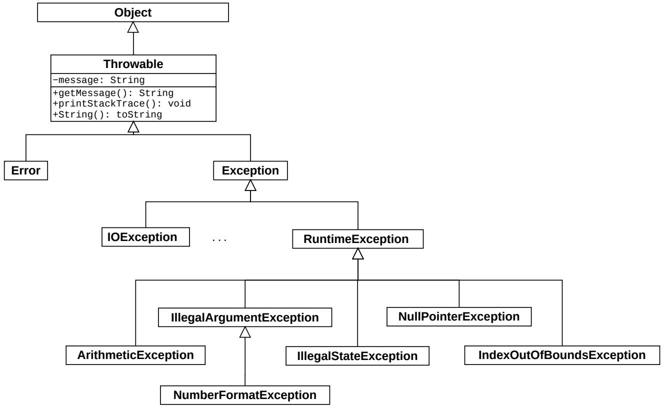
Lectures:

- Pages 559, 608–619 of E. Koffman and P. Wolfgang.

Throws

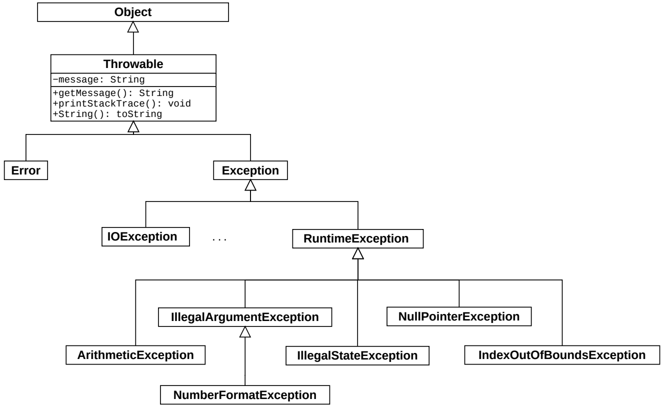
Mandatory declaration or not

Checked or unchecked



- The descendants of the class **Exception** are **checked**.

Checked or unchecked



- **Except** if they are subclasses of the class **RuntimeExcetion**, then they are **unchecked**.

Discussion: Sources of runtime errors

Checked exceptions

A method using a method that can throw a **checked exception** shall:

- Handle the exception (**catch**), or;
- Let the exception flow and declare it (**throws**).

Throws

Syntax

Declaring an exception

- The statement **throws** is used to declare one or more exceptions.
  - Here we inform users of the method **do** that it might throw an exception of the type **IOException**.

```
public static void do(String name) throws IOException {

    // ...
}
```

Throws

Example

```java
import java.io.*;

public class Keyboard {
    public static int getInt() {
        byte[] buffer = new byte[ 256 ];

        System.in.read(buffer); // throws IOException

        String s = new String(buffer);
        int num = Integer.parseInt(s.trim());
        return num;
    }
    public static void main(String[] args) {
        System.out.print("Please enter a number: ");
        int n = Keyboard.getInt();
        System.out.println("Number is: " + n);
    }
}
```

Compile time error

- **Checked** exceptions must be **handled** or **declared**.
  - Otherwise, they cause compile time errors.

```
> javac Keybord.java
Keyboard.java:9: unreported exception java.io.IOException;
must be caught or declared to be thrown
     System.in.read(buffer);
                   ^

1 error
```

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[256];

        System.in.read(buffer); // throws IOException

        String s = new String(buffer);
        int num = Integer.parseInt(s.trim());
        return num;
    }
    public static void main(String[] args) {
        System.out.print("Please enter a number: ");

        int n = Keyboard.getInt(); // throws IOException

        System.out.println("Number is: " + n);
    }
}
```

Compile time error

- **Checked** exceptions must be **handled** or **declared**.
    - Otherwise, they cause compile time errors.

```
> javac Keyboard.java
Keyboard.java:22: unreported java.io.IOException;
must be caught or declared to be thrown
     int n = Keyboard.getInt();
                               ^
1 error
```

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[256];

        System.in.read(buffer); // throws IOException

        String s = new String(buffer);
        int num = Integer.parseInt(s.trim());
        return num;
    }
    public static void main(String[] args)
    throws IOException {

        System.out.print("Please enter a number: ");

        int n = Keyboard.getInt(); // throws IOException

        System.out.println("Number is: " + n );
    }
}
```

Declaring an exception

- When **declaring** (**throws**) an exception without processing it (**catch**) the method terminates abruptly when the exception is thrown.

```
> java Keyboard
Please enter a number: oups
Exception in thread "main" java.lang.NumberFormatException
For input string: "oups"
     at java.lang.NumberFormatException.
        forInputString(NumberFormatException.java:48)
     at java.lang.Integer.parseInt(Integer.java:468)
     at java.lang.Integer.parseInt(Integer.java:518)
     at Keyboard.getInt(Keyboard.java:13)
     at Keyboard.main(Keyboard.java:23)
```

```java
import java.io.*;

public class Keyboard {

    public static int getInt() throws IOException {
        byte[] buffer = new byte[256];

        System.in.read(buffer);

        String s = new String(buffer);

        int num = Integer.parseInt(s.trim());

        return num;
    }

    // ...
    // ...

    public static void main(String[] args) throws IOException {

        int n;
        boolean done = false;

        while (! done) {
            System.out.print("Please enter a number: ");
            try {
                n = Keyboard.getInt();
                System.out.println("The number is " + n);
                done = true;
            } catch (NumberFormatException e) {
                System.out.println("Not a number!");
            }
        }
    }
}
```

Exemple

- This example handles exceptions of type **NumberFormatException**, but leaves out exceptions of type **IOException**.

```
> java Keyboard
Please enter a number: oups
Not a number!
Please enter a number: a1
Not a number!
Please enter a number:  1
The number is 1
```

New types

Theory

Learning objectives:

- **Creating** new types of exceptions.

Readings:

- Pages 29–36 of E. Koffman and P. Wolfgang.

New types

Syntax

How do we create new types of exceptions?

- Exceptions are **objects**.
- So all you have to do is **create new classes**.
- The subclasses of the class **Exception** are at **checked**.
- Unless they are subclasses of the class **RuntimeException**, then they are at **unchecked**.

Creating new types of exception

```
public class MyException extends Exception {
}
```

```
public class MyException extends RuntimeException {
    public MyException() {
        super();
    }
    public MyException(String message) {
        super(message);
    }
}
```

- **Why** creating new types of exception?

New types

Example

Example: Time

- In the following example, the method **parseTime** catches exceptions of type **NumberFormatException** or **NoSuchElementException** in order to throw an exception of a more informative type, **TimeFormatException**.

```java
public class TimeFormatException extends IllegalArgumentException {

    public TimeFormatException() {
        super();
    }

    public TimeFormatException(String msg) {
        super(msg);
    }
}
```

Example: Time

```
public class Time {

    // ...

    public static Time parseTime( String timeString ) {

        StringTokenizer st;
        st = new StringTokenizer(timeString, ":", true);

        int h, m, s;

        try {
            h = Integer.parseInt(st.nextToken ());
        } catch (NumberFormatException e1) {
            throw new TimeFormatException("not a number: "+timeString);
        } catch (NoSuchElementException e2) {
            throw new TimeFormatException("separator not found: "+timeString);
        }

        try {
            st.nextToken();
        } catch (NoSuchElementException e2) {
            throw new TimeFormatException("separator not found: "+timeString);
        }
```

Example: Time

```
try {
   m = Integer.parseInt(st.nextToken());
} catch (NumberFormatException e1) {
   throw new TimeFormatException("not a number: "+timeString);
} catch (NoSuchElementException e2) {
   throw new TimeFormatException("separator not found: "+timeString);
}

try {
   st.nextToken();
} catch (NoSuchElementException e2) {
   throw new TimeFormatException("separator not found: "+timeString);
}

try {
   s = Integer.parseInt(st.nextToken());
} catch (NumberFormatException e1) {
   throw new TimeFormatException("not a number: "+timeString);
} catch (NoSuchElementException e2) {
   throw new TimeFormatException("third field not found: "+timeString);
}
```

Example: Time

```
        if (st.hasMoreTokens()) {
            throw new TimeFormatException("invalid suffix:" + timeString);
        }

        if ((h<0) || (h>23) || (m<0) || (m>59) || (s<0) || (s>59) ) {
            throw new TimeFormatException( "values out of range:" + timeString);
        }

        return new Time(h,m,s);
    }
}
```

Summary

- A **precondition** is a condition that must be met before a method can be executed.
  - In object-oriented programming, we need to validate not only the **parameter values**, but also the **state of the object**.
- In Java, we use the **exceptions** to report run-time errors.
- The **try/catch** block is used to handle exceptions, i.e. stopping propagation.

Next module

- Abstract Data Type (ADT): **queue**.

References I

E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**
Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)
University of Ottawa