

UEF4.3. Programmation Orientée Objet

{s_sadeg, n_bousbia}@esi.dz

Ecole nationale Supérieure d'Informatique
(ESI)

Exceptions

A series of horizontal lines in shades of green and yellow, some solid and some dashed, extending across the bottom of the slide.

Introduction

Correctness and robustness

- A software program is **correct** if it successfully accomplishes the task for which it was designed.
- It is **robust** if it can handle illegal inputs and other unexpected situations in a reasonable way.

Example: A program that reads numbers entered by the user and displays them in ascending order is correct if it works for any set of input numbers. It is robust if it can also deal with non-numeric input, for example.

Introduction

Even if a program is correct, its execution may be abruptly interrupted due to exceptional events (incorrect data types, premature end of arrays or files, etc.) These events are called **Exceptions**. The term exception is shorthand for the phrase "exceptional event."

An exception is an event, which occurs **during the execution** of a program, that **disrupts** the normal flow of the program's instructions.

Introduction

- One approach to writing robust programs is to anticipate the problems that might arise by:
 - Adding test sequences to prevent them,
 - Implementing a system to signal abnormal program behavior by returning specific values.
- However, none of these solutions are entirely satisfactory because:
 - It is very difficult to account for all possible errors
 - The code becomes tedious and complex to write, making maintenance challenging due to its lack of readability (buried in nested conditionals).

Exception example

```
public class TestException {  
  
    public static void main(String[] args) {  
        int i = Integer.parseInt (args [0]);  
        int j = Integer.parseInt (args [1]);  
        System.out.println("result = " + (i / j));  
    }  
}
```

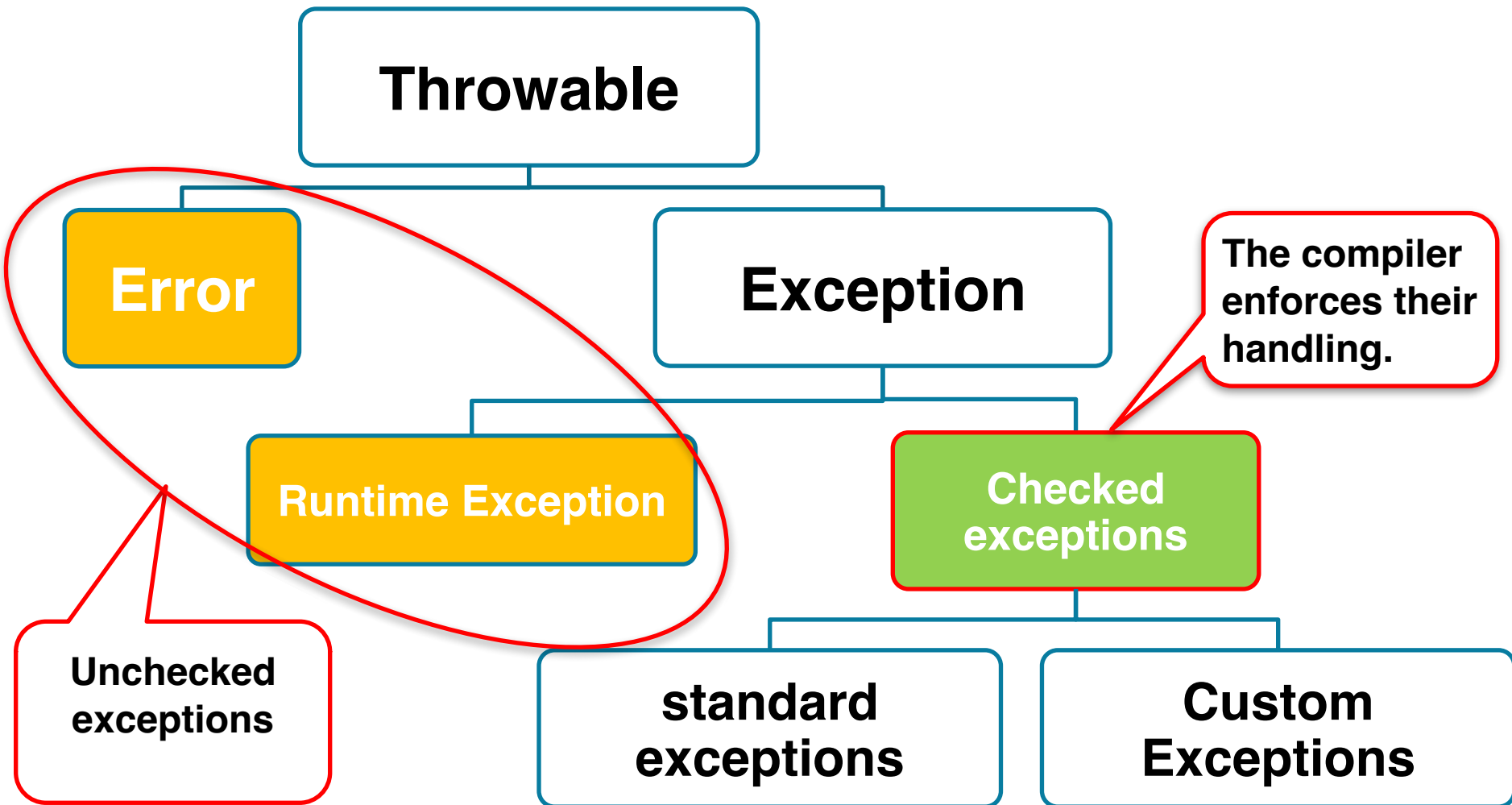
If the user enters a zero as the second value, the following message will be displayed

```
Exception in thread "main"  
java.lang.ArithmeticException: / by zero  
at TestException.main(TestException.java:6)
```

Exception types

- Java provides a flexible mechanism called exception handling, which allows detecting and managing errors that may occur during program execution.
- This mechanism is used to handle abnormal behavior in a portion of the code.
- There are three main categories of exceptions:
 - Error
 - Exception
 - RuntimeException

Exception inheritance tree



Exception types

1. Error exceptions

- These are exceptional events external to the program (such as class loading failures or hardware malfunctions) that cannot be anticipated or recovered from.
- They belong to the Error class or its subclasses (e.g., VirtualMachineError, IOError, ...etc.).

Example: Suppose an application successfully opens a file but fails to read it due to a hardware failure or an operating system issue. This situation would trigger an exception of type `java.io.IOException`.

Exception types

2. Checked exceptions

Checked exceptions are exceptions that a well-designed program should anticipate and handle. They are subclasses of the Exception class.

Example: Suppose an application prompts the user to enter a filename. If the user provides the name of a non-existent file, a **java.io.FileNotFoundException** will be thrown. A well-implemented program should handle this exception by notifying the user that the file does not exist and prompting them to enter a valid filename.

Exception types

3. RuntimeException exceptions

RuntimeException is a subclass of Exception. These are exceptional events that occur internally within the JVM and the program (related to the language) and can be handled by the programmer.

Such exceptions include: Arithmetic exceptions (e.g., division by zero). Null pointer exceptions (when trying to access an object through a null reference). Indexing exceptions (index out of bounds).

Example: If an application passes null instead of a filename to a method, a `NullPointerException` will be thrown.

Exception handling

- A checked exception is an exception that extends the Exception class. It can be either a standard exception (provided by the JDK) or a custom exception.
- In a well-structured Java program, any code that may generate (or throw) a checked exception must handle it. using one of the following approaches:
 1. Handling the exception with a try/catch block.
 2. Propagating the exception using the throws keyword

1. try-catch block

A decorative graphic consisting of a solid green horizontal bar, followed by a white horizontal bar, and then a series of three thin, parallel green horizontal lines of varying lengths extending to the right.

try-catch block

- The **try** block contains a set of statements representing normal execution, but which may potentially generate errors.
- The **catch** block handles specific exceptions by defining how to process a particular type of error. When an exception occurs, the corresponding catch block is executed based on the exception class passed as a parameter.

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

Example

```
public class TestException2 {  
  
    public static void main(String[] args) {  
        int i = Integer.parseInt (args [0]);  
        int j = Integer.parseInt (args [1]);  
  
        try {  
  
            System.out.println (« result = " + (i / j) );  
        }  
  
        catch (ArithmeticException e) {  
            System.out.println ("Division by zero ");  
        }  
    }  
}
```

what happens when an exception is thrown?

- At any point during execution, a statement or method can throw an exception.
- A method can either catch (handle) the exception or allow it to propagate to the calling method, which may either handle it or let it continue propagating further up the call stack,

what happens when an exception is thrown?

Case 1: Outside a try block

1. The method immediately returns, and the exception propagates to the calling method.
2. Control is handed over to the calling method.
3. The exception may then be caught and handled by this calling method or continue propagating up the stack.

what happens when an exception is thrown?

Case 2. Inside a try block

If an instruction within the try block throws an exception:

1. The subsequent instructions in the try block are not executed.
2. If at least one catch clause matches the exception type,
 - the first appropriate catch clause is executed.
 - Execution then resumes immediately after the try/catch block.

Otherwise:

- The method returns immediately.
- The exception propagates to the calling method.

what happens when an exception is thrown?

Remarks (1)

1. When no exception is thrown within the try block
 - The execution of the try block proceeds as if there were no try-catch blocks.
 - The program continues after the try-catch block.
2. If the catch block is empty (i.e., no instructions between the braces), the caught exception is ignored. Such usage of the try/catch statement is considered bad practice: it is always preferable to provide appropriate handling when catching an exception.

Que se passe t-il quand une exception est levée ?

Remarks (2)

3. Avoid catching an exception with a catch block that only displays a message without providing meaningful information about the issue. At a minimum, print the stack trace using `printStackTrace()` to help diagnose the problem.
4. If a variable is declared inside a try block, it cannot be accessed within the catch block. Common variables should be declared outside the try/catch block.

What if the exception is not processed?

If an exception propagates to the main method without being handled (and is not handled by any other method either):

- The program execution is terminated.
- The exception message is displayed, along with a stack trace showing the sequence of method calls leading to the exception.

Handling multiple exceptions

- When multiple types of errors and exceptions need to be handled, a separate catch block should be defined for each type of event.
- An exception type refers to an instance of the exception class or one of its subclasses, Therefore, in the sequential order of catch clauses, a more specific exception type should not appear after a catch block handling its superclass.
- It is essential to order catch clauses correctly, handling more specific exceptions (subclasses) first, followed by more general exceptions. Otherwise, the compiler will generate an error message.

Handling multiple exceptions

```
public class TestException2 {  
  
    public static void main(String[] args) {  
        // Insert code to start the application here.  
        int i = Integer.parseInt (args [0]);  
        int j = Integer.parseInt (args [1]);  
        try {  
            System.out.println("result = " + (i / j));  
        }  
        catch (Exception e) {  
        }  
        catch (ArithmeticException e) {  
            System.out.println("Divion by zero");  
        }  
    }  
}
```



unreachable catch
block

The finally block

- When an exception occurs, the execution of a method is interrupted, and control is transferred to the appropriate catch block. However, a finally block can be used to define instructions that will always execute, regardless of whether an exception is thrown or not.
 - If no exception is thrown, the finally block executes after the try block.
 - If an exception is thrown and caught, the finally block executes after the catch block.
 - If an exception is thrown but not caught, the finally block still executes before the exception propagates further up the call stack.
- This ensures that critical operations, such as closing resources or releasing memory, are always performed.

The finally block

- The finally block is introduced by the finally keyword and must always be placed after the last catch block.

```
try {  
    // Code that may throw an exception  
}  
catch (ExceptionType e) {  
    // Exception handling  
}  
finally {  
    /* Cleanup instructions , such as closing files,  
    releasing memory, or terminating network connections*/  
}
```

Remark The try statement can include zero or more catch clauses and, optionally, a finally clause. A catch block is not mandatory when using a finally bloc.

Throwing an exception

A decorative graphic consisting of several horizontal lines in shades of green and yellow, extending across the width of the slide below the title.

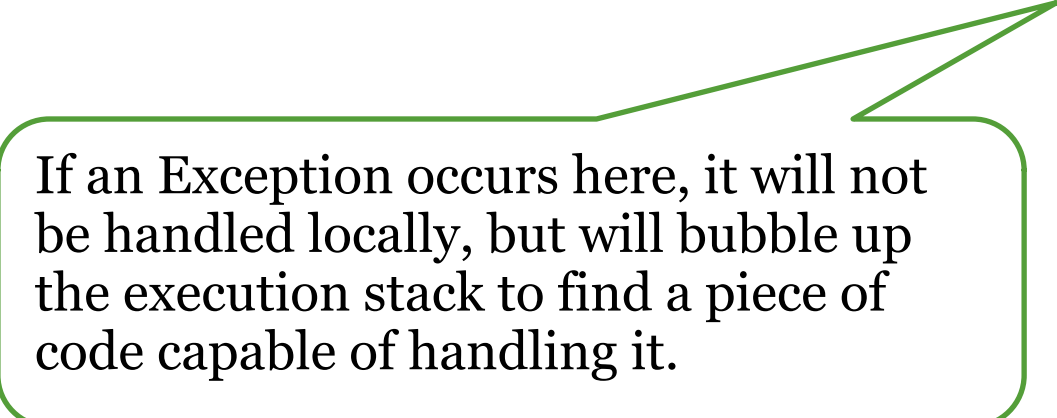
The *throws* keyword

- Any method that can throw a checked exception must either:
 - Handle the exception locally using a try/catch block, or
 - Propagate the exception to be handled by another method using the throws keyword.
- We have already seen how to use the try/catch block. Now, let's explore the use of throws.

The *throws* keyword

Example

```
public class TestThrows {  
    public void test(String[] a) throws ArithmeticException {  
        int i = Integer.parseInt (a[0]);  
        int j = Integer.parseInt (a[1]);  
  
        System.out.println("résultat = " + (i / j));  
    }  
    ...  
}
```



If an Exception occurs here, it will not be handled locally, but will bubble up the execution stack to find a piece of code capable of handling it.

The *throw* keyword

- In such cases, the program can throw an exception, expecting that another part of the program will catch and handle it appropriately.
- Exceptions can be explicitly thrown within methods to signal errors or unexpected conditions.
 - They may come from built-in exception classes provided by the JDK.
 - They can also be custom exceptions, specifically designed to fit the application's logic.
- This typically occurs when the program encounters an exceptional or error condition **that cannot be handled immediately** at the point where it is detected.

The *throw* keyword

Example

When a method throws an exception using the throw statement, it must declare the exception in its signature using the throws keyword. This informs the caller that the method may generate an exception, which must be handled appropriately.

```
public class TestThrow {  
  
    public void test(String[] a) throws ArithmeticException{  
        int i = Integer.parseInt (a [0]);  
        int j = Integer.parseInt (a [1]);  
        if(j==0) throw new ArithmeticException();  
  
        System.out.println("result = " + (i / j));  
    }  
}
```

Use of throws

- If a method m2 has a header containing throws `XXException`, and a method m1 calls m2, then:
 - Either m1 wraps the call to m2 inside a try-catch (`XXException`) block.
 - Or m1 declares (using the `throws` keyword) that it may throw an `XXException` (or a broader type).

Custom Exceptions

- When necessary, custom exceptions can be created by defining subclasses of the Exception class (by convention, the word "Exception" is included in the name of the new class).
- Then, the throw keyword is used, followed by an object of the custom exception class.

Custom Exceptions

Example

- Let's assume we want to work with points that have only non-negative coordinates. To enforce this rule, we will use a `Point` class with a constructor that takes two arguments. Within this constructor, we will validate the user-provided coordinates. If either coordinate is invalid, an exception will be thrown using the *throw* keyword.
- The *throw* statement requires an object of the relevant exception type.
- Therefore, we will create a class named *CoordinatesException*, which extends the `Exception` class:

```
class CoordinatesException extends Exception {}
```

```

class Point{
private int x; private int y;

public Point(int x, int y) throws CoordinatesException{
if((x<0) || (y<0)) throw new CoordordinatesException();
this.x = x;
this.y = y;}
}

```

```

public class ExceptionPersonnalisee {
public static void main(String args[]){
int x = Integer.parseInt(args[0]);
int y = Integer.parseInt (args[1]);

try{
Point p = new Point (x,y);}
catch(CoordonneesException e){
System.out.println(" negative coordinate");}
}
}

```

The Throwable class

- Throwable is the superclass of all exceptions and directly extends the Object class. Its main methods are:
 - **String getMessage():** Returns the exception message.
 - **void printStackTrace():** Prints the error message and the call stack that led to the problem.
 - **void printStackTrace(PrintStream s):** Similar to printStackTrace(), but directs the output to a specified stream.

La classe Throwable

```
public class TestMethodesThrowable {  
    public static void main(java.lang.String[] args) {  
        int i = 3;  
        int j = 0;  
        try {  
            System.out.println("result = " + (i / j));  
        }  
        catch (ArithmeticException e) {  
            System.out.println("getmessage");  
            System.out.println(e.getMessage());  
            System.out.println(" ");  
            System.out.println("printStackTrace");  
            e.printStackTrace();  
        }  
    }  
}
```

Execution Result

```
getMessage  
/ by zero
```

```
printStackTrace
```

```
java.lang.ArithmeticException: / by zero
```

```
at TestMethodesThrowable.main(TestMethodesThrowable.java:6)
```

Exercise 1. The following Java program has a compilation problem. What is the problem? Suggest a solution.

```
public class Exercice {  
  public static void main ( String args[ ]) {  
    int k ;  
    try {  
      k = 1 / Integer.parseInt ( args [0] ) ;}  
    catch ( RuntimeException ex ) {  
      System.err.println( " Runtime "+ex ) ;}  
    catch ( IndexOutOfBoundsException ex ) {  
      System.err.println( " Index "+ex ) ;}  
    catch ( ArithmeticException ex ) {  
      System.err.println(" Index "+ex ) ;}  
    }  
  }  
}
```

Exercise 2. What do the following programs display?

```
public class C1 {  
    public static void main(String[] args) {  
        try {  
            return; }  
        finally{  
            System.out.println( "Finally" ); }  
        }  
    }  
}
```

Exercise 2: What do the following programs display?

```
public class C1 {  
    public static void main(String[] args) {  
        try {  
            return; }  
        finally{  
            System.out.println( "Finally" ); }  
        }  
    }
```



Finally

Exercise 2: What do the following programs display?

```
public class C2 {  
    public static void main(String [] args) {  
        try {  
            M();  
            System.out.print("A");  
        }  
        catch (Exception ex) {  
            System.out.print("B");  
        }  
        finally {  
            System.out.print("C");  
        }  
        System.out.print("D");  
    }  
    public static void M() {}  
}
```

Exercise 2: What do the following programs display?

```
public class C2 {  
    public static void main(String [] args) {  
        try {  
            M();  
            System.out.print("A");  
        }  
        catch (Exception ex) {  
            System.out.print("B");  
        }  
        finally {  
            System.out.print("C");  
        }  
        System.out.print("D");  
    }  
    public static void M() {}  
}
```



ACD

Exercise 2: What do the following programs display?

```
public class C3 {  
    public static void main(String [] args){  
        try {  
            M();  
            System.out.print("A");  
        }  
        catch (Exception ex) {  
            System.out.print("B");  
        }  
        finally {  
            System.out.print("C");  
        }  
        System.out.print("D");  
    }  
    public static void M() {  
        throw new Error();  
    }  
}
```

Exercise 2: What do the following programs display?

```
public class C3 {  
    public static void main(String [] args){  
        try {  
            M();  
            System.out.print("A");  
        }  
        catch (Exception ex) {  
            System.out.print("B");  
        }  
        finally {  
            System.out.print("C");  
        }  
        System.out.print("D");  
    }  
    Public static void M() {  
        throw new Error();  
    }  
}
```

C

```
java.lang.Error  
        at C3.M(C3.java:23)  
        at  
C3.main(C3.java:11)
```

Exercise 1: What do the following programs display?

```
public class C4 {  
    public static void main(String [] args) {  
        try {  
            M();  
            System.out.print("A");  
        }  
        catch (RuntimeException ex){  
            System.out.print("B");  
        }  
        catch (Exception ex1) {  
            System.out.print("C");  
        }  
        finally {  
            System.out.print("D");  
        }  
        System.out.print("E");  
    }  
    public static void M(){  
        throw new RuntimeException();  
    }  
}
```

Exercise 1: What do the following programs display?

```
public class C4 {  
    public static void main(String [] args) {  
        try {  
            M();  
            System.out.print("A");  
        }  
        catch (RuntimeException ex){  
            System.out.print("B");  
        }  
        catch (Exception ex1) {  
            System.out.print("C");  
        }  
        finally {  
            System.out.print("D");  
        }  
        System.out.print("E");  
    }  
    public static void M(){  
        throw new RuntimeException();  
    }  
}
```



BDE