People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University M'hamed Bouguerra - Boumerdes
Institute of Electrical and Electronics Engineering
Electronics Department

**Option:** Computer engineering

**LAB REPORT n°7**

January 6, 2023

**Title**

# EE423: Advanced Programming/ Working with Threads and Exceptions

Authored by :
**Agli Wafa**
**Zidane Aymen**

Instractor :
**Dr.A Zitouni**

**Session : 2022/2023**

## Contents

**1 Part 1: Exceptions** **1**

1.1 *Question 1: Handling an exception (try-catch statement)* . . . . . 1

1.2 *Question 2: throwing a runtime exception (RuntimeException)* . . . . . 2

1.3 *Question 3: throwing a checked exception (Exception)* . . . . . 2

1.4 *Question 4: Create your own exceptions* . . . . . 3

1.5 *Question 5: Catching different kinds of exceptions* . . . . . 5

**2 Part 2: Threads** **6**

2.1 *Question 1: Know how many processors your computer has* . . . . . 6

2.2 *Question 2: Creating and starting many threads* . . . . . 6

2.3 *Question 3: Create a thread using the Runnable interface* . . . . . 7

2.4 *Question 4: Accessing common shared data* . . . . . 7

2.5 *Question 5: Checking priorities* . . . . . 8

2.6 *Question 6: Polling with Sleep* . . . . . 9

2.7 *Question 7: Protecting methods with synchronized* . . . . . 10

**3 conclusion** **11**

**Introduction**

In this lab, in the first part, we will learn to use exceptions to handle the errors that might arise during the execution of a code. we will see the syntax of defining your own custom exceptions and use them to handle alternative behaviors. In the second part, we will learn about multi-threading in Java. Java supports concurrent programming via threads. we will see the different methods of creating threads and the steps to start the threads.

**Tools and Software:**

1. A PC with ECLIPSE IDE V8.

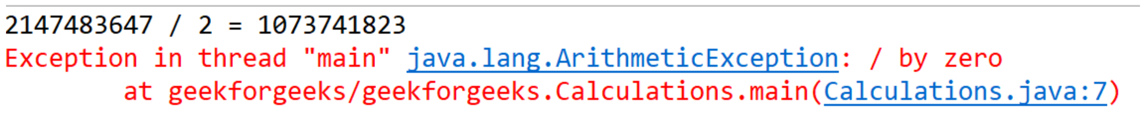
2. Online LaTeX Editor for writing the report.

## 1 Part 1: Exceptions

### 1.1 *Question 1: Handling an exception (try-catch statement)*

```java
public class Calculations{
public static void main(String[] args) {
    int[] dividers = {2, 0, -5, 0, 100};
    for(int i =0; i<dividers.length; i++) {
    int half = Integer.MAX_VALUE/dividers[i];
    System.out.println(Integer.MAX_VALUE+" / "+dividers[i]+" = "+half);
    }
    }
}

```



```
2147483647 / 2 = 1073741823
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at geekforgeeks/geekforgeeks.Calculations.main(Calculations.java:7)
```

Figure 1: The output of the code above

A runtime error has been raised because of the devision by 0; in order to handle this exception we may use try/catch as follows:

```java
   int[] dividers = { 2, 0, -5, 0, 100 };
        for (int i = 0; i < dividers.length; i++) {
            try {
                int half = Integer.MAX_VALUE / dividers[i];
                System.out.println(Integer.MAX_VALUE + " / " + dividers[i] + " = " + half);
            } catch (ArithmeticException e) {
                System.out.println("You can not divide " + Integer.MAX_VALUE + " by zero ");
            }

        }

```

### 1.2 *Question 2: throwing a runtime exception (RuntimeException)*

```java
public class Calculations {
    public static boolean isPalindrome(String value) {
        for(int i = 0; i< value.length()/2; i++)
        if(value.charAt(i) != value.charAt(value.length()-i-1))
        return false;
        return true;
    }
    public static void main(String[] args) {
        String a = null;
        System.out.println(isPalindrome(a));
        a = "madam";
        System.out.println(isPalindrome(a));
    }
    }

```

- When running this code a NullPointerException was raised because we are trying to take the length of null as if it were an array.
- we should throw a new IllegalArgumentException each time we pass a value to the palindrome method

```java
        public static void Check(String num) {
           if (num == null)
           throw new IllegalArgumentException();
           }

```

```java
    public static void main(String[] args) {
     String a = null;
     try {
       Check(a);
       System.out.println(isPalindrome(a));
       } catch (IllegalArgumentException z) {
       System.out.println("The string shouldn't be null");
         }
     a = "madam";
     try {
       Check(a);
       System.out.println(isPalindrome(a));
           } catch (IllegalArgumentException z) {
       System.out.println("The string shouldn't be null");
             }

}

```

### 1.3 *Question 3: throwing a checked exception (Exception)*

```java
public class Calculations {
public static void checkFile(String path) {
```

```
File file = new File(path);
if (file.exists()) {
System.out.println("The file exists");
} else {
System.out.println("The file does not exist");
}
}
public static void main(String[] args) {
String path = "C:/unknow_file.txt";
checkFile(path);
}
}

```

The output of the previous snippet is The file does not exist.

- by throwing a FileNotFoundException we create a checked exception, so if not handled a compilation error occur:

```
 public static void checkFile(String path) throws FileNotFoundException {
File file = new File(path);
if (file.exists()) {
System.out.println("The file exists");
} else {
System.out.println("The file does not exist");
throw new FileNotFoundException();
}
}

```

- in order to handle this exception the call should be surrounded by a try/catch

```
String path = "C:/unknow_file.txt";
 try {
 checkFile(path);
   } catch (FileNotFoundException e) {
   //  throw new FileNotFoundException();
     System.out.println("File: " +path + " (Not found)");
   }

```

## 1.4 *Question 4: Create your own exceptions*

- A checked exception called EmptyException:

```
class EmptyException extends Exception {}

```

- an unchecked exception called NegativeException:

```
class NegativeException extends RuntimeException{}

```

- a method that searches the max value of a non empty array:

```
      public static int maxVal(int[] array){
    int max= array[0];
    for(int i = 0; i<array.length; i++ ) {
       if(array[i]>max) {
            max = array[i];
           }
         }

    return max;
  }

```

- EmptyException:

```
    public static int maxVal(int[] array) throws Exception{
   if (array.length == 0) {throw new EmptyException();}
    int max= array[0];
    for(int i = 0; i<array.length; i++ ) {
          if(array[i]>max) {
            max = array[i];
           }
         }

    return max;
  }

```

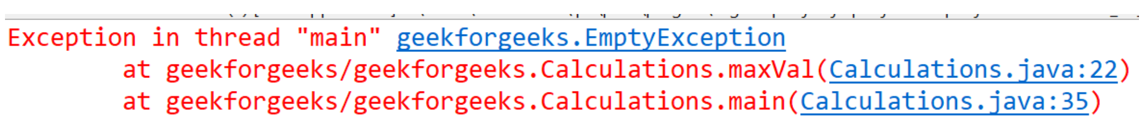
- NegativeException:

```
   public static int maxVal(int[] array) throws Exception{
   if (array.length == 0) {throw new EmptyException();}
    int max= array[0];
    for(int i = 0; i<array.length; i++ ) {
      if (array[i]<0) {throw new NegativeException();}
      else if(array[i]>max) {
            max = array[i];
           }
         }

    return max;
  }

```

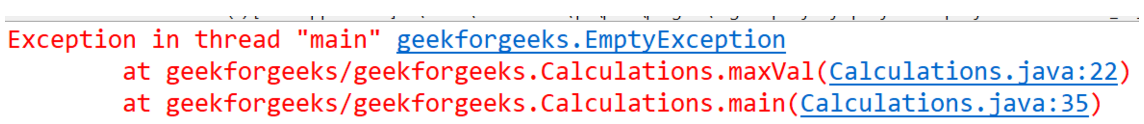
- method Call with empty array:



```
Exception in thread "main" geekforgeeks.EmptyException
	at geekforgeeks/geekforgeeks.Calculations.maxVal(Calculations.java:22)
	at geekforgeeks/geekforgeeks.Calculations.main(Calculations.java:35)
```

Figure 2

- method Call with negative value:



```
Exception in thread "main" geekforgeeks.EmptyException
	at geekforgeeks/geekforgeeks.Calculations.maxVal(Calculations.java:22)
	at geekforgeeks/geekforgeeks.Calculations.main(Calculations.java:35)
```

Figure 3

- handling exceptions:

```
try {
maxVal(array);
}catch (EmptyException g) {
  System.out.println("the array must not be empty");
}
catch(NegativeException z) {
  System.out.println("the array must not contain any negative values");
}

```

## 1.5 *Question 5: Catching different kinds of exceptions*

- AnimalException:

```
  class AnimalException extends Exception {
  public AnimalException() {}
public AnimalException(String a) {
  super(a);
}
  }

```

- MammalException:

```
  class MammalException extends AnimalException {
  public MammalException() {}
public MammalException(String a) {
  super(a);
}
  }

```

- TigerException:

```
class TigerException extends MammalException {
public TigerException() {}
public TigerException(String a) {
  super(a); }
}

```

- Surrounding the call with try/catch:

```
  public static void animal() throws AnimalException {
  Random random = new Random();
  int val = random.nextInt(4);
```

```
    switch(val) {
    case 0: System.out.println("Animal"); throw new AnimalException();
    case 1: System.out.println("Mammal"); throw new MammalException("A mammal
   exception occurred");
    case 2: System.out.println("Tiger"); throw new TigerException("A tiger
   exception occurred");
    }
    }
    public static void main(String[] args) {
    try {
      animal();
      System.out.println("try");
    }catch (TigerException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
    catch (MammalException d) {
      System.out.println(d.getMessage());
    }
    catch (AnimalException v) {
      System.out.println(v.getMessage());
    }
    finally {
      System.out.println("Finished!");
    }
    }

```

## 2 Part 2: Threads

### 2.1 *Question 1: Know how many processors your computer has*

In order to get running processors number we print:
**System.out.println(Runtime.getRuntime().availableProcessors());** //4

### 2.2 *Question 2: Creating and starting many threads*

```
 class PrintThreadInfo extends Thread{
  public void run() {
    System.out.println("My ID: "+this.getId()+" My name: "+this.getName()+
        " My priority: "+this.getPriority() +" My thread group: "+ this.getThreadGroup()
   );
    }
}

public class StartingThreads {

    public static void main(String[] args) {
      for(int i = 0; i<20; i++) {
      Thread t = new PrintThreadInfo();
        t.start();

        try {
          t.sleep(10);
```

```java

        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
  }
}

```

## 2.3 *Question 3: Create a thread using the Runnable interface*

```java
class MinValue implements Runnable {
  double min;

  public MinValue(double[] array) {
    double min = array[0];
    for (int i = 0; i < array.length; i++) {
      if (array[i] < min) {
        min = array[i];
      }
      this.min = min;
    }
  }

  public void run() {
    Thread currentThread = Thread.currentThread();
    System.out.println("Thread: " + currentThread.getName());
    System.out.println("Min value: " + min);
  }
}

public class exo2 {
  public static void main(String[] args) {
    Thread currentThread = Thread.currentThread();
    System.out.println("Thread: " + currentThread.getName());
    double[] array = { 1, 7, 8, 0 ,-1};
    Thread t = new Thread(new MinValue(array),"rr");
    t.start();
  }
}

```

## 2.4 *Question 4: Accessing common shared data*

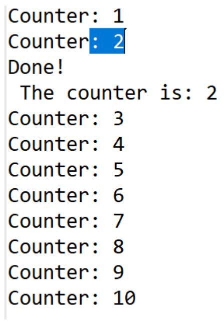
```java
class thread extends Thread{
  static int counter =0;
  public void run() {
    counter+=1;
    System.out.println("Counter: " + counter);
  }

}

public class Counter {
  public static void main(String[] args) {
    for(int i = 0; i<10; i++) {
    Thread t = new thread();
    t.start();
    }
    System.out.println("Done!\n The counter is: "+thread.counter);
}
}

```

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. we can solve the problem through synchronization



```
Counter: 1
Counter: 2
Done!
 The counter is: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
Counter: 9
Counter: 10
```

Figure 4

```
class thread extends Thread{
  static int counter =0;
  public void run() {
    counter+=1;
    System.out.println("Counter: " + counter);
  }

}

public class Counter {
  public static void main(String[] args) throws InterruptedException {
    for(int i = 0; i<10; i++) {
    Thread t = new thread();
    t.start();
    t.sleep(10000);

    }
    System.out.println("Done!\n The counter is: "+thread.counter);
}
}

```

## 2.5 *Question 5: Checking priorities*

```
class thread1 extends Thread {
  static long val(int n) {
    return n <= 0 ? 1 : val(n - 1) + val(n - 2);
  }

  public void run() {
    long v = val(45);
    System.out.println(v + " " + this.getPriority());
  }
}

public class CheckingPriorities {
  public static void main(String[] args) {
    Thread ThreadPrio1 = new thread1();
    ThreadPrio1.setPriority(1);
    ThreadPrio1.start();
    Thread ThreadPrio5 = new thread1();
    ThreadPrio5.setPriority(5);
    ThreadPrio5.start();
    Thread ThreadPrio9 = new thread1();
    ThreadPrio9.setPriority(9);
    ThreadPrio9.start();
  }
}

```

```
2971215073 9          2971215073 5          2971215073 1
```

## 2.6 *Question 6: Polling with Sleep*

```
class Counter1 extends Thread{
  static int counter1 =0;
  public void run() {
    do {
    counter1+=1;
    System.out.println("Counter value: " + counter1);
    try {
      sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    }while (counter1<100);
  }
}

class Monitor extends Counter1{
  public void run() {
    do {
    if (this.counter1 >= 100) {
      System.out.println("reached.");
      break;
    }
    try {
      sleep(10);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    }while(true);
  }

}
public class CheckResults {
  public static void main(String[] args) {
  Thread t = new Counter1();
  t.start();

  Thread m = new Monitor();
  m.start();
  }
}

```

## 2.7 *Question 7: Protecting methods with synchronized*

Thread safety is the property of an object that guarantees safe execution by multiple threads at the same time. Now that we have multiple threads capable of accessing the same objects in memory, we have to make sure to organize our access to this data such that we don't end up with invalid or unexpected results. Since threads run in a shared environment and memory space, how do we prevent two threads from interfering with each other?

```

class Transaction extends Thread{
    BankAccount accountSender;
    BankAccount accountReceiver;
    double amount;
    Object monitor;

    public Transaction(BankAccount aS, BankAccount aR, double amount/*, Object monitor*/
    ) {
        this.accountSender = aS;
        this.accountReceiver = aR;
        this.amount = amount;
     //   this.monitor = monitor;
    }

    @Override
    public void run() {
     //   synchronized (monitor) {
        if (this.accountSender.getAmount()>amount) {
            this.accountSender.reduceAmount(amount);
            try {
                sleep(100);
            } catch (InterruptedException e) {}
            this.accountReceiver.addAmount(amount);
     //   }

    }}
}
public class BankAccount {
    public double amount;
    public BankAccount(double amount) {this.amount = amount;}
    public double getAmount() {return amount;}
    public void addAmount(double amount) {this.amount+= amount;}
    public void reduceAmount(double amount) {this.amount-= amount;}
    public static void main(String[] args)  throws InterruptedException {
        Object monitor = new Object();
        BankAccount account1 = new BankAccount(1000);
        BankAccount account2 = new BankAccount(1000);

        Transaction t1 = new Transaction(account1, account2, 10/*, monitor*/);
        Transaction t2 = new Transaction(account1, account2, 10/*, monitor*/);
        Transaction t3 = new Transaction(account1, account2, 10/*, monitor*/);
        t1.start();t1.join();
        t2.start();t2.join();
        t3.start();t3.join();



        System.out.println(account1.amount+ " "+ account2.amount);
    }
}


```

**Explanation:** This code is referred to as **a synchronized block** . Each thread that arrives will first check if any threads are in the block. In this manner, a thread "acquires the lock" for the monitor. If the lock is available, a single thread will enter the block, acquiring the lock and preventing all other threads from entering. While the first thread is executing the block, all threads that arrive will attempt to acquire the same lock and wait for first thread to finish. Once a thread finishes executing the block, it will release the lock, allowing one of the waiting threads to proceed.

## 3 conclusion

In this laboratory in the first part we introduced Exceptions in Java, how to catch them and handling them, in the second part we were dealing with Threads in java and we concluded the following:

1. **A thread** is the smallest unit of execution that can be scheduled by the operating system.
2. **A process** is a group of associated threads that execute in the same, shared environment.
3. It follows,then, that a single-threaded process is one that contains exactly one thread, whereas a multi-threaded process is one that contains one or more threads.