Lab Assignment 7: Working with Threads and Exceptions

Course: Advanced Programming (EE423)
Institution: IGEE, Boumerdes University

December 9, 2022

1 Objectives

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

2 Assignment

2.1 Part 1: Exceptions

Exceptions are used when "something goes wrong." However, the word "wrong" is subjective. The following code returns –1 instead of throwing an exception if no match is found:

```
public int indexOf(String[] names, String name) {
 for (int i = 0; i < names.length; i++)
  if (names[i].equals(name)) return i;
 return -1;
}
```

This approach is common when writing a method that does a search. For example, imagine being asked to find the name Joe in the array. It is perfectly reasonable that Joe might not appear in the array. When this happens, a special value is returned. An exception should be reserved for exceptional conditions like names being null. In general, try to avoid return codes. Return codes are commonly used in searches, so programmers are expecting them. In other methods, you will take your callers by surprise by returning a special value. An exception forces the program to deal with them or end with the exception if left unhandled, whereas a return code could be accidentally ignored and cause

problems later in the program. An exception is like shouting, "Deal with me!" In this part of this lab, you will learn how to create different types of exceptions and handle them.

Why exceptions?

```
public class Calculations{
 public static void main(String[] args) {
   System.out.println(10/0);
   System.out.println("Finished");
 }
}
```

If you run the previous code you will application will crash and the JVM will generate the following exception:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Calculations.main(Calculations.java:3)
```

The message `Finished` will not be printed.

To prevent your code from crashing, you can handle the exception and let the program finish its execution.

```
public class Calculations {
 public static void main(String[] args) {
  try {
   System.out.println(10/0);
  } catch (ArithmeticException e) {
   System.out.println("An ArithmeticException occurred, and it was handled");
  }
  System.out.println("Finished");
 }
}
```

When you run the previous code, you get the following output:

```
An ArithmeticException occurred, and it was handled
Finished
```

2.1.1 Question 1: Handling an exception (try-catch statement)

Given the following code, what is its output?

```
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);
```

```
  }
 }
}
```

You want this code to run without crashing. For this you will need to handle the exceptions being raised when you try to divide by 0. Instead of letting the code crash, print the message

```
"You can not divide "+Integer.MAX_VALUE+" by zero"
```

and then resume the rest of the divisions.

What is the output of the code after handling the exceptions?

2.1.2 Question 2: throwing a runtime exception (RuntimeException)

What is the output of this code?

```
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));
  }
}
```

Which exception was raised when you ran the previous code? Why was it raised? You want to handle that exception by throwing the exception

```
IllegalArgumentException
```

when the user passes a value equal to null. Which changes should you add to the previous code to make it happen? Code it.

After you have added the code that throws the required exception, use the try-catch to intercept it, and not let your code crash. The code should be able run the second case of isPalindrome.

2.2 Question 3: throwing a checked exception (Exception)

As you have seen from the previous case, RuntimeException can remain unchecked. Now, we will see what will happen when we throw a checked exception (not a RuntimeException). Given the following code:

```
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);
 }
}
```

What is the output of this code?

We want this code to throw the exception FileNotFoundException when the file does not exist. What happens when you throw that exception?

As you already know, a checked exception must be handled. There are two ways you can do that:

1. the method must handle it itself.
2. the method pass the exception to the caller who will handle it.

In this case, handle the exception in the main function by printing the message:

```
"File: " +path + " (Not found)"
```

if that exception is raised.

2.2.1 Question 4: Create your own exceptions

- Create a checked exception called EmptyException.
- Create an unchecked exception called NegativeException.
- Write a method that searches the max value of a non empty array. The array must contain only positive numbers.
- If the array passed is empty throw the exception EmptyException.
- If the array passed contains any negative values throw the exception NegativeException.
- Call the method twice with the following arrays: and 1,2,-3
- Your code must handle both exception and should terminate without throwing any exception.

- If the array passed is empty, print: the array must not be empty
- if the array passed contains any negative values print: the array must not contain any negative values

2.2.2 Question 5: Catching different kinds of exceptions

- Create a checked exception called AnimalException
- Create a checked exception called MammalException which extends the class AnimalException
- Create a checked exception called TigerException which extends the class MammalException
- Each exception should have two constructors:
  - One without parameters
  - and one with a string parameters that holds the error message.

Given the following code:

```
public class Calculations {
 public static void animal() throws AnimalException {
  Random random = new Random();
  int val = random.nextInt(4); // it can: not throw an exception
  switch(val) {
   case 0: System.out.println("Animal"); throw new AnimalException("Animal");
   case 1: System.out.println("Mammal"); throw new MammalException("Mammal");
   case 2: System.out.println("Tiger"); throw new TigerException("Tiger");
  }
 }
 public static void main(String[] args) {
  animal();
 }
}
```

- Surround the call to the method animal with a try and catch block. For each exception a different message is printed according to the exception raised. Use getMessage() to get the message of each exception.
  - For AnimalException print: An animal exception occurred instead of Animal
  - For MammalException print: A mammal exception occurred instead of Mammal
  - For TigerException print: A tiger exception occurred instead of Tiger
- At the end, if any exception occurred, or not, you should print the message: Finished!

2.3 Part 2: Threads

2.3.1 Question 1: Know how many processors your computer has

Run the following code to know how many processors your CPU has.

```
public class CPUs {
 public static void main(String[] args) {
   System.out.println(Runtime.getRuntime().availableProcessors());
 }
}
//Example for (Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz): 8
```

2.3.2 Question 2: Creating and starting many threads

Create a class called `StartingThreads` that does create and start 20 threads. The class of the threads you create is called `PrintThreadInfo` and must extend the class `Thread`. Each thread will print the message:

```
My ID: ?? My name: ?? My priority: ?? My thread group: ??.
```

To get a thread's information you can use the functions: getName(), getPriority(), getId(), ... of the class Thread. To create a thread you can use the code:

```
Thread t = new PrintThreadInfo(); // create a thread object
t.start(); // start the thread
```

You should get as an output something similar to this:

```
...
My ID: 15 My name: Thread-0 My priority: 5 My thread group: java.lang.ThreadGroup[name=main,maxpri=10]
My ID: 21 My name: Thread-6 My priority: 5 My thread group: java.lang.ThreadGroup[name=main,maxpri=10]
My ID: 20 My name: Thread-5 My priority: 5 My thread group: java.lang.ThreadGroup[name=main,maxpri=10]
...
```

2.3.3 Question 3: Create a thread using the Runnable interface

Create a thread called `MinValue` that implements the `Runnable` interface. The `run()` function should print the min value contained in an array of double values passed to the constructor of the class `MinValue`. Your code should print something like this:

```
Thread: main
Thread: Thread-0
Min value: 3.0
```

Use this code to get the name of the current thread:

```
Thread currentThread = Thread.currentThread();
System.out.println("Thread: " + currentThread.getName());
```

2.3.4 Question 4: Accessing common shared data

Create a class called `Counter` that creates 10 threads that share a common static int variable called `counter` initially initialized to 0. Each thread, when created, will increment the variable `counter`. Print the counter each time you increment it. You should get all the numbers from 1 till 10. If you run the code many times you may get duplicate numbers. Can you explain in which case that might happen and how can we solve this potential bug?

At the end, when all the threads finish running print the value of `counter`. What should be the expected value of `counter`? What is the output of counter you got? Why? How to solve that problem?

2.3.5 Question 5: Checking priorities

Create a class called `CheckingPriorities`. In this class, create and call three different threads. Each thread will call the following function with n=45 and will print the returned value.

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

```
\item For the thread called Thread-Prio-1, give it a priority of 1.
\item For the thread called Thread-Prio-5, give it a priority of 5.
\item For the thread called Thread-Prio-9, give it a priority of 9.
```

You can use the function `setPriority` to the set the priority of a thread. Each thread should calculate `val` and print:

```
Thread: Thread-Prio-?? finished, val=??
```

Run your code many times, in which order do the threads usually finish?

From Question 1(Part 2), you can see how many CPUs your machine has. If you have 4 or more, the 3 Threads will not be competing much for CPUs, because each thread can run on its own CPU. Try to create more threads than CPUs and analyze the results.

2.3.6 Question 6: Polling with Sleep

Write a class `CheckResults` that starts two threads. One thread instantiated from the class `Counter` that will be incrementing a counter each 100MilliSec and prints

```
counter value= ??"
```

and another thread instantiated from the class `Monitor` that checks each 10 MilliSec if the counter reached 100. When the counter reaches 100, `Monitor` should print `"Reached"` and stops running. Your code should not be blocking your main function.

2.3.7 Question 7: Protecting methods with synchronized

Given the following code:

```
class Transaction extends Thread {
 BankAccount accountSender;
 BankAccount accountReceiver;
 double amount;
 public Transaction(BankAccount aS, BankAccount aR, double amount) {
  this.accountSender = aS;
  this.accountReceiver = aR;
  this.amount = amount;
 }
 @Override
 public void run() {
  System.out.println(this.getName() + " Entered");
  if (this.accountSender.getAmount()>amount) {
   this.accountSender.reduceAmount(amount);
   try {
    sleep(10);
   } catch (InterruptedException e) {

   }
   this.accountReceiver.addAmount(amount);
  }
  System.out.println(this.getName() + " Exited");
 }
}

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{
  BankAccount account1 = new BankAccount(1000);
  BankAccount account2 = new BankAccount(1000);
  Transaction t1 = new Transaction(account1, account2, 10);
  Transaction t2 = new Transaction(account1, account2, 10);
  Transaction t3 = new Transaction(account1, account2, 10);

  t1.start(); t2.start(); t3.start();
  t1.join(); t2.join(); t3.join();
  System.out.println("Account1: " + account1.getAmount());
  System.out.println("Account2: " + account2.getAmount());
 }
```

```
}
```

As you can see, we create two bank accounts, each one with 1000$ initially. We transfer 3 times 10$ from account1 to account2. What is the expected balance of the accounts (account1 and account2)? Do you get the same expected values each time you run the code? Of course you do not! Explain why. Solve this problem so that the output will always be the same with the expected values. Hint: you can create a `monitor` (an object shared by all the transactions), so that the code inside the run() function get synchronized using the object `monitor`.

3 Conclusion

In this lab you learned about Exceptions and Multi-threading. Exceptions allow us to manage efficiently alternative flows when an event occurs. They help build relied software that prevent crashes. Java is a multi-threaded language. In allows you to create concurrent threads that do compete for the CPU. Each thread will run a specific task. Those task will be executed in parallel.