07 Java Handling Exceptions

Problema 1

What is wrong with the following code?

```
class MyException extends Exception {}
public class TestClass{
   public static void main(String[] args){
      TestClass tc = new TestClass();
      try{
         tc.m1();
      }
      catch (MyException e){
         tc.m1();
      }
      finally{
         tc.m2();
      }
   }
   public void m1() throws MyException{
      throw new MyException();  // Vuelve a mandar otra excepción
                                // que se tiene que volver a atrapar
                               // El problema es con los 'checked'
   }
   public void m2() throws RuntimeException{
      throw new NullPointerException(); // 'unchecked', sin problemas
   }
}
```

You had to select 1 option

- It will not compile because you cannot throw an exception in finally block.
    - You can, but then you have to declare it in the method's throws clause.
- It will not compile because you cannot throw an exception in catch block.
    - You can, but then you have to declare it in the method's throws clause.
- It will not compile because NullPointerException cannot be created this way.
- It does have a no args constructor.
- It will not compile because of unhandled exception.
- It will compile but will throw an exception when run.

Explanation:

The catch block is throwing a checked exception (i.e. non-RuntimeException) which must be handled by either a try catch block or declared in the throws clause of the enclosing method.

Note that finally is also throwing an exception here, but it is a RuntimeException so there is no need to handle it or declare it in the throws clause.

// Como mandó otra 'exception' nueva, y esta ya no la manejo, requiere otro bloque try-catch.

Problema 2

What will the following code print when run?

```
public class Test{
static String j = "";
public static void method( int i){
  try{
   if(i == 2){
     throw new Exception();
   }
   j += "1";
  }
  catch (Exception e){
   j += "2";
   return;
  }
  finally{
   j += "3";
  }
  j += "4";
}
public static void main(String args[]){
  method(1);
  method(2);
  System.out.println(j);
}
}
```

You had to select 1 option

- 13432
- 13423
- 14324
- 12434
- 12342

Explanation:
Try to follow the flow of control :

1. in method(1) : i is not 2 so, j gets "1" then finally is executed which makes j = "13" and then the last statement (j +=4) is executed which makes j = "134".

2. in method(2) : i is 2, so it goes in the if block which throws an exception. So none of the statements of try block are executed and control goes to catch which makes j = "1342", then finally makes j = "13423" and the control is returned. Note that the last statement (j+=4) is not executed as there was an exception thrown in the try block, which cause the control to go to the catch block, which in turn has a return.

Problema 3

What will be the output of the following class.

```
class Test{
   public static void main(String[] args){
      int j = 1;
      try{
         int i = doIt() / (j = 2);
      } catch (Exception e){
         System.out.println(" j = " + j);
      }
   }
   public static int doIt() throws Exception {
     throw new Exception("FORGET IT");  }
}
```

You had to select 1 option

- It will print j = 1;
- It will print j = 2;
- The value of j cannot be determined.
- It will not compile.
- None of the above.

Explanation:

If evaluation of the left-hand operand of a binary operator completes abruptly, no part of the right-hand operand appears to have been evaluated.

So, as `doIt()` throws exception, `j = 2` never gets executed.

Problema 4

Java's Exception mechanism helps in which of the following ways?

You had to select 2 options

- It allows creation of new exceptions that are custom to a particular application domain.
    - You can define your own exceptions based on your application business domain. For example, in a banking application, you might want to create a InsufficientFundsException.
    - This increases code clarity as compared to having a single (or a few standard) exception class(es) and looking at the exception code to determine what happened.
- It improves code because error handling code is clearly separated from the main program logic.
    - The error handling logic is put in the catch block, which makes the main flow of the program clean and easily understandable.
- It enhances the security of the application by reporting errors in the logs.
    - Exception handling as such has nothing to do with the security of the application but good exception handling in an application can prevent security holes.
- It improves the code because the exception is handled right at the place where it occured.
    - Just the opposite is true. It improves the code because the code does not have to include error handling code if it is not capable of handling it. It can propagate the exception up the chain and so that the exception can be handled somewhere at a more appropriate place.
- It provides a vast set of standard exceptions that covers all possible exceptions.
    - Although it does provide a vast set of standard exceptions, they cannot cover all scenarios. But you can always create new exceptions tailored for your application.

Problema 4.5 (Conceptos)

Which of the following are standard Java exception classes?
You had to select 2 options

- `java.io.FileNotFoundException`
- `java.io.InputException`
    - There is an java.io.IOException but no InputException or OutputException.
- `java.lang.CPUError`
    - There is no such class.
- `java.lang.MemoryException`
    - There is a java.lang.OutOfMemoryError but no MemoryException. There is also a java.lang.StackOverflowError.
- `java.lang.SecurityException`
    - Java has a java.lang.SecurityException. This exception extends RuntimeException

Problema 5

What will be the output when the following program is run?

```
package exceptions;
public class TestClass{
    public static void main(String[] args) {
        try{
            hello();
        }
        catch(MyException me){
            System.out.println(me);   //imprime mensaje
        }
    }
    static void hello() throws MyException{
        int[] dear = new int[7];
        dear[0] = 747;   // Esta bien
        foo();
    }
    static void foo() throws MyException{
        throw new MyException("Exception from foo"); // Mensaje to print
    }
}

class MyException extends Exception {
    public MyException(String msg){
        super(msg); // manda mensaje, Constructor String
    }
}
```

(Assume that line numbers printed in the messages given below are correct.)

You had to select 1 option

- ```
  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at exceptions.TestClass.doTest(TestClass.java:24)
        at exceptions.TestClass.main(TestClass.java:14)
  ```
  - You are creating an array of length 7. Since array numbering starts with 0, the first element would be array[0]. So ArrayIndexOutOfBoundsException will NOT be thrown.
- Error in thread "main" java.lang.ArrayIndexOutOfBoundsException
  - java.lang.ArrayIndexOutOfBoundsException extends java.lang.RuntimeException, which in turn extends java.lang.Exception.
  - Therefore, ArrayIndexOutOfBoundsException is an Exception and not an Error.
- `exceptions.MyException: Exception from foo`
- ```
  exceptions.MyException: Exception from foo
        at exceptions.TestClass.foo(TestClass.java:29)
        at exceptions.TestClass.hello(TestClass.java:25)
        at exceptions.TestClass.main(TestClass.java:14)
  ```
  - me.printStackTrace() would have produced this output.

Explanation:

// Note that there are a few questions in the exam that test your knowledge about how exception messages are printed.

When you use `System.out.println(exception)`, a stack trace is not printed. Just the name of the exception class and the message is printed.
When you use `exception.printStackTrace()`, a complete chain of the names of the methods called, along with the line numbers, is printed. It contains the names of the methods in the chain of method calls that led to the place where the exception was created going back up to the point where the thread, in which the exception was created, was started.

Problema 5 V2

What will be the output when the following program is run?

```
package exceptions;
public class TestClass {
    public static void main(String[] args) {
        try{
            doTest();
        }
        catch(MyException me){
            System.out.println(me);
        }
    }

    static void doTest() throws MyException{
        int[] array = new int[10];
        array[10] = 1000;  // Out of Index
        doAnotherTest();
    }

    static void doAnotherTest() throws MyException{
        throw new MyException("Exception from doAnotherTest");
    }
}
class MyException extends Exception {
    public MyException(String msg){
     super(msg);
    }
}
```

(Assume that there is no error in the line numbers given in the options.)

You had to select 1 option

- `Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10`
  `at exceptions.TestClass.doTest(TestClass.java:14)`
  `at exceptions.TestClass.main(TestClass.java:5)`
  - You are creating an array of length 10. Since array numbering starts with 0, the last element would be array[9]. array[10] would be outside the range of the array and therefore an ArrayIndexOutOfBoundsException will be thrown, which cannot be caught by catch(MyException) clause.
  - The exception is thus thrown out of the main method and is handled by the JVM's uncaught exception handling mechanism, which prints the stack trace.
- `Error in thread "main" java.lang.ArrayIndexOutOfBoundsException`
  - java.lang.ArrayIndexOutOfBoundsException extends java.lang.IndexOutOfBoundsException, which extends java.lang.RuntimeException, and which in turn extends java.lang.Exception. Therefore, ArrayIndexOutOfBoundsException is an Exception and not an Error.
- `exceptions.MyException: Exception from doAnotherTest`
- `exceptions.MyException: Exception from doAnotherTest`
  `at exceptions.TestClass.doAnotherTest(TestClass.java:29)`
  `at exceptions.TestClass.doTest(TestClass.java:25)`
  `at exceptions.TestClass.main(TestClass.java:14)`

Exception:

When you use `System.out.println(exception)`, a stack trace is not printed. Just the name of the exception class and the message is printed.
When you use `exception.printStackTrace()`, a complete chain of the names of the methods called, along with the line numbers, is printed. It contains the names of the methods in the chain of method calls that led to the place where the exception was created going back up to the point where the thread, in which the exception was created, was started.

El `printStackTrace()` en este caso fue hecho por el mismo JVM, es necesario ponerlo para imprimirlo en el resultado usandolo como método.

Problema 6

What will the following code print when compiled and run?

```
abstract class Calculator{
   abstract void calculate();
   public static void main(String[] args){
      System.out.println("calculating");
      Calculator x = null;
      x.calculate();
   }
}
```

You had to select 1 option

- It will not compile.
    - It will compile without any issue.
- It will not print anything and will throw NullPointerException
- It will print calculating and then throw NullPointerException.
    - After printing, when it tries to call calculate() on x, it will throw NullPointerException since x is null.
- It will print calculating and will throw NoSuchMethodError
- It will print calculating and will throw MethodNotImplementedException

Problema 7 -> Revisar

What will the following code print when run?

```
public class Test {

    static String s = "";

    public static void m0(int a, int b) {
        s += a;  // 1
        m2();  // Llama método
        m1(b);
    }

    public static void m1(int i) {
        s += i;
    }

    public static void m2() {
        throw new NullPointerException("aa");  //Rompe el programa 'unchecked'
    }

    public static void m() {
        m0(1, 2);
        m1(3);
    }

    public static void main(String args[]) {
        try {
            m();
        } catch (Exception e) {
        }
        System.out.println(s);
    }
}
```

You had to select 1 option

- 1
- 12
- 123
- 2
- It will throw exception at runtime.

Explanation:

Try to follow the control flow:

1. `m()` calls `m0(1, 2)`.
2. `m0(1, 2)` first executes `s += 1` (so s is now 1) and then calls `m2()`.
3. Now, `m2()` throws an exception which is not caught by `m2()` so it is propagated back to `m0(1, 2)`. Since, within m0 method, the call to `m2()` is not wrapped in a try catch block, this exception further propagates up to `m()`. (The next line in `m0(1, 2)`, which is `m1(2)`, is not executed).
4. Again, `m()` also does not have the try catch block so the same exception is further propagated up to the `main()` method. (The next line in `m()`, which is a call to `m1(3)` is not called).
4. In main method, the call to `m()` is wrapped in a try catch block and so the exception is handled here.
5. Finally, s stays as "1".

The point to note here is that if you do not catch an exception, it is propagated up the stack of method calls until it is handled. If nobody handles it, the JVM handles that exception and kills the thread. If that thread is the only user thread running, the program ends

Problema 8

Which exact exception class will the following class throw when compiled and run?

```
class Test{
   public static void main(String[] args) throws Exception{
      int[] a = null;
      int i = a [ m1() ];  // Tiene que arreglar primero lo interno, el
método
   }
   public static int m1() throws Exception{
      throw new Exception("Some Exception");
   }
}
```

You had to select 1 option

- NullPointerException
- ArrayIndexOutOfBoundsException
- Exception
- RuntimeException

Explanation:

A `NullPointerException` never occurs because the index expression must be completely evaluated before any part of the indexing operation occurs, and that includes the check as to whether the value of the left-hand operand is null.
If the array reference expression produces `null` instead of a reference to an array, then a `NullPointerException` is thrown at runtime, but only after all parts of the array reference expression have been evaluated and only if these evaluations completed normally.

In an array access, the expression to the left of the brackets appears to be fully evaluated before any part of the expression within the brackets is evaluated.
Note that if evaluation of the expression to the left of the brackets completes abruptly, no part of the expression within the brackets will appear to have been evaluated.

Here, `m1()` is called first, which throws Exception and so `a` is never accessed and `NullPointerException` is never thrown.

Meaning of "the expression to the left of the brackets appears to be fully evaluated":

Consider this line of code:
`Object o = getArray()[getIndex()];`
(Assume that the method getArray returns an object array and getIndex returns an int)

1. The expression to the left of the brackets is evaluated first. So, first `getArray()` will be called. This gives the reference to the array. It could be `null` as well. (But if the `getArray()` method throws an exception, the rest of the statement will not be evaluated i.e. `getIndex()` will not be called.)

2. Now, the expression in the brackets is evaluated, therefore, `getIndex()` will be called, which will give you the index. If `getIndex()` throws an exception then step 3 will not happen.

3. Finally, the index will be applied on the array (whose reference was returned in step 1). At this time, if the array reference is `null`, you will get a `NullPointerException`.

Problema 9

Checked exceptions are meant for…
You had to select 1 option

- exceptional conditions external to an application that a well written application should anticipate and from which it can recover.
    - Note that here recovery doesn't necessarily mean to keep functioning normally. It means that the program shouldn't just crash. If it absolutely cannot proceed, it should notify the user appropriately and then end gracefully.
- exceptional conditions external to the program that a well written program cannot anticipate but should recover from.
- exceptional conditions from which recovery is difficult or impossible.
    - Errors are meant for this purpose.
- exceptional situations internal to an application that the application can anticipate but cannot recover from.
    - Generally, if the exception is caused by problems internal to the program, a RuntimeException is used.

Explanation:
There are multiple view points regarding checked and and unchecked exceptions. As per the official Java tutorial (http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) : If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.
Here, the client basically means the caller of a method.

Another way to look at exceptions is to see the cause of the exception in terms of whether it is internal or external to the program's code. For example, an incorrectly written code may try to access a reference pointing to null, or it may try to access an array beyond its length. These are internal sources of exception. Here, using runtime exceptions is appropriate because ideally these problems should be identified while testing and should not occur when the program is ready for deployment.

On the other hand, a program interacting with files may not be able to do its job at all if a file is not available but it should anticipate this situation. This is an external source of an exception and has nothing to do with a program's code as such. It is therefore appropriate to use a checked exception here.

Problema 10

Given that SomeException is a checked exception, consider the following code:

```
//in file A.java
public class A{
   protected void m() throws SomeException{}
}

//in file B.java
public class B extends A{
   public void m(){ }
}

//in file TestClass.java
public class TestClass{
   public static void main(String[] args){
      //insert code here. //1
   }
}
```

Which of the following options can be inserted at //1 without resulting in any compilation or runtime errors?

You had to select 1 option

- ```
  B b =  new A();
  b.m();
  ```
  - B b = new A(); is not valid because a superclass object can never be assigned to a base class reference.
- ```
  A a = new B();
  a.m();  //g.play() en ese problema entra ene ste conflicto
  ```
  - A's m() declares 'throws SomeException', which is a checked exception. But the main() method doesn't declare any throws clause that can capture this exception. So, the call to a.m() must be wrapped within a try/catch block.
- ```
  A a = new B();
  ( ( B) a ).m();  // Evito el conflicto por el cast
  ```
  - Due to explicit casting of 'a' to B, the compiler knows that 'a' will point to an object of class B (or its subclass), whose method m() does not throw an exception. So there is no need for a try catch block here.
- ```
  Object o = new B();
  o.m();
  ```
  - Object class does not have method m(). So o.m() will not compile. You can do ((B) o).m();
- None of these.

Problema 11

What will be the result of attempting to compile and run the following program?

```
class TestClass{
   public static void main(String args[]){
      int i = 0;
      loop :          // 1
      {
         System.out.println("Loop Lable line");
         try{
            for (  ;  true ;  i++ ){
               if( i >5) break loop;        // 2
            }
         }
         catch(Exception e){
            System.out.println("Exception in loop.");
         }
         finally{
            System.out.println("In Finally");       // 3
         }
      }
   }
}
```

You had to select 1 option

- Compilation error at line 1 as this is an invalid syntax for defining a label.
    - You can apply a label to any code block or a block level statement (such as a for statement) but not to declarations. For example: `loopX : int i = 10;`
- Compilation error at line 2 as 'loop' is not visible here.
- No compilation error and line 3 will be executed.
    - Even if the break takes the control out of the block, the finally clause will be executed.
- No compilation error and line 3 will NOT be executed.
- Only the line with the label loop will be printed.

Explanation:

A `break` without a label breaks the current loop (i.e. no iterations any more) and a break with a label tries to pass the control to the given label.
'Tries to' means that if the break is in a try block and the try block has a finally clause associated with it then it will be executed.

Problema 12

What will be the output of the following program:

```
public class TestClass{
   public static void main(String args[]){
      try{
         m1(); // Imprime m1 Starts
      }catch(IndexOutOfBoundsException e){  // Este atrapa
         System.out.println("1");  //2 imprime "1"
         throw new NullPointerException();  // lo arroja dentro de este
catch
                                            // Aborará al programa
      }catch(NullPointerException e){  // pertenece al try de m1()
         System.out.println("2");
         return;
      }catch (Exception e) {          // pertenece al try de m1()
         System.out.println("3");
      }finally{                    //Si o si pasa por aqui
         System.out.println("4");  //3 imprime "4"
      }
      System.out.println("END");    // NO imprime por aborto del primer catch
   }

   static void m1(){
      System.out.println("m1 Starts");  //1 imprime "m1 Starts"
      throw new IndexOutOfBoundsException( "Big Bang " );  //unchecked
   }
}
```

You had to select 3 options

- The program will print m1 Starts.
- The program will print m1 Starts, 1 and 4, in that order.
- The program will print m1 Starts, 1 and 2, in that order.
- The program will print m1 Starts, 1, 2 and 4 in that order.
- END will not be printed.
-

Explanation:

The code has a try block and the try block is associated with three catch clauses. All of the catch clauses are at the same level and are meant to catch exceptions thrown only from that try block. They cannot catch any exception thrown from anywhere else in the code (including the catch blocks themselves).

The `m1()` method prints `m1 Starts` and then throws an `IndexOutOfBoundsException`. This exception is caught by the `catch(IndexOutOfBoundsException e)` clause, so the control goes inside this catch block. Inside the catch block, `1` is printed and a `NullPointerException` is thrown.

Since this exception is not thrown inside any try block, it will not be caught. It will remain "unhandled" by this method and will be propagated to the caller of the `main()` method. But before the exception is sent to the caller, the `finally` block is executed, which causes `4` to be printed.

The code that prints `END` is never reached because of the uncaught `NullPointerException`. This exception will finally be handled by the JVM itself, which prints the stack trace associated with the exception and exits.

Problema Leo

What will be the result of compiling and running the following program ?

```
class NewException extends Exception {}
class AnotherException extends Exception {}
public class ExceptionTest{
   public static void main(String [] args) throws Exception{
      try{
         m2();
      }
      finally{ m3(); }   // Se quedará con la última excepción,Si se hace un
                         // árbol con otro try-final dentro de aqui
                         // seguirá siendo el método m3();
                         // si el árbol fuera de catch, en lugar de final
                         // sigue cancelando en el primer catch
    }
    public static void m2() throws NewException{  throw new NewException();
}
    public static void m3() throws AnotherException{  throw new
AnotherException();  }
}
```

You had to select 1 option

- It will compile but will throw AnotherException when run.
- It will compile but will throw NewException when run.
- It will compile and run without throwing any exceptions.
- It will not compile.
- None of the above.

Explanation:
m2() throws NewException, which is not caught anywhere. But before exiting out of the main method, `'finally'` must be executed.
Since `'finally'` throw AnotherException (due to a call to m3()), the NewException thrown in the try block (due to call to m2()) is ignored and AnotherException is thrown from the main method.

Problema Leo v2

What will be the result of compiling and running the following program ?

```
class NewException extends Exception {}

class AnotherException extends Exception {}

public class ExceptionTest{
    public static void main(String[] args) throws Exception{
        try{
            m2();
        }
        finally{
            m3();
        }
        catch (NewException e){}
    }

    public static void m2() throws NewException { throw new NewException();
}

    public static void m3() throws AnotherException{ throw new
AnotherException(); }

}
```

You had to select 1 option

- It will compile but will throw AnotherException when run.
- It will compile but will throw NewException when run.
- It will compile and run without throwing any exceptions.
- It will not compile.
    - Because a catch block cannot follow a finally block!
- None of the above.

Problema Leo v3

What will the following code print when compiled and run?
(Assume that MySpecialException is an unchecked exception.)

```
1. public class ExceptionTest {
2.    public static void main(String[] args) {
3.        try {
4.            doSomething();
5.        } catch (MySpecialException e) {
6.            System.out.println(e);
7.        }
8.    }
9.
10.    static void doSomething() {
11.        int[] array = new int[4];
12.        array[4] = 4;  // Fuera del Array
13.        doSomethingElse();
14.    }
15.
16.    static void doSomethingElse() {
17.        throw new MySpecialException("Sorry, can't do something else");
18.    }
}
```

You had to select 1 option

- It will not compile.
- ```
  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at ExceptionTest.doSomething(ExceptionTest.java:12)
        at ExceptionTest.main(ExceptionTest.java:4)
  ```
- ```
  Exception in thread "main" MySpecialException: 4
        at ExceptionTest.doSomethingElse(ExceptionTest.java:17)
        at ExceptionTest.doSomething(ExceptionTest.java:12)
        at ExceptionTest.main(ExceptionTest.java:4)
  ```
- ```
  Exception in thread "main" MySpecialException: Sorry, can't do something else
        at ExceptionTest.doSomethingElse(ExceptionTest.java:17)
        at ExceptionTest.doSomething(ExceptionTest.java:12)
        at ExceptionTest.main(ExceptionTest.java:4)
  ```
- ```
  Exception in thread "main" MySpecialException: Sorry, can't do something else
        at ExceptionTest.doSomethingElse(ExceptionTest.java:17)
        at ExceptionTest.doSomething(ExceptionTest.java:13)
        at ExceptionTest.main(ExceptionTest.java:4)
  ```

Explanation:
Since the length of array is only 4, you can't do `array[4]`, because that would access the 5th element.

Problema 13

What two changes can you do, independent of each other, to make the following code compile:

```
//assume appropriate imports
class PortConnector {

    public PortConnector(int port) {
        if (Math.random() > 0.5) {
            throw new IOException();
        }

        throw new RuntimeException();
    }
}

public class TestClass {

    public static void main(String[] args) {
        try {
            PortConnector pc = new PortConnector(10);
        } catch (RuntimeException re) {
            re.printStackTrace();
        }
    }
}
```

You had to select 2 options

- add throws IOException to the main method.
- add throws IOException to PortConnector constructor.
- add throws IOException to the main method as well as to PortConnector constructor.
- Change RuntimeException to java.io.IOException.
- add throws Exception to PortConnector constructor and change catch(`RuntimeException re`) to catch(`Exception re`) in the main method.

Explanation:
`IOException` is a checked exception and since the `PortConnector` constructor throws `IOException`, this exception (or its superclass) must be present in the throws clause of the constructor.

Now, the `main` method has two options, either catch `IOException` (or whatever exception PortConnector throws) in its catch block (i.e. option 5) or put that exception in its throws clause (i.e. option 3).

Problema 14

What will be the output of the following program?

```
class TestClass{
   public static void main(String[] args) throws Exception{
      try{
         amethod();
         System.out.println("try ");
      }
      catch(Exception e){
         System.out.print("catch ");
      }
      finally   {
         System.out.print("finally ");
      }
      System.out.print("out ");
   }
   public static void amethod(){ }
}
```

You had to select 1 option

- try finally
- try finally out
- try out
- catch finally out
- It will not compile because amethod() does not throw any exception.

Explanation:

Since the method `amethod()` does not throw any exception, `try` is printed and the control goes to finally which prints `finally`. After that `out` is printed.

Problema 15

Considering the following program, which of the options are true?

```
public class FinallyTest{
   public static void main(String args[]){
      try{
          if (args.length == 0) return;
          else throw new Exception("Some Exception");
      }
      catch(Exception e){
          System.out.println("Exception in Main");
      }
      finally{
          System.out.println("The end");
      }
   }
}
```

You had to select 2 options

- If run with no arguments, the program will only print 'The end'.
- If run with one argument, the program will only print 'The end'.
- If run with one argument, the program will print 'Exception in Main' and 'The end'.
- If run with one argument, the program will only print 'Exception in Main'.
- If run with no arguments, the program will not print anything.
- If run with no arguments, the program will generate a stack trace on the console.

Explanation:
There are two points to understand here:
1. Even if the program is executed without any arguments, the 'args' is NOT NULL. In such case it will be initialized to an array of Strings containing zero elements.
2. The finally block is always executed, no matter how control leaves the try block. Only if, in a try or catch block, System.exit() is called then finally will not be executed.

Problema 16

What is the result of compiling and running this code?

```
class MyException extends Throwable{}
class MyException1 extends MyException{}
class MyException2 extends MyException{}
class MyException3 extends MyException2{}
public class ExceptionTest{
   void myMethod() throws MyException{
      throw new MyException3();
   }
   public static void main(String[] args){
      ExceptionTest et = new ExceptionTest();
      try{
         et.myMethod();
      }
      catch(MyException me){
         System.out.println("MyException thrown");
      }
      catch(MyException3 me3){
         System.out.println("MyException3 thrown");
      }
      finally{
         System.out.println(" Done");
      }
   }
}
```

You had to select 1 option

- MyException thrown
- MyException3 thrown
- MyException thrown Done
- MyException3 thrown Done
- It fails to compile

Explicación

// Jerarquia u orden de las excepciones, no pasará por catch de MyException3

You can have multiple catch blocks to catch different kinds of exceptions, including exceptions that are subclasses of other exceptions. However, the catch clause for more specific exceptions (i.e. a SubClassException) should come before the catch clause for more general exceptions (i.e. a SuperClassException). Failure to do so results in a compiler error as the more specific exception is unreachable.

In this case, catch for `MyException3` cannot follow catch for `MyException` because if `MyException3` is thrown, it will be caught by the catch clause for `MyException`. And so, there is no

way the catch clause for `MyException3` can ever execute. And so it becomes an unreachable statement.

Problema 17

Consider the following hierarchy of Exception classes :

```
java.lang.RuntimeException
  +---- IndexOutOfBoundsException
          +----ArrayIndexOutOfBoundsException,
StringIndexOutOfBoundsException
```

Which of the following statements are correct for a method that can throw ArrayIndexOutOfBounds as well as StringIndexOutOfBounds Exceptions but does not have try catch blocks to catch the same?

You had to select 3 options

- The method calling this method will either have to catch these 2 exceptions or declare them in its throws clause.
- It is ok if it declares just throws ArrayIndexOutOfBoundsException
- It must declare throws ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException
- It is ok if it declares just throws IndexOutOfBoundsException
- It does not need to declare any throws clause.

Explanation:
// Runtime, no importa si los pones, está bien si solo pones uno y puede ser cualquiera, no necesario.

Note that both the exceptions are `RuntimeExceptions` so there is no need to catch these. But it is ok even if the method declares them explicitly.

Problema 18

What is wrong with the following code written in a single file named TestClass.java?

```
class SomeThrowable extends Throwable { }
class MyThrowable extends SomeThrowable { }
public class TestClass{
   public static void main(String args[]) throws SomeThrowable{
      try{
         m1();
      }catch(SomeThrowable e){
         throw e;
      }finally{
         System.out.println("Done");
      }
   }
   public static void m1() throws MyThrowable{
      throw new MyThrowable();
   }
}
```

You had to select 1 option

- The main declares that it throws SomeThrowable but throws MyThrowable.
  - That's OK. You can put a Super class in the throws clause and then you can throw any subclass exception.
- You cannot have more than 2 classes in one file.
  - You sure can. The only limitation is you can have only one top level public class in a file.
- The catch block in the main method must declare that it catches MyThrowable rather than SomeThrowable.
  - You can catch a subclass exception in the catch clause that catches a super class.
- There is nothing wrong with the code and Done will be printed.
  - Done will be followed by an exception. Finally is always executed (Only exception is System.exit();)

// Si vuelve a lanzar la exception. Entonces main si necesita atrapar excepciones.
// En este programa no hay algo después del try-catch-finally por lo que es

Problema 19

What will be the result of attempting to compile and run the following program?

```
public class TestClass{
   public static void main(String args[]){
      Exception e = null;
      throw e;
   }
}
```

You had to select 1 option

- The code will fail to compile.
- The program will fail to compile, since it cannot throw a null.
- The program will compile without error and will throw an Exception when run.
- The program will compile without error and will throw java.lang.NullPointerException when run
- The program will compile without error and will run and terminate without any output.

Explanation:

The main method is throwing a checked exception but there is no try/catch block to handle it and neither is there a throws clause that declares the checked exception. So, it will not compile.
If you either put a try catch block or declare a throws clause for the method then it will throw a `NullPointerException` at run time because e is `null`.
A method that throws a "checked" exception i.e. an exception that is not a subclass of `Error` or `RuntimeException`, either must declare it in throws clause or put the code that throws the exception within a try/catch block.

Problema 20

Consider the following code...

```
class MyException extends Exception {}

public class TestClass{
     public void myMethod() throws XXXX{
         throw new MyException();
     }
}
```

What can replace XXXX?

You had to select 3 options

- MyException
- Exception
    - Because Exception is a superclass of MyException.
- No throws clause is necessary
    - It is needed because MyException is a checked exception. Any exception that extends java.lang.Exception but is not a subclass of java.lang.RuntimeException is a checked exception.
- Throwable
    - Because Throwable is a super class of Exception.
- RuntimeException

Problema 21

Given the class

```
// Filename: Test.java
public class Test{
   public static void main(String args[]){
      for(int i = 0; i< args.length; i++){
         System.out.print("  "+args[i]);
      }
   }
}
```

Now consider the following 3 options for running the program:

a: java Test

b: java Test param1

c: java Test param1 param2

Which of the following statements are true?

You had to select 2 options

- The program will throw java.lang.ArrayIndexOutOfBoundsException on option a.
- The program will throw java.lang.NullPointerException on option a.
- The program will print Test param1 on option b.
    - Unlike in C++, Name of the file is not passed in args because for a public class it is always same as the name of the class.
- It will print param1 param2 on option c.
- It will not print anything on option a.

Explanation:

It will not throw `NullPointerException` because `args[]` is never `null`. If no argument is given (as in option a) then the length of args is 0.

Problema 22

What will be the output when the following code is compiled and run?

```
//in file Test.java
class E1 extends Exception{ }
class E2 extends E1 { }
class Test{
   public static void main(String[] args){
      try{
         throw new E2();
      }
      catch(E1 e){
         System.out.println("E1");
      }
      catch(Exception e){
         System.out.println("E");
      }
      finally{
         System.out.println("Finally");
      }
   }
}
```

You had to select 1 option

- It will not compile.
- It will print E1 and Finally.
- It will print E1, E and Finally.
- It will print E and Finally.
- It will print Finally.

Explanation:

Since E2 is a sub class of E1, `catch(E1 e)` will be able to catch exceptions of class E2. Therefore, E1 is printed. Once the exception is caught the rest of the catch blocks at the same level (that is the ones associated with the same try block) are ignored. So E is not printed. finally is always executed (except in case of `System.exit()`), so Finally is also printed.

Problema 23

What will the following class print ?

```
class Test{
   public static void main(String[] args){
      int[][] a = { { 00, 01 }, { 10, 11 } };
      int i = 99;
      try {
         a[val()][i = 1]++;  // Sale por error antes de asignar
      } catch (Exception e) {
         System.out.println( i+", "+a[1][1]);
      }
   }
   static int val() throws Exception {
     throw new Exception("unimplemented");
   }
}
```

You had to select 1 option

- 99 , 11
- 1 , 11
- 1 and an unknown value.
- 99 and an unknown value.
- It will throw an exception at Run time.

Explanation:

If evaluation of a dimension expression completes abruptly, no part of any dimension expression to its right will appear to have been evaluated.
Thus, while evaluating `a[val()][i=1]++`, `val()` throws an exception and `i=1` will not be executed. Therefore, `i` remains `99` and `a[1][1]` will print `11`.

Problema 24

Identify the exceptions that will be received when the following code snippets are executed.

```
1. int factorial(int n){
       if(n==1) return 1;
      else return n*factorial(n-1);
   }
```

Assume that it is called with a very large integer. // StackOverflow

```
2. void printMe(Object[] oa){
       for(int i=0; i<=oa.length; i++)
       System.out.println(oa[i]);
    }
```

Assume that it is called as such: printMe(null); //NullPointer, como el que daba el valor de e=null

```
3. Object m1(){
       return new Object();
    }
    void m2(){
       String s = (String) m1();
    }
```

Assume that method m2 is invoked. // Compila pero error por cast en runtime por JVM

You had to select 1 option

ClassCastException	ClassCastException
ArrayIndexOutOfBoundsException	ArrayIndexOutOfBoundsException
StackOverflowError	SecurityException
No Exception Will Be Thrown	StackOverflowError
SecurityException	NullPointerException
Will Not Compile	No Exception Will Be Thrown
StackOverflowError	StackOverflowError
ArrayIndexOutOfBoundsException	NullPointerException
ClassCastException	NullPointerException
SecurityException	StackOverflowError
NullPointerException	NullPointerException
No Exception Will Be Thrown	ClassCastException

Problema 25

Which of the following standard java exception classes extend java.lang.RuntimeException?
You had to select 4 options

`java.lang.SecurityException`
SecurityException extends RuntimeException: It is thrown by the security manager upon security violation. For example, when a java program runs in a sandbox (such as an applet) and it tries to use prohibited APIs such as File I/O, the security manager throws this exception. Since this exception is explicitly thrown using the new keyword by a security manager class, it can be considered to be thrown by the application programmer.

`java.lang.ClassCastException`
ClassCastException extends RuntimeException: Usually thrown by the JVM. Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a ClassCastException: Object x = new Integer(0); System.out.println((String)x);

`java.lang.NullPointerException`
NullPointerException extends RuntimeException: Usually thrown by the JVM. Thrown when an application attempts to use null in a case where an object is required. These include: Calling the instance method of a null object. Accessing or modifying the field of a null object. Taking the length of null as if it were an array. Accessing or modifying the slots of null as if it were an array. Throwing null as if it were a Throwable value. Applications should throw instances of this class to indicate other illegal uses of the null object.

`java.lang.CloneNotSupportedException`
public class CloneNotSupportedException extends Exception.
Thrown to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface. Applications that override the clone method can also throw this exception to indicate that an object could not or should not be cloned.

`java.lang.IndexOutOfBoundsException`
IndexOutOfBoundsException extends RuntimeException: Usually thrown by the JVM. Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range. Applications can subclass this class to indicate similar exceptions. ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException both extend IndexOutOfBoundsException.

The other two exceptions you should know about are:

- `IllegalArgumentException` extends `RuntimeException`.
- `IllegalStateException extends RuntimeException`:

Problema 26

Which statements regarding the following code are correct ?

```
class Base{
   void method1() throws java.io.IOException, NullPointerException{
      someMethod("arguments");
      // some I/O operations
   }
   int someMethod(String str){
      if(str == null) throw new NullPointerException();
      else return str.length();
   }
}
public class NewBase extends Base{
      void method1(){
           someMethod("args");
      }
}
```

You had to select 2 options

- method1 in class NewBase does not need to specify any exceptions.
- The code will not compile because RuntimeExceptions cannot be specified in the throws clause.
    - Any Exception can be specified in the throws clause.
- method1 in class NewBase must at least specify IOException in its throws clause.
- method1 in class NewBase must at least specify NullPointerException in its throws clause.
    - This is not needed because NullPointerException is a RuntimeException.
- There is no problem with the code.

Explanation:

Overriding method only needs to specify a subset of the list of exception classes the overridden method can throw. A set of no classes is a valid subset of that list.

Remember that `NullPointerException` is a subclass of `RuntimeException`, while `IOException` is a subclass of `Exception`.

Problema 27

Following is a supposedly robust method to parse an input for a float :

```
public float parseFloat(String s){
   float f = 0.0f;
   try{
      f = Float.valueOf(s).floatValue();
      return f ;
   }
   catch(NumberFormatException nfe){
      System.out.println("Invalid input " + s);
      f = Float.NaN ;
      return f;
   }
   finally { System.out.println("finally");  }
   return f ;
}
```

Which of the following statements about the above method is/are true?
You had to select 1 option

- If input is "0.1" then it will return 0.1 and print finally.
- If input is "0x.1" then it will return Float.NaN and print Invalid input 0x.1 and finally.
- If input is "1" then it will return 1.0 and print finally.
- If input is "0x1" then it will return 0.0 and print Invalid input 0x1 and finally.
- The code will not compile.
    - Note that the return statement after finally block is unreachable. Otherwise, if this line were not there, choices 1, 2, 3 are valid.

Problema 28

What will the following code snippet print:

```
Float f = null;
try{
   f = Float.valueOf("12.3");
   String s = f.toString();
   int i = Integer.parseInt(s);
   System.out.println(""+i);
}
catch(Exception e){
   System.out.println("trouble : "+f);
}
```

You had to select 1 option

- 12
- 13
- trouble : null
- trouble : 12.3
- trouble : 0.0

Explanation:
`f = Float.valueOf("12.3");` executes without any problem.
`int i = Integer.parseInt(s);` throws a `NumberFormatException` because `12.3` is not an integer.
Thus, the catch block prints trouble : `12.3`

Problema 29

What will the following code print?

```
public class Test {
    public int luckyNumber(int seed) {
        if (seed > 10) {
            return seed % 10;
        }

        int x = 0;
        try {
            if (seed % 2 == 0) {
                throw new Exception("No Even no.");
            } else {
                return x;
            }
        } catch (Exception e) {
            return 3;       <= regresa qui?
        } finally {
            return 7;       <= regresa aqui?    gana final, regresa '7'
        }
    }

    public static void main(String args[]) {
        int amount = 100, seed = 6;
        switch (new Test().luckyNumber(6)) {
            case 3:
                amount = amount * 2;
            case 7:
                amount = amount * 2; //evita el 3, 100 * 2 = 200
            case 6:                  // No hay break, entonces pasa por este case
                amount = amount + amount;    // 200 + 200 = 400
            default:
        }
        System.out.println(amount);    // print 400
    }
}
```