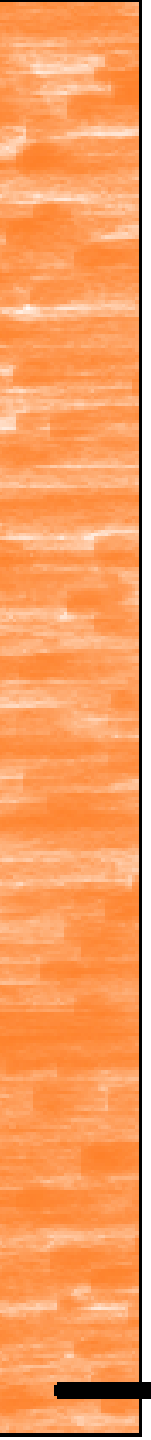


WEB APPLICATION DEVELOPMENT

PROGRAMMING

Exception handling in Java.

1. Exceptions. Concept.
2. Hierarchy of exceptions.
3. Exception handling.
    3.1. Exception capture.
    3.2. Propagate exceptions.
    3.3. Throwing exceptions.
4. Methods of the Exception class.

1. Exceptions.

- One of the most important problems in writing programs is error handling.
- EXCEPTIONS are the possible errors that may occur during the execution of the program, due to extraordinary situations that may occur.
- The situations that cause a **runtime** error to occur can be quite varied:
  - The program asks for a number and the user inserts letters.
  - Divide by zero.
  - We are trying to access an out-of-bounds position in a static array.
  - You want to open a file that does not exist.
  - Make use of an object, which does not point to any memory space.
  - The user attempts to save a file in a write-protected directory.

1. Exceptions.

- If the **exception** is not caught, the execution of the program will stop, so we must program with this type of situation in mind.
- To do this, we would control the code susceptible to execution errors so that, if an **exception** occurs, a block of code prepared by the programmer would be executed to handle it.

1. Exceptions.

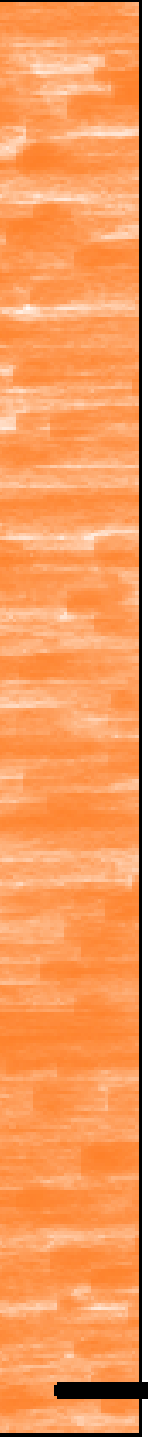
- **Example:** a user enters a letter when prompted for a number.

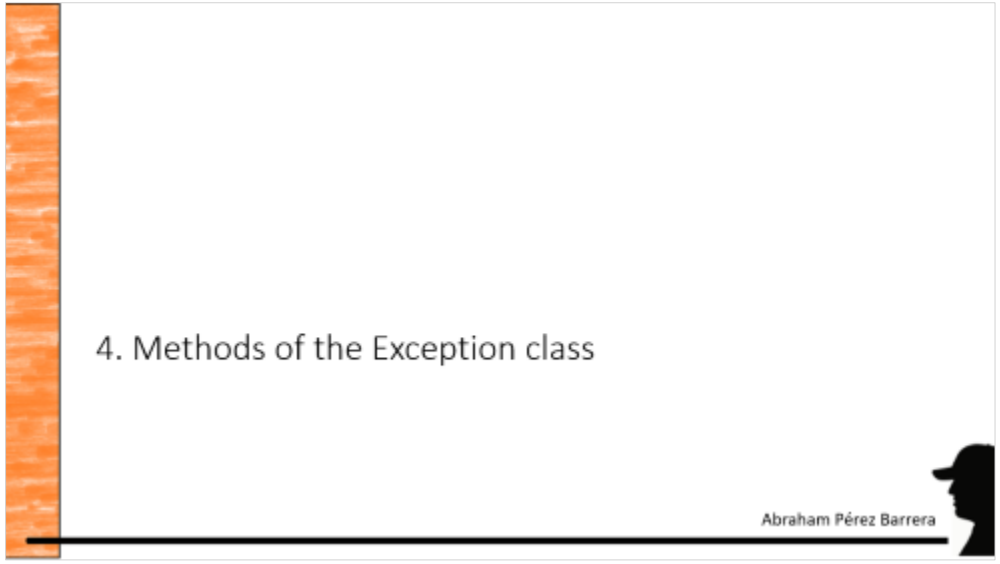
  If we do not control it, when the user inserts letters, the program will close unexpectedly due to an execution error.

  How do we control it?

  a) Closing the program by displaying an error message.

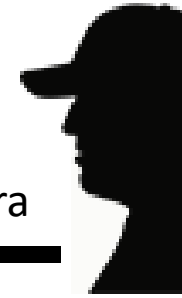
  b) Allowing the user to re-insert the number again.




4. Methods of the Exception class
Abraham Pérez Barrera

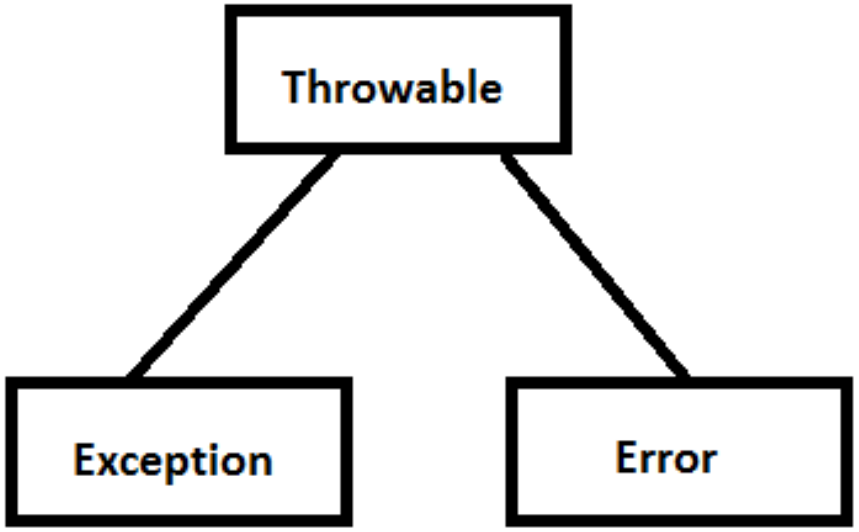
Exceptions hierarchy

Abraham Pérez Barrera



2. Hierarchy of exceptions.

- All exceptions (errors) that can occur in a program are represented by classes derived from the **Throwable class.**
- This **class** gives rise to two groups of hierarchies: those derived from the **Error class** and those derived from the **Exception class**.



2. Hierarchy of exceptions.

- The Throwable class is divided into two groups:

1. **Error class.** The errors controlled by this class are those produced by abnormal situations in the execution of the program, produced by the system and very difficult to recover. Therefore, they are not controlled by the programmer.
2. **Exception class.** These are the types of errors that occur during program execution, due to abnormal situations that may arise. These errors can and must be controlled, thus providing a solution to the situation.

   From this class derives a large number of classes that can be grouped into two blocks:

   a. The **RuntimeException class.**

   b. The rest of the classes.

2. Hierarchy of exceptions.

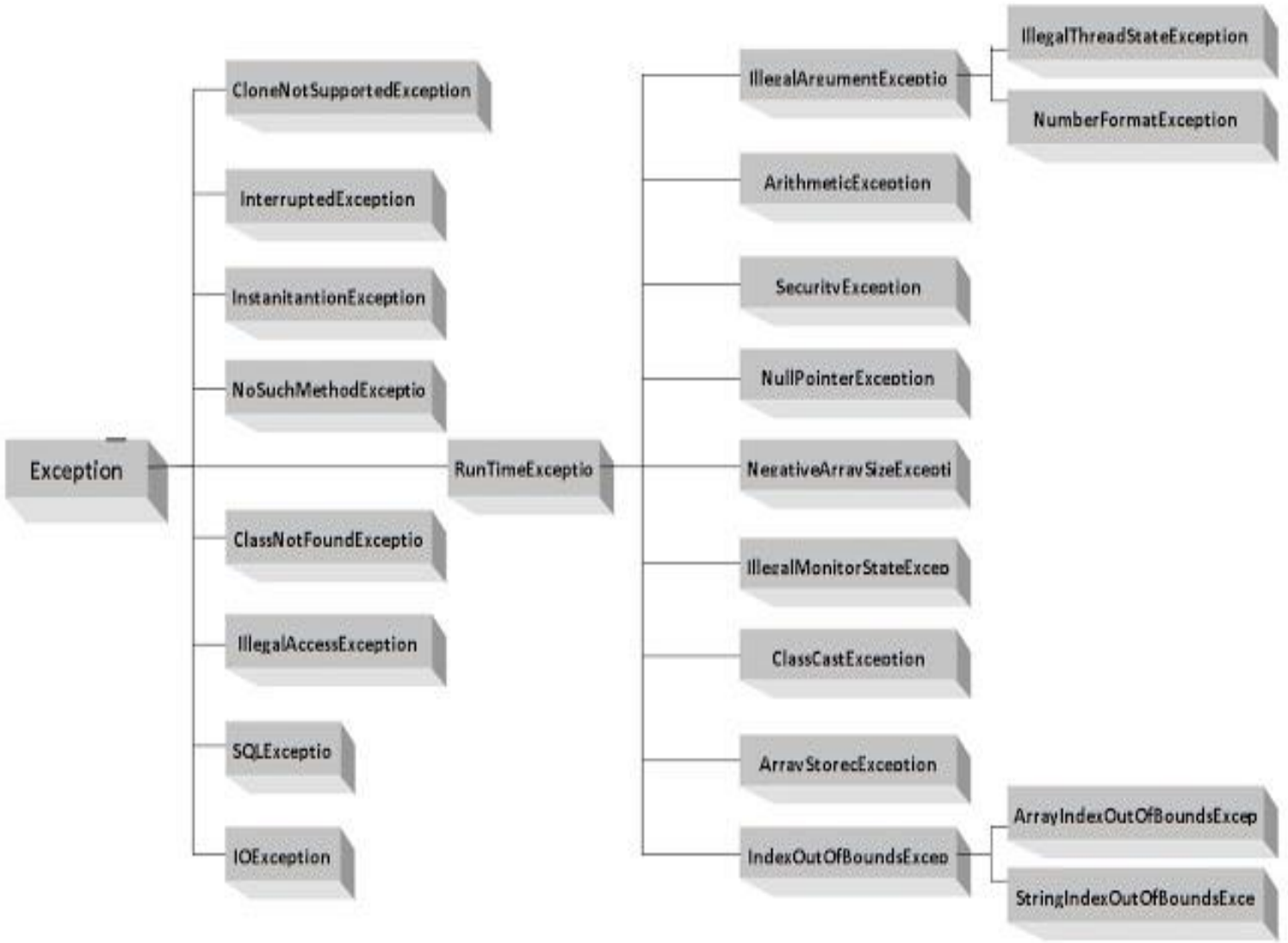
a. The RuntimeException class. From this class derive classes such as:

- ✓ NullPointerException. A null reference has been used to access a field.
- ✓ ArithmeticException. Calculation errors such as division by zero.
- ✓ IllegalArgumentException. From this class derive others, such as InputMismatchException and NumberFormatException: you insert letters when you are asked for numbers.
- ✓ IndexOutOfBoundsException. Access to a vector out of bounds.

IT IS NOT MANDATORY TO CONTROL THIS TYPE OF ERRORS. IT IS UP TO THE PROGRAMMER WHETHER HE WANTS TO DO IT OR NOT.

b. The rest of the classes. IS MANDATORY TO MANAGE THEM.

2. Hierar


Exception
CloneNotSupportedException
InterruptedException
InstanitantionException
NoSuchMethodExceptio
ClassNotFoundExceptio
IllegalAccessException
SQLExceptio
IOException
RunTimeExceptio
IllegalAreumentExceptio
ArithmeticException
SecuritvException
NullPointerException
NegativeArravSizeExcepti
IllegalMonitorStateExcep
ClassCastException
ArravStorecException
IndexOutOfBoundsExcep
IllegalThreadStateException
NumberFormatException
ArrayIndexOutOfBoundsExcep
StringIndexOutOfBoundsExce

3.Exception Handling

3.1 Capture Exceptions

3.2. Propagate exceptions.

3.3. Throwing exceptions.

Abraham Pérez Barrera

3. Exception handling.

3.1. Exception capture.

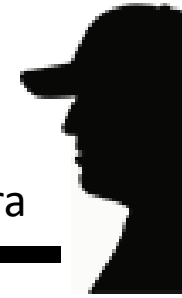
- When we foresee that a code fragment may cause an **exception**, we must capture it in a block of the form

```
try {
  /*instructions that are executed unless there is an error//
}catch (ClassException objectThatCatchesException) {
  /*instructions to be executed if there is an error//
}
```

3. Exception handling.

- If the error occurs, the instructions in the **try** block are no longer executed and the instructions in the **catch** block are executed instead.

- If no error occurs, the instructions of the **try** block will be executed until the end of the block is reached.



3. Exception handling.

- We can include more than one **catch** block, each catching a different type of error.
- The order in which the **catch** blocks are placed is very important.
- When an error occurs in any of the instructions in the **try,** it will exit the **try** and will go through the **catches** as it finds them and will enter the first **catch** it finds, where the type of error it catches matches the one that has occurred. When it goes through a **catch**, it will not go through the following catches.

3. Exception handling.

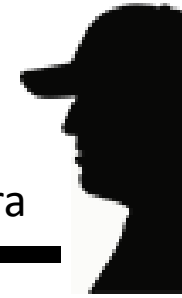
- Therefore, the order of the **catches** is very important, taking into account the hierarchy of exceptions, because if in the first catch we put:

```
catch (Exception e){
}
```

The Exception class includes all types of controllable errors; thus, when an error occurs, it will pass through this **catch** and will no longer pass through any of the other **catches** below it.

3. Exception handling.

- Strategy: first we put the **catch** blocks that catch specific errors and then those that catch a set of errors.
- To follow this strategy, it is advisable to consult the hierarchy of **exception classes**.



3. Exception handling.

- In addition to **try** and **catch**, we can use a third type of block: **finally.**
- **The finally** block includes all the instructions that we want to be executed regardless of whether a problem has occurred or not. It must always be at the end of all **catches**.

3. Exception handling.

Exercise 1: Program that asks the user for the dividend and the divisor and performs the division. When you run the exercise, you will see that if you insert letters or a zero (as divisor), the execution of the program is aborted and an error is thrown.

In order to control this situation and ensure that the program can continue with the execution of the program, these errors must be captured.

Exercise 2: Program that controls the possible errors or exceptions of the previous exercise.

3. Exception handling.

Exercise 3: in the previous exercise write this block before the other two catch blocks:

```
catch (Exception e)
    {
       System.out.println("Some error has occurred so no."
            + "division is to be performed");
    }
```

- **What happens?** It gives a coding error on the second and third catch, telling you that the error has already been caught.
- Therefore, it is very important in which order the catches are placed. FIRST, you put the **catches** that catch specific errors and THEN the **catches** that group a set of errors. To do this, you must take into account the Java API and see what is the inheritance between exception classes.

3. Exception handling.

Exercise 4: change the previous exercise, including the finally block. Display a message inside this block.

Run the program and notice that the message in the finally block is always displayed (regardless of whether an error has occurred or not).

Exercise 5: starting from the previous exercise, create a new project in such a way that if the user inserts letters instead of numbers, the program displays a message and gives the option to insert the dividend and the divisor again.

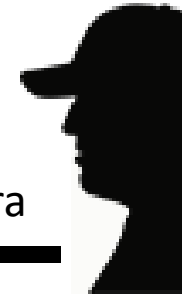
If the user inserts 3 times letters instead of numbers, the program will close and a message like this will be displayed:

"We closed the program. There are too many wrong entries."

If the user inserts a zero as divisor, the division will be terminated, displaying a message like this before closing:

"You have inserted a zero as divisor so the division is not going to be performed."

IMPORTANT: dividend and divisor are String.



3. Exception handling.

3.2. Propagate exceptions.

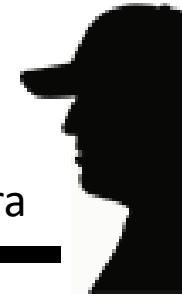
- When an error occurs within a method, it can be caught within the method and handled from there.
- If the error is not caught, Java automatically propagates the error to the calling method for that method to catch it. So on and so forth, all the way to the main.
- The main, in turn, would propagate it to the operating system. This operating system would handle it as it is established (in the cases in which a program closes throwing an error).
- Only errors that are derived from the RuntimeException class are propagated. Errors of the Exception class (not belonging to the RuntimeException class) are not propagated and force you to catch or throw them.

3. Exception handling.

Exercise 6: In this exercise, we are going to see how the function askEnter() does not control the possible error that could occur if the user inserts letters. Therefore, in the event of such an error, the method askEnter() will automatically propagate the error to the main so that this method can control it.

(Code on the next slide)

Create a new project to test its operation.



3. Exce

```java
public static void main(String[] args) {

    int num;

    System.out.print("\n\nInserte un número: ");

    try {
        num=pedirEntero();
        System.out.println("El número insertado es " + num);
    }

    catch (NumberFormatException e)
    {
        System.out.println("Se han insertado letras en vez de un número. MENSAJE QUE VISUALIZA EL MAIN.");
    }
}

static String pedirCadena()
{
    String cadena="";
    Scanner entrada=new Scanner(System.in);

    try
    {
        cadena =entrada.nextLine();

    }
    catch(Exception e)
    {
        System.out.println("Salimos del programa porque ha ocurrido un error grave.");
        System.exit(0);
    }
    return cadena;
}

static int pedirEntero()
{
    String cadena;
    int num;

    cadena = pedirCadena();
    num = Integer.parseInt(cadena);

    return num;
}
```


3. Exception handling.

Exercise 7: Do the previous exercise again but catching the error inside the method. You will see that the error is no longer propagated, therefore, it no longer goes through the catch of the main. To do this, we change the **method** “askInteger( )" by:

```
static int pedirEntero()
{
    String cadena;
    int num=0;

    cadena = pedirCadena();
    try
    {
        num = Integer.parseInt(cadena);
    }
    catch(NumberFormatException e)
    {
        System.out.println("Se han insertado letras en vez de números.");
        System.exit(0);
    }

    return num;
}
```

"**System.exit**(0);" is used to force the program to terminate.

3. Exception handling.

Exercise 8: Write a program that creates and initializes an array and handles the exception that we access an index that is out of range.

Exercise 9: Write a program that asks for numbers, in order to display the result of adding all the inserted numbers. Numbers will be requested **until a letter is inserted** or until five numbers are inserted. **Using JoptionPane**

3. Exception handling.

3.3. Throwing exceptions.

- Errors can be thrown in two different situations:

1st) Inside a method where one of the errors that Java forces you to control (for example, IOException) can occur.

Within this method, there are two options:

a) Capture the error with the try-catch (take from the platform exercise ExampleWithoutLaunch).

b) Throw the error. A method throws the error by putting this header (to the method):

typeReturned MethodName(parameters declaration) **throws ClassNameException**

(taken from the EjemploLanzando exercise platform).

3. Exception handling.

- A method can throw more than one exception. To do so, the header of such a method would be:

  typeReturned methodName(parameter declaration) **throws NameClassException1, NameClassException2,....**

```
static int calculate(String str1, String str2)throws ExceptionInterval,
NumberFormatException, ArithmeticException{
        int num=Integer.parseInt(str1);
        int den=Integer.parseInt(str2);
        if((num>100)||(den<-5)){
            throw new ExceptionInterval("Numbers outside the interval");
        }
        return (num/den);
    }
```


3. Exception handling.

IMPORTANT TO REMEMBER: as we have already seen in the previous point, the errors derived from the RuntimeException class, such as NumberFormatException, do not need to be thrown, since in the case that such error occurs, it will be propagated automatically, if it is not caught inside the method.

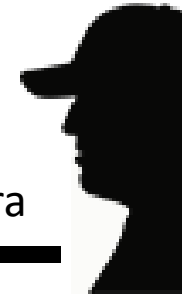
3. Exception handling.

2nd) Within a method, we may want to control an error, but we may also want to control it outside the method. To do this, it is necessary to catch the error inside the method itself (with the try-catch) and, inside the catch, throw the error so that the method that called the function can receive it.

- An error is launched like this:

throw ObjectNameTypeException;

- The throw instruction causes the execution flow of that method to be abandoned, giving control to the calling method and passing through the catch (of the calling method) that catches the error.



3. Exception handling.

- If the method throws an error NOT belonging to the RuntimeException class, in the header of the method you must put:

  typeReturned methodName(parameter declaration) **throws ClassNameException**

- If the error DOES belong to the RuntimeException class, it does not force me to put that header.

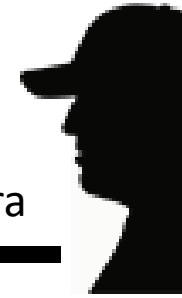
3. Exception handling.

Exercise 10: Program that asks for a series of numbers until letters are inserted. The sum of all the inserted numbers must be displayed.

To do this, we want to use a method that asks for an integer and we want to control, within that method, the fact that it can insert letters. In this case, the method will display a message like this: "You have inserted letters instead of numbers".

Therefore, this method must control the error, but it must also throw it so that the main can receive the error, thus knowing that the exception has occurred and be able to close the program.

Exercise 11: Try to remove the catch that controls the error "NumberFormatException" in main and insert letters. See what happens.



Own exceptions

Abraham Pérez Barrera

Proprietary Exceptions

- The java language provides classes that handle almost any type of error at run time, by means of exceptions.

- But it may be the case that we need to make it known that an error is occurring that is not covered by Java.

Proprietary Exceptions

- In order to throw an exception created by us, we must first create this exception.
- To create an exception we have to create a class that **inherits** from the *Exception* class.

EXAMPLE:

We are going to create an exception to throw in case a number is out of an interval:

```
public class ExceptionInterval extends Exception {
    public ExceptionInterval(String msg) {
        super(msg);
    }
}
```

The definition of the class is very simple. A string *msg*, which contains a message, is passed in the only parameter that the constructor of the derived class has, and it is passed to the base class through **super**.

Proprietary Exceptions

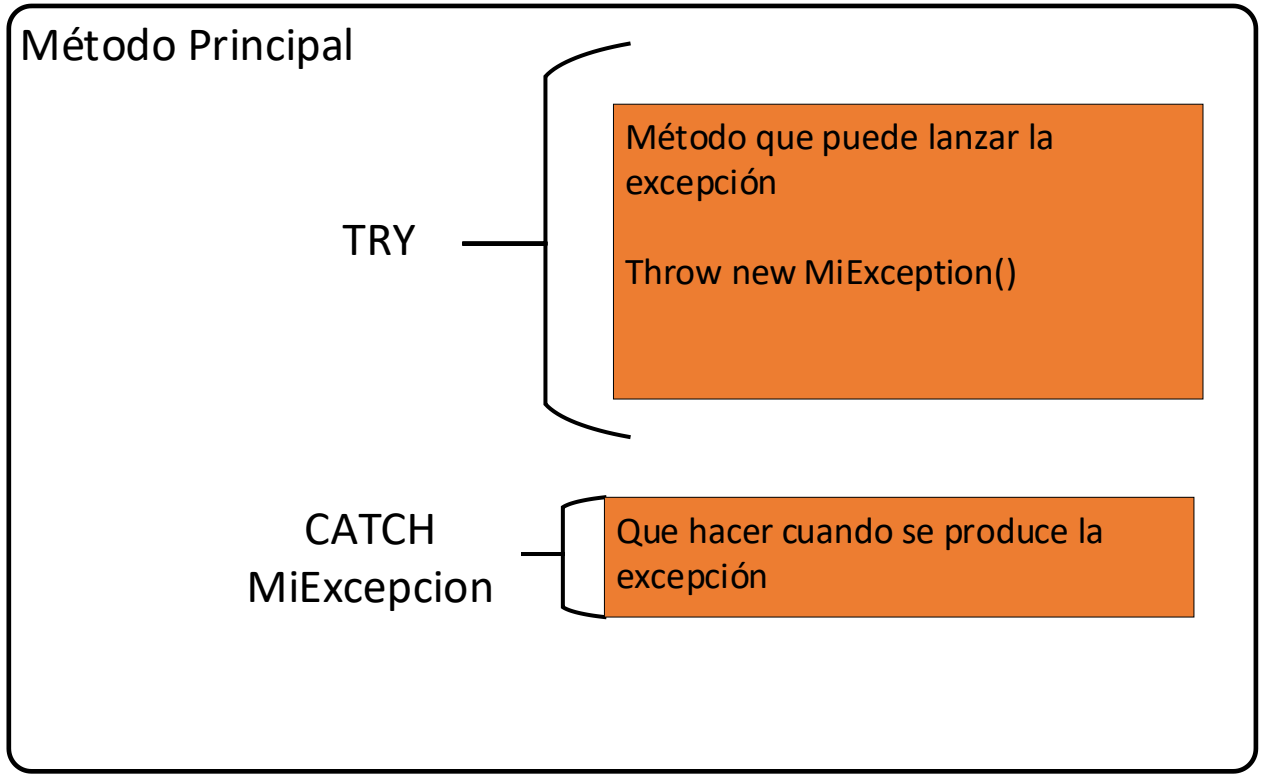
The method that can throw an exception has the usual declaration of any method, but the reserved word **throws** is added to it followed by the exception or exceptions it can throw.

```
static void range(int num, int den)throws ExceptionInterval{
        if((num>100)||(den<-5)){
            throw new ExceptionInterval("Numbers outside the interval");
        }
    }
```

When the condition or conditions that cause the error are met; in this case: the num is greater than 100 or the den is less than 5, an exception is thrown, using the reserved word **throw** and an object of the ExceptionInterval class.

This object is created by calling the constructor of this class and passing it a String containing the message "Numbers out of range".

Own Exceptions - Life Cycle of an Exception

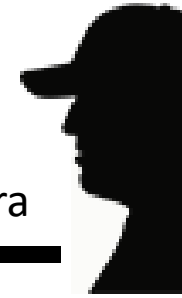


Own Exceptions - Life Cycle of an Exception

The life cycle of an exception can be summarized as follows:

- The call to the method likely to raise an exception is placed in a **try** block...**catch**
- In this method, an object of the *Exception* class or an object derived from it is created by means of **new.**
- The newly created object is **thrown**
- It is captured in the corresponding **catch** block
- This block notifies the user of this eventuality by printing the message associated with the exception, or by performing a specific task.

4. Methods of the Exception class



4. Methods of the Exception class.

- The **Exception class** is the **superclass** of all **exception** types.
- This allows the use of a number of **methods** common to all **exception** classes, such as:
  - **String getMessage**(). Gets the message describing the **exception** or a specific indication of the error occurred.
  - **String toString**(). Writes a string about the status of the **exception**. Usually indicates the **type** of **exception** and the text of **getMessage**().
  - **void printStackTrace().** Writes the **method** and **exception** message (the stack information). This is the same message displayed by the Java virtual machine when the **exception** is not handled.

4. Methods of the Exception class.

- An example of using **toString** and **getMessage** is as follows:

```
public static void main(String[] args) {

    int x = 15;
    int y = 0;
    try {
        System.out.println("El resultado de la división entera de " + x
                + " entre " + y + " es " + x/y);
    } catch (Exception e) {
        System.out.println("Ha intentado dividir por 0;");
        System.out.println(e.getMessage());
        System.out.println(e.toString());
    }
}
```


4. Methods of the Exception class.

- Which produces the following output:

```
run:
Ha intentado dividir por 0;
/ by zero
java.lang.ArithmeticException: / by zero
BUILD SUCCESSFUL (total time: 0 seconds)
```

- In the example we have tried to perform a division by 0, which has caused an **exception**.
- The value returned by **getMessage** is "/by zero>.
- The value returned by **toString** is "java.lang.ArithmeticException: /by zero".

4. Methods of the Exception class.

Other common exceptions

- **ClassCastException**: occurs when we try to **cast** an **object** to a **class** of which it is not a **subclass**. For example, if we try to **cast** integer to string, as the **Integer class** is not a **subclass** of the **String class**, it cannot be done.
- **NegativeArraySizeException**: occurs when we try to create an **array** with negative length. For example, if we ask for the number of **objects** to declare the size of an **array** and a negative number is entered.

4. Methods of the Exception class.

- **InputMismatchException**: occurs when invoking some **methods** of the **Scanner class** in search of a data and this does not exist in the input. For example, if we invoke **nextFloat**() to read a real number and a string has been entered.
- **NullPointerException**: occurs when we try to access any method or attribute of an **object** that is set to **null**. For example, if we try to query the name **attribute** of an **object** of type "Boss" in which data has not yet been inserted.
- **IOException**: occurs when there is a problem reading or writing to an external medium. For example, when writing to a file.