Improving software quality: runtime errors (exceptions) and bugs (debugging)

Why is it important to learn methods to improve software quality (**reliability**)?

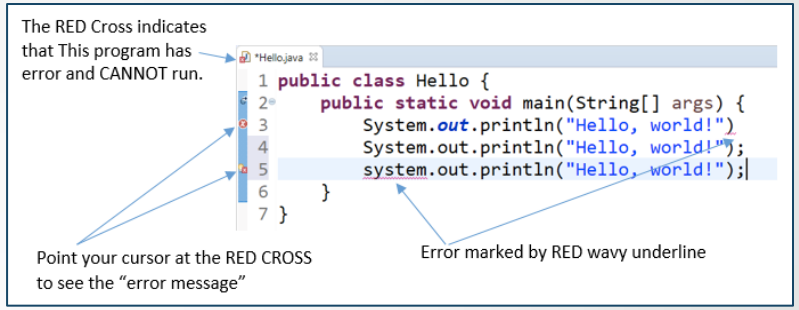
The Ariane rocket disaster of 1996, two reasons:

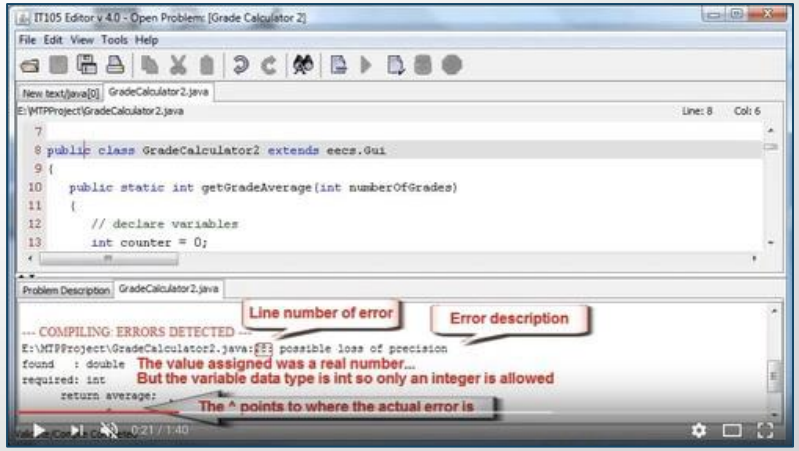
1. **A numeric overflow error**: attempt to fit a 64-bit format data (float) in 16-bit space (integer) and there was no exception handler associated with that conversion.
2. **Code reuse in a different context without testing**: as the facility that failed was not required for Ariane 5, there was no requirement associated with it and therefore there were no tests of that part of the software and hence no possibility of discovering the problem!

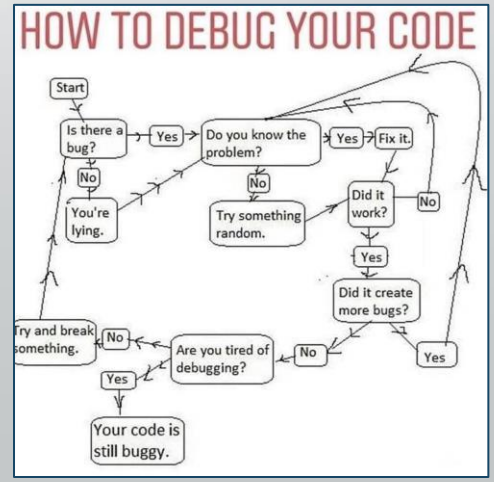
Very good analysis by Ian Somerville here: https://www.youtube.com/watch?v=W3YJeoYgozw

Java errors

1. **Syntax errors** (compile-time errors, managed by modern IDEs)
   - Violation of Java's grammatical rules
   - The code will not even compile
2. **Runtime errors** (managed through exceptions)
   - Happen while program is running
   - Will cause the program to crash
3. **Bugs** (logical errors that can be found and avoided thanks to modern debugging and testing tools, respectively)
   - Program just doesn't do what you'd expect
   - Program has not the expected quality (security and performance bugs, for instance)

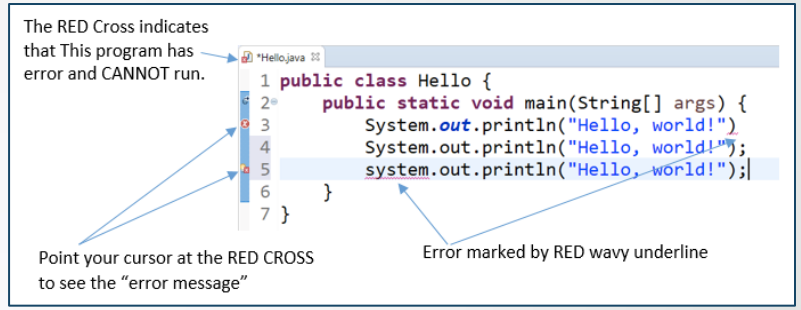


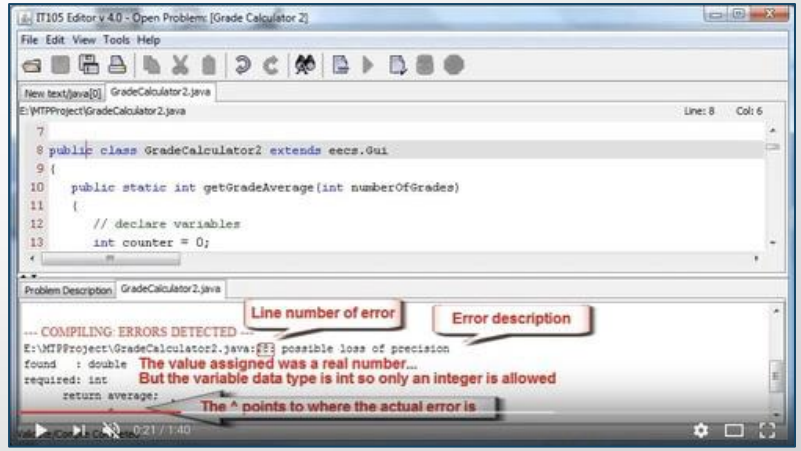


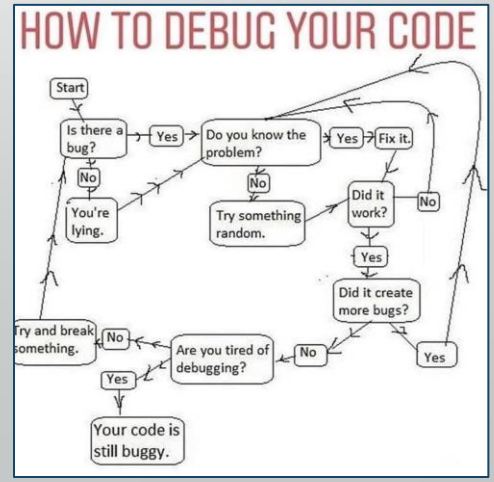


Java errors

1. **Syntax errors** (compile-time errors, managed by modern IDEs)
   - Violation of Java's grammatical rules
   - The code will not even compile
2. **Runtime errors** (managed through exceptions)
   - Happen while program is running
   - Will cause program to crash
3. **Bugs** (logical errors that can be found and avoided thanks to modern debugging and testing tools, respectively)
   - Program just doesn't do what you'd expect
   - Program has not the expected quality (security and performance bugs, for instance)







Exceptions

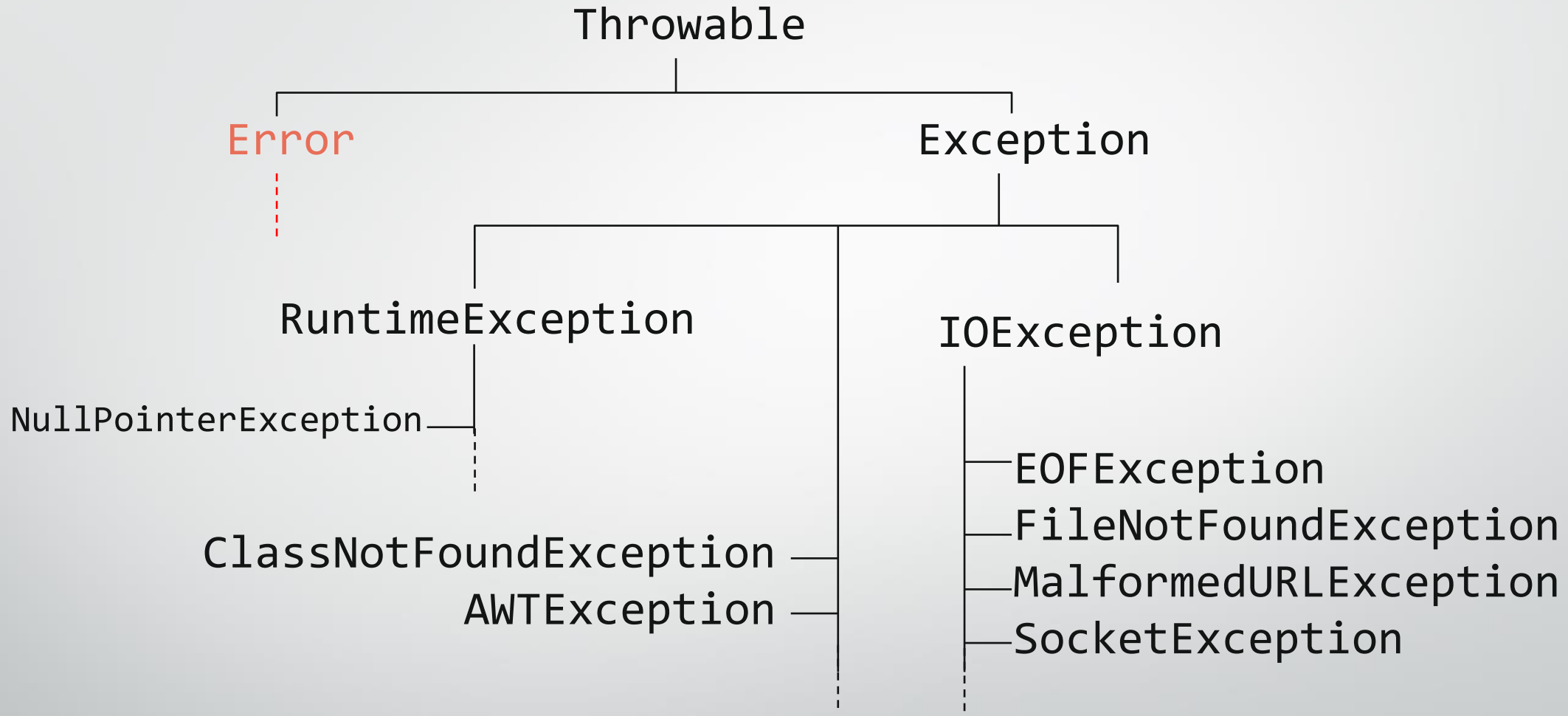
- Permettent de signaler toute **erreur survenue pendant l'exécution d'une méthode** :
  - débordement d'indice d'un tableau,
  - erreur d'accès à un fichier,
  - ...
- Ecriture du traitement d'erreur plus lisible : séparation du **fonctionnement normal (mode nominal)** du **mode erreur**.

Interception des erreurs : classes d'exceptions

L'interception des erreurs d'exécution empêche tout arrêt brutal d'une application

- A chaque fois qu'une erreur est détectée à l'exécution : une instance de la classe **Throwable** est créée.
- Deux sous-classes héritent de la classe **Throwable** : **Error** et **Exception**.
  - **Error** : cas d'erreur au niveau machine virtuelle Java (erreur fatale)
  - **Exception :** erreur d'exécution du programme, type d'erreur sur laquelle le programmeur peut intervenir.
- Ces classes comportent toute sorte de méthodes et de champs, qui permettent une gestion d'erreur bien plus riche qu'un simple code erreur.
- Si besoin est, le programmeur peut créer sa propre hiérarchie de classes d'exceptions.

Hiérarchie des classes d'exceptions



Les erreurs (dérivant de java.lang.Error) représente des erreurs d'exécution dans la JVM. Autant vous dire que si vous interceptez ces types d'objets dans un bloc try / catch, vous ne pourrez pas y faire grand-chose. Les deux classes d'erreurs les plus connues sont OutOfMemoryError (plus de possibilité de faire des new) et StackOverflowError (vous avez saturé la pile d'exécution).

Exceptions levées par la JVM

TestExceptions.java

```
public class TestExceptions {
    public static void main(String[] args) {
        int[] intArray = new int[3];
        intArray[0] = 1;
        intArray[1] = 2;
        intArray[2] = 3;
        for(int i=0; i<=intArray.length; i++){
            System.out.println(intArray[i]);
        }
        System.out.println("End of the main");
    }
}
```

Les exceptions générées par le système sont automatiquement levées par le système d'exécution Java.

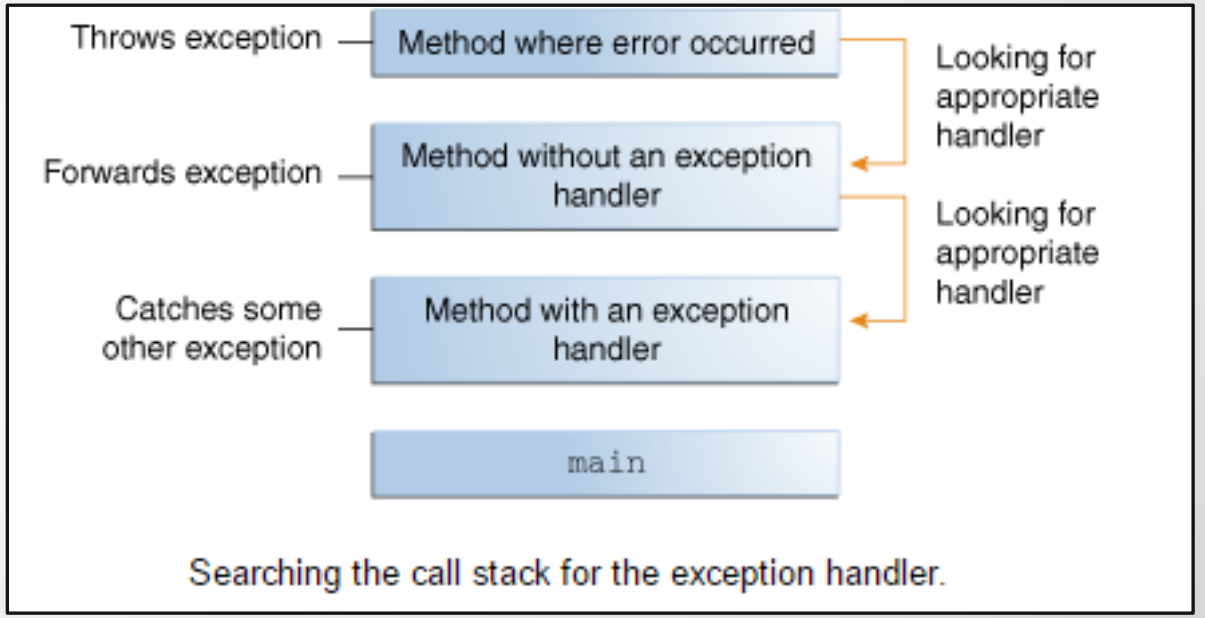
Comment ça marche?

```
run:
1
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at holamundo.TestExcepciones.main(TestExcepciones.java:19)
C:\Users\-Daniel\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

TestException.java

```java
public class TestException {
    public static void method3() {
        System.out.println( "BEGIN method3" );
        int value = 1 / (int) (Math.random() * 3);
        System.out.println( "Value == " + value );
        System.out.println( "END method3" );
    }
    public static void method2() {
        System.out.println( "BEGIN method2" );
        method3();
        System.out.println( "END method2" );
    }
    public static void method1() {
        System.out.println( "BEGIN method1" );
        method2();
        System.out.println( "END method1" );
    }
    public static void main(String[] args) {
        System.out.println( "BEGIN main" );
        method1();
        System.out.println( "END main" );
    }
}
```

How does it works? After a method throws an exception, the runtime system attempts to find something to handle it.



Searching the call stack for the exception handler.

```
BEGIN main
BEGIN method1
BEGIN method2
BEGIN method3
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at exceptions.TestException.method3(TestException.java:8)
    at exceptions.TestException.method2(TestException.java:15)
    at exceptions.TestException.method1(TestException.java:21)
    at exceptions.TestException.main(TestException.java:27)
```

Comment éviter que le programme s'arrête prématurément?

Il faut donc capturer et traiter les exceptions!

Pour cela Java propose: `try {...} catch {…}` et `finally {…}`

- **Le bloc `try`** encadre les instructions où une ou plusieurs exceptions sont susceptibles d'être déclenchées.

  Ce bloc représente le traitement normal du programme. Elles sont censées se dérouler s'il n'y a pas d'erreur.

- **Les blocs `catch`** suivent un bloc `try`.
  - Chaque catch spécifie quelle classe d'exception il est capable d'intercepter quand une exception est déclenchée dans le bloc du `try`.
  - Le catch reçoit en paramètre l'exception déclenchée.
  - Si vous fournissez plusieurs blocs `catch`, ils devront alors être ordonnés du bloc le plus spécifique ou bloc le plus général (sinon, erreur de compilation).

```
try {
  operation_risquée1;
  opération_risquée2;
} catch (ExceptionInteressante e) {
  traitements
} catch (ExceptionParticulière e) {
  traitements
} catch (Exception e) {
  traitements
} finally {
  traitement_pour_terminer_proprement;
}
```

- **Le bloc `finally`** s'exécute toujours lorsque le bloc `try` s'exécute. Le but est de pouvoir traiter les exceptions non capturées, même si une exception inattendue survient.

Exemple

Quelle différence avec les exceptions levées par la JVM (voir 3 slides avant)?

TestException.java

```
public class TestException {
   public static void main(String[] args) {
      try {
         int[] intArray = new int[3];
         intArray[0] = 1;
         intArray[1] = 2;
         intArray[2] = 3;
         for(int i=0; i<=intArray.length; i++){
            System.out.println(intArray[i]);
         }
      }catch (Exception e){
         System.out.println("Attention! il faut traiter l'exception : "+e.toString());
      }
      finally {
         System.out.println("Et le finally s'execute toujours: avec ou sans exception");
      }
   }
}
```

```
1
2
3
Attention! il faut traiter l'exception : java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
Et le finally s'execute toujours: avec ou sans exception
```

The try-with-resources block

```
public static void main(String[] args) {
        System.out.print("Please provide an integer \n> ");
        try (Scanner sc = new Scanner(System.in)) {
                int x = sc.nextInt();
                System.out.println(x);
    } catch(InputMismatchException ime) {
                System.out.println("You did not input an integer");
        ime.printStackTrace(System.out);
    } catch(NoSuchElementException nse) {
        System.out.println("There is no input to print");
        nse.printStackTrace(System.out);
    } catch(IllegalStateException ise) {
        System.out.println("The Scanner object to get user input is not in an open state");
        ise.printStackTrace(System.out);
    }
}
```

The try block can be followed by a series of **resource declarations** (separated by commas if there is more than one). A **resource** is an object that must be closed after the try statement is finished, and thus, the effect of this block is automatically calling **close()** on our resource, freeing us from needing to call it explicitly. Otherwise, it behaves just like a normal **try/catch** clock. This helps us also avoid using **finally** or making calls to the resource outside of the block.

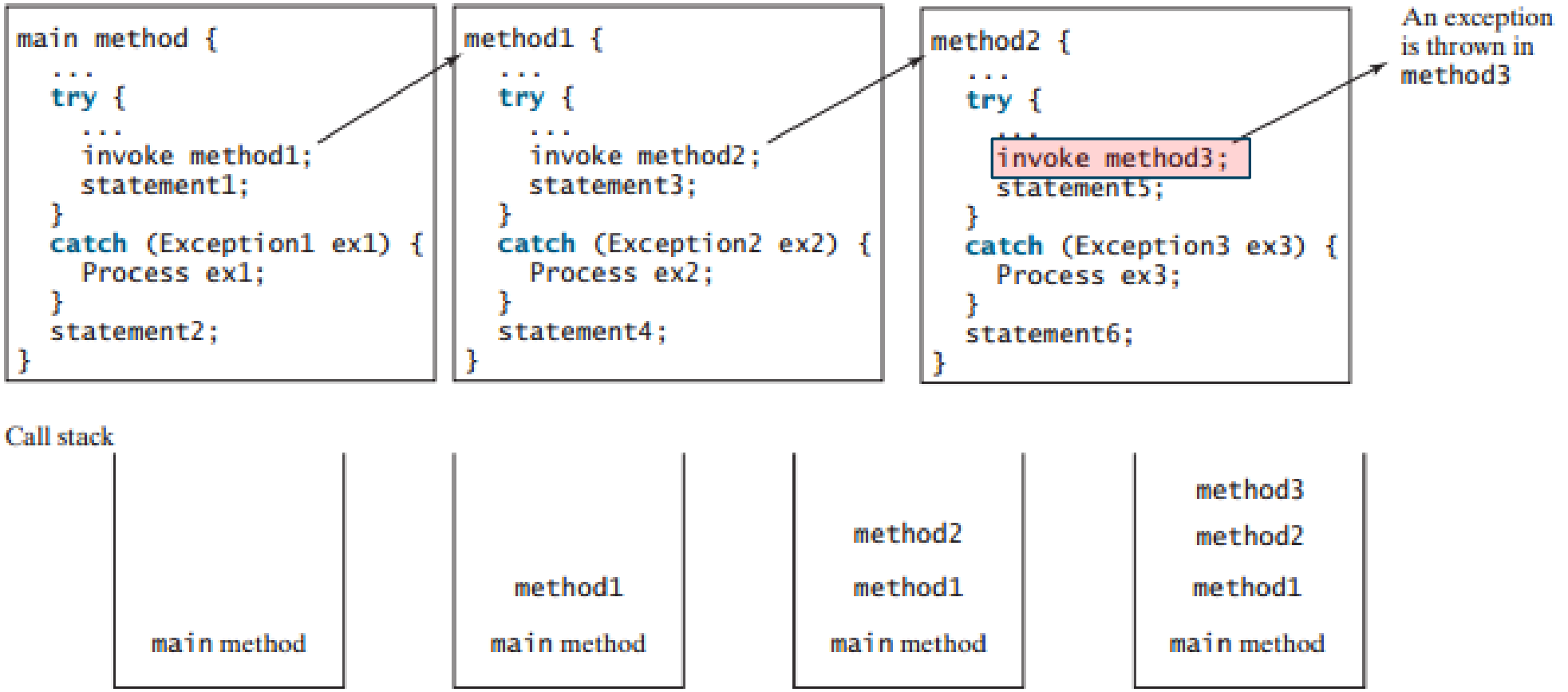


FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

Exercice:

What statements are executed if the method3 throws an exception of the type:
(i) Exception3? (ii) Exception2? and (iii) Exception1?

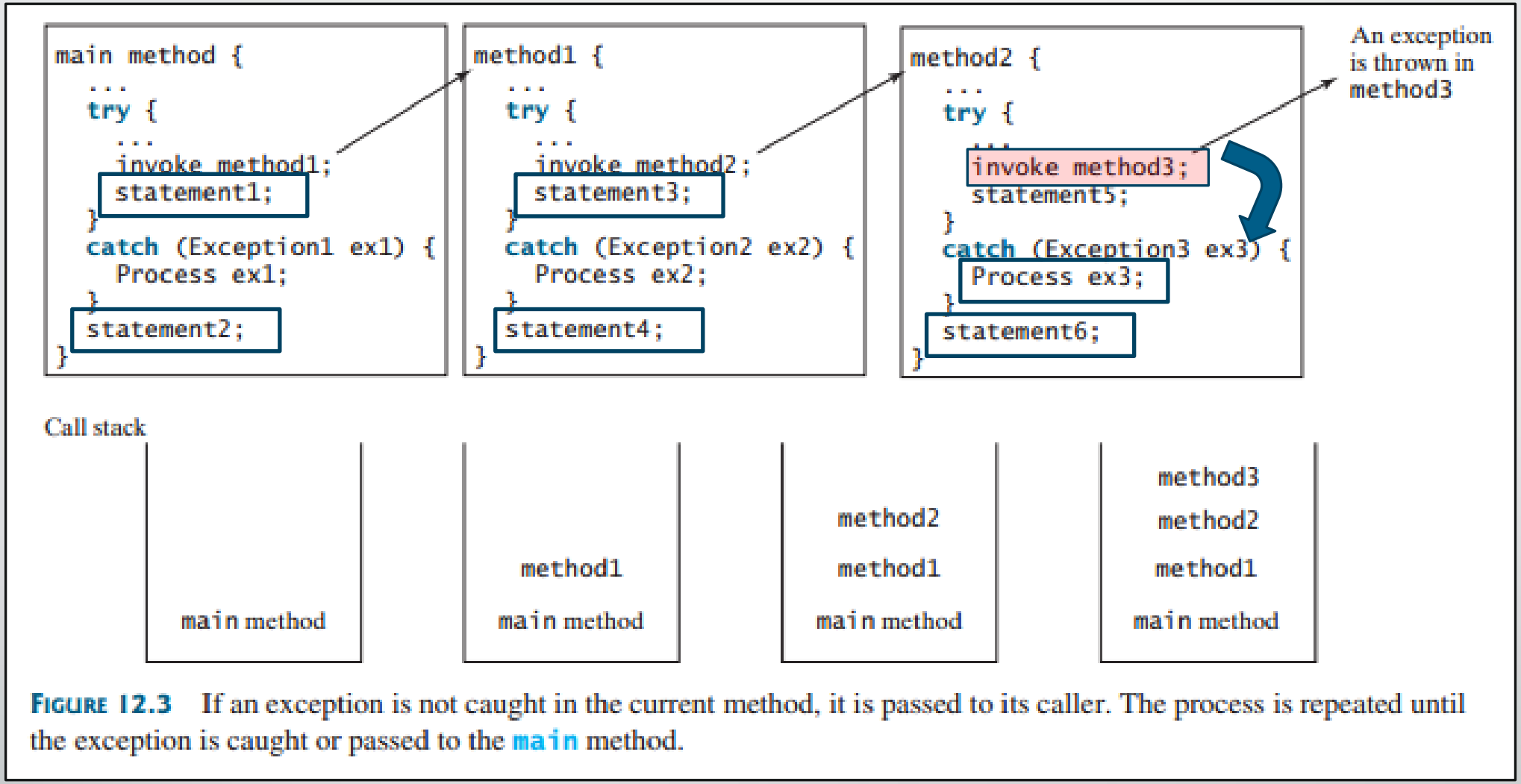


FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

Exercice:

If method3 throws an Exception3 exception

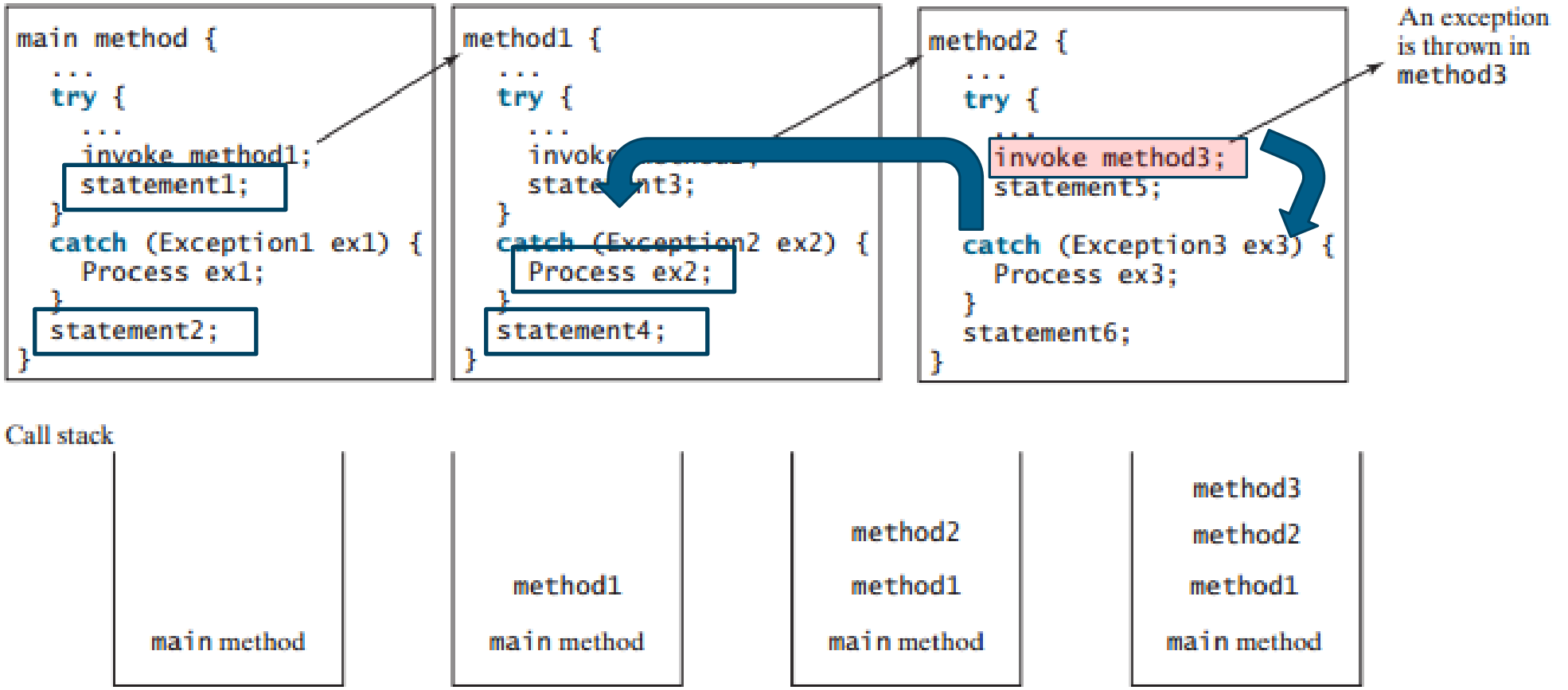


FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

Exercice:

If method3 throws an Exception2 exception

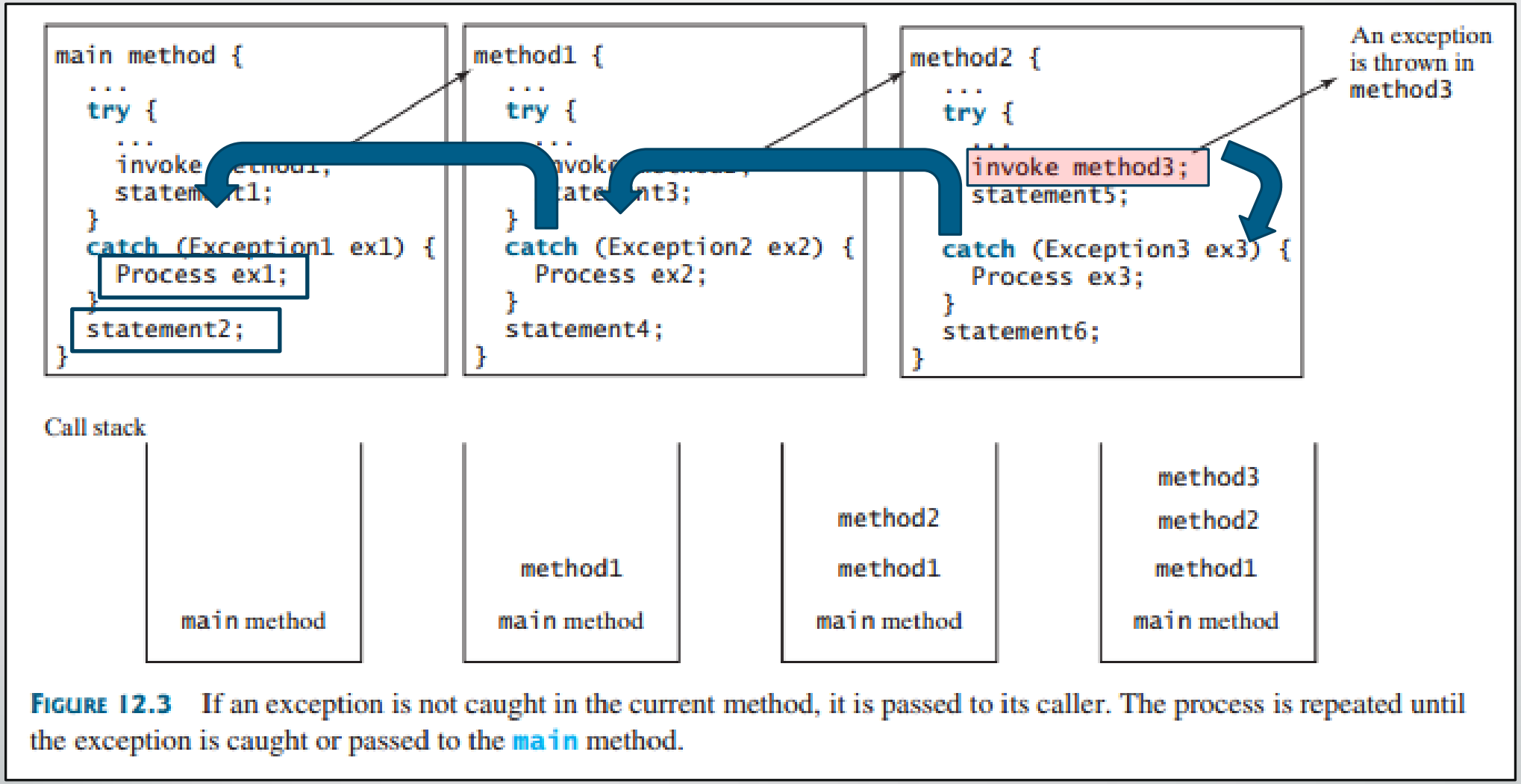


FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

Exercice:

If method3 throws an Exception1 exception

Exercice:

Quelle est la sortie de ce programme?

```
public class A {
  public static void main(String[] args){
    try{
      m1();
      System.out.println("statement0");
    }
    catch(Exception e){
      System.out.println("exception0");
    }
    System.out.println("final0");
  }

  static void m1(){
    try{
      m2();
      System.out.println("statement1");
    }
    catch(ArithmeticException e){
      System.out.println("exception1");
    }
    System.out.println("final1");
  }

  static void m2(){
    try{
      m3();
      System.out.println("statement2");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("exception2");
    }
    System.out.println("final2");
  }

  static void m3(){
    try{
      int x = 0; int y = 0; y=x/y;
      System.out.println("statement3");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("exception3");
    }
    System.out.println("final3");
  }
}
```

```
public class A {
  public static void main(String[] args){
    try{
      m1();
      System.out.println("statement0");
    }
    catch(Exception e){
      System.out.println("exception0");
    }
    System.out.println("final0");
  }

  static void m1(){
    try{
      m2();
      System.out.println("statement1");
    }
    catch(ArithmeticException e){
      System.out.println("exception1");
    }
    System.out.println("final1");
  }
```

```
  static void m2(){
    try{
      m3();
      System.out.println("statement2");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("exception2");
    }
    System.out.println("final2");
  }

  static void m3(){
    try{
      int x = 0; int y = 0; y=x/y;
      System.out.println("statement3");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("exception3");
    }
    System.out.println("final3");
  }
}
```

Exercice:

Quelle est la sortie de ce programme?

```
public class A {
  public static void main(String[] args){
    try{
      m1();
      System.out.println("statement0");
    }
    catch(Exception e){
      System.out.println("exception0");
    }
    System.out.println("final0");
  }

  static void m1(){
    try{
      m2();
      System.out.println("statement1");
    }
    catch(ArithmeticException e){
      System.out.println("exception1");
    }
    System.out.println("final1");
  }
```

```
  static void m2(){
    try{
      m3();
      System.out.println("statement2");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("exception2");
    }
    System.out.println("final2");
  }

  static void m3(){
    try{
      int x = 0; int y = 0; y=x/y;
      System.out.println("statement3");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("exception3");
    }
    System.out.println("final3");
  }
}
```

Exercice: Quelle est la sortie de ce programme?

```
exception1
final1
statement0
final0
```

Vous avez donc remarqué que en cas de déclenchement d'exception:

1. votre code ne s'exécute pas (toujours) entièrement, et
2. l'exécution du code est donc redirigée vers le bloc catch le plus adapté et **une fois le bloc catch terminé, le bloc try n'est pas rejoué!**

Si vous souhaitez corriger cela, il vous appartient d'utiliser une boucle pour retenter l'exécution du bloc try. Voici un exemple de code qui oblige l'utilisateur à effectuer une saisie de valeur numérique entier.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Sample {
   public static void main( String [] args ) {
      int value = 0;
      boolean flag = true;
      do {
         try {
            BufferedReader reader = new BufferedReader(
                  new InputStreamReader( System.in ));
            System.out.print( "Veuillez saisir un entier : " );
            String strValue =  reader.readLine();
            value = Integer.parseInt( strValue );
            break;  // Pour sortir de la boucle
         } catch( NumberFormatException exception ) {
            System.out.println( "Ceci n'est pas un entier, essaye encore" );
         } catch( Exception exception ) {
            System.out.println( "Ceci posse problème, essaye encore" );
         }
      } while(flag);
      System.out.println( "Value == " + value );
   }
}
```

Est-ce qu'on peut utiliser un try-with-resources ici?

Exercise

Consider this simple code example:

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        int x = sc.nextInt();
        System.out.println(x);
        sc.close();
}
```

Applying what you've learnt before:

1. Using Java's documentation, **determine which of the preceding operations are risky and can throw exceptions**
2. Once you've done so, **write a try/catch block that adequately deals with each possible case in this program**

Exercise: an in-depth solution

Let's take this line by line:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Please provide an integer \n> ");
    int x = sc.nextInt();
    System.out.println(x);
    sc.close();
}
```

The documentation for the Scanner class is here:
https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Scanner.html

We see that its constructor is defined for several different types, so we must determine what type ***System.in*** is to know what constructor is used exactly. We can see that it is a field of type ***InputStream***.

According to the documentation, **the constructor called throws no errors or exceptions**. We know, then, that this operation will not need to be surrounded in the try/catch block.

Scanner

```
public Scanner(InputStream source)
```

Constructs a new Scanner that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the underlying platform's default charset.

Parameters:

source - An input stream to be scanned

Exercise: an in-depth solution

Let's take this line by line:

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        int x = sc.nextInt();
        System.out.println(x);
        sc.close();
}
```

From <the System class docs> we observe that ***out*** is a static field of type **PrintStream** representing the system's **standard output**. We must then examine <the PrintStream class docs> on the **print(String)** method, whence we can infer that this operation will also not throw any exceptions, and thus we can continue to the next line.

out

```
public static final PrintStream out
```

The "standard" output stream. This stream is already open and ready to accept output data.

print

```
public void print(String s)
```

Prints a string. If the argument is `null` then the string `"null"` is printed.

Exercise: an in-depth solution

Let's take this line by line:

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        int x = sc.nextInt();
        System.out.println(x);
        sc.close();
}
```

From the Scanner documentation, we can determine that ***nextInt()*** can throw three possible exceptions: ***InputMismatchException, NoSuchElementException,*** and ***IllegalStateException***.

We know, then, that this operation is risky and must be included in the try/catch block. We must consider each of these exceptions for inclusion in the try/catch block.

Scanner documentation

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Scanner.html#nextInt()

nextInt

`public int nextInt()`

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:
the `int` scanned from the input

Throws:
`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

Exercise: an in-depth solution

Let's take this line by line:

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        int x = sc.nextInt();
        System.out.println(x);
        sc.close();
}
```

The analysis we performed for the ***print()*** instruction up above holds as well for the **println()** instruction (according to the <documentation>).

However, the value of ***x*** depends on the result of the previous operation, which can generate exceptions. Therefore, this operation should probably not be carried out in the event of an exception in the previous instruction. It would be logical, then, to group it with the previous instruction in the try/catch block.

Exercise: an in-depth solution

Let's take this line by line:

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        int x = sc.nextInt();
        System.out.println(x);
        sc.close();
}
```

The <Scanner Class documentation> that we have seen before indicates that this instruction closes our **Scanner** instance so that no further input operations may be done with it.

To prevent **resource leaks**, it is important that we always explicitly close any input resources. To address this, we know that, exception or not, we must call this method on the **Scanner** instance.

Exercise: an in-depth solution

Here is a possible solution:

```
public static void main(String[] args) {
        //Here, we do no formatting or any other operation than text output,
        //so we can consider it safe.
        System.out.print("Please provide an integer \n> ");
    //We use the special try-with-resources statement, which allows us to declare a resource
    //that will be used within the block and will be automatically closed. Thus avoiding
    //the need to remember to call close() on the resource explicitly.
    try (Scanner sc = new Scanner(System.in)) {
            //Since we have no control over user input we need to cover all
            //possible exceptions thrown by the nextInt method.
            int x = sc.nextInt();
                System.out.println(x);
    } catch(InputMismatchException ime) {
    //Assuming that we are testing this software, it is important to understand the details of
    //the error we observe; we also print to the standard output the details of the exception
        System.out.println("You did not input an integer");
        ime.printStackTrace(System.out);
    } catch(NoSuchElementException nse) {
        System.out.println("There is no input to print");
        nse.printStackTrace(System.out);
    } catch(IllegalStateException ise) {
        System.out.println("The Scanner object to get user input is not in an open state");
        ise.printStackTrace(System.out);
    }
}
```

Exercise: an in-depth solution

Here is a possible solution:

```
public static void main(String[] args) {
        //Here, we do no formatting or any other operation than text output.
        //so we can consider it safe.
        System.out.print("Please provide an integer \n> ");
    //We use the special try-with-resources statement, which allows us to
    //that will be used within the block and will be automatically closed. T
    //the need to remember to call close() on the resource explicitly.
    try (Scanner sc = new Scanner(System.in)) {
            //Since we have no control over user input we need to cover all
            //possible exceptions thrown by the nextInt method.
            int x = sc.nextInt();
                System.out.println(x);
    } catch(InputMismatchException ime) {
    //Assuming that we are testing this software, it is important to unders
    //the error we observe; we also print to the standard output the detai
        System.out.println("You did not input an integer");
    } tch(NoSuchElementException nse) {
        System.out.println("There is no input to print");
        nse.printStackTrace(System.out);
    } tch(IllegalStateException ise) {
        System.out.println("The Scanner object to get user input is not in an open state");
        ise.printStackTrace(System.out);
    }
}
```

Some things to ponder:

1. Are these two exception cases actually possible? Under what circumstances?
2. Can we safely do away with them in this case? And in general?

Exercise: an in-depth solution

Here is a possible solution:

Some things to ponder:

1. Are these two exception cases actually possible? Under what circumstances?
2. Can we safely do away with them in this case? And in general?

The **NoSuchElement** exception is defined as: “Thrown by various accessor methods to indicate that the element being requested does not exist.” Which means that this is thrown when the resource we are scanning can no longer provide any more token (such as having reached the end of a file). While very unlikely when simply run on the console, one cannot exclude the user piping input through the standard input to the program and thus it could happen ($ java myprogram.jar < myfile.txt and such input is an empty file, for instance). In the general case it is probably wise to always be aware of the possibility of this exception, especially when working with files.

```
        ime.printStackTrace(System.out);
    } catch(NoSuchElementException nse) {
        System.out.println("There is no input to print");
        nse.printStackTrace(System.out);
    } catch(IllegalStateException ise) {
        System.out.println("The Scanner object to get user input is not in an open state");
        ise.printStackTrace(System.out);
    }
}
```

Exercise: an in-depth solution

Here is a possible solution:

Some things to ponder:

1. Are these two exception cases actually possible? Under what circumstances?
2. Can we safely do away with them in this case? And in general?

The **IllegalStateException** exception is defined as: “Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation.” In this case this would mean that the **Scanner has already been closed before we seek the next integer**. However, **this cannot be the case since we are using a try-with-resources block that will only call close() after we have run the code in the try block** (or an exception has been thrown). One can argue then that **this exception is unnecessary**. If one however passes around the scanner instance in the program, one can’t necessarily guarantee that it hasn’t been closed elsewhere, so it may be prudent to always prepare for such unforseen cases.

```
    } catch(IllegalStateException ise) {
        System.out.println("The Scanner object to get user input is not in an open state");
        ise.printStackTrace(System.out);
    }
}
```

Can we avoid try catch completely?

```
public static void main(String[] args) {
        int[] intArray = new int[3];
        intArray[0] = 1;
        intArray[1] = 2;
        intArray[2] = 3;
        for(int i=0; i < intArray.length; i++) {
                System.out.println(intArray[i]);
        }
}
```

Is this safer?

What happens when the array is empty?

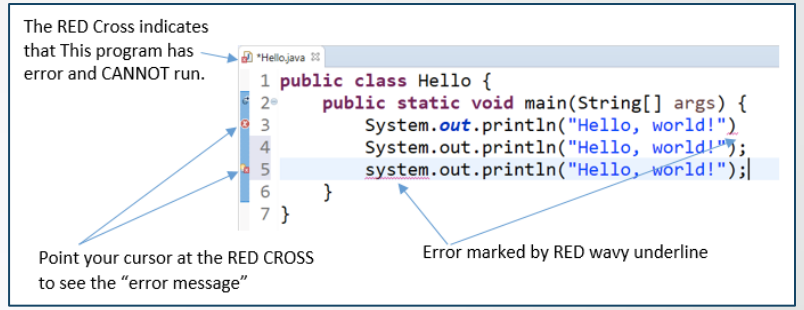
Is this better?

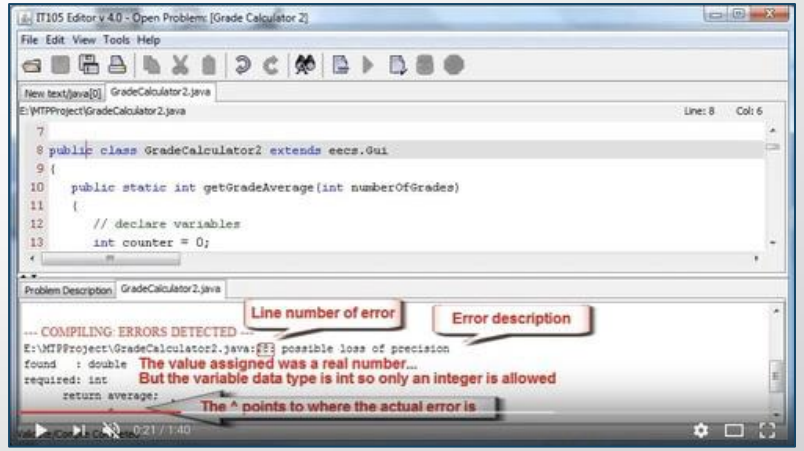
Are there unknown side effects?

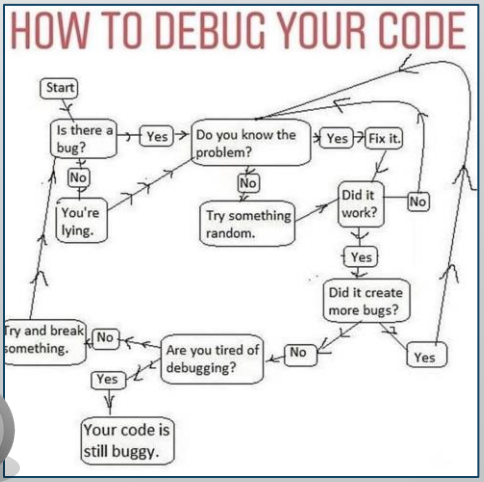
```
public static void main(String[] args) {
        var intArray = new ArrayList<Integer>();
        intArray.add(1);
        intArray.add(2);
        intArray.add(3);
        intArray.stream().forEachOrdered((i) -> System.out.println(i));
}
```

Java errors

1. **Syntax errors** (compile-time errors, managed by modern IDEs)
   - Violation of Java's grammatical rules
   - The code will not even compile
2. **Runtime errors** (managed through exceptions)
   - Happen while program is running
   - Will cause the program to crash
3. **Bugs** (logical errors that can be found and avoided thanks to modern debugging and testing tools, respectively)
   - Program just doesn't do what you'd expect
   - Program has not the expected quality (security and performance bugs, for instance)







QUIZ QUESTION I

What type of error does this code have?

```
for (int i = 1; i >= 100; i++){
    System.out.println(i);
}
```

1. Syntax error
2. Runtime error
3. Bug
4. Nothing is wrong

QUIZ QUESTION II

Consider this code sample…

```
public static void main(String[] args) {
        //Let's test some propositional Logic
        //(Adapted from Velleman D. (1994) - How to Prove it: A Structured Approach)
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        //Assume this operation to obtain user input is safe
        //Assume we can guarantee that he will provide an integer
        int x = sc.nextInt();
        //Consider if the integer is > 2
        //if == true then set the string negationText to the empty string
        //otherwise, set it to ” not "
        String negationText = (x>2) ? " " : " not ";
        //Notice that the purpose of our program is to simply output
        //a property based on the following (possibly incorrect) proposition
        //if x > 2 and x is a Real
        //then x^2 > 4,
        //otherwise x^2 < 4.
        //Given a number we output whether this is the case for any given number, thus, with the
        //insertion of not, we can give either converse statement.
        System.out.println("x is" + negationText + "larger than 2, therefore");
        System.out.println("x^2 (" +(x*x) + ") is" + negationText + "larger than 4");
        sc.close();
}
```

When given a number, it tells us some mathematical property about said number…

QUIZ QUESTION II

1. What mistake did the programmer make? Hint: It's not I/O related
2. How would you fix it?

```
public static void main(String[] args) {
        //Let's test some propositional Logic
        //(Adapted from Velleman D. (1994) - How to Prove it: A Structured Approach)
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        //Assume this operation to obtain user input is safe
        //Assume we can guarantee that he will provide an integer
        int x = sc.nextInt();
        //Consider if the integer is > 2
        //if == true then set the string negationText to the empty string
        //otherwise, set it to "not"
        String negationText = (x>2) ? " " : " not ";
        //Notice that the purpose of our program is to simply output
        //a property based on the following (possibly incorrect) proposition
        //if x > 2 and x is a Real
        //then x^2 > 4,
        //otherwise x^2 < 4.
        //Given a number we output whether this is the case for any given number, thus, with the
        //insertion of not, we can give either converse statement.
        System.out.println("x is" + negationText + "larger than 2, therefore");
        System.out.println("x^2 (" +(x*x) + ") is" + negationText + "larger than 4");
        sc.close();
}
```

QUIZ QUESTION II

```
public static void main(String[] args) {
        //Let's test some propositional Logic
        //(Adapted from Velleman D. (1994) - How to Prove it: A Structured Approach)
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        //Assume this operation to obtain user input is safe
        //Assume we can guarantee that he will provide an integer
        int x = sc.nextInt();
        //Consider if the integer is > 2
        //if == true then set the string negationText to the empty string
        //otherwise, set it to "not"
        String negationText = (x>2) ? " " : " not ";
        //Notice that the purpose of our program is to simply output
        //a property based on the following (possibly incorrect) proposition
        //if x > 2 and x is a Real
        //then x^2 > 4,
        //otherwise x^2 < 4.
        //Given a number we output whether this is the case for any given number, thus, with the
        //insertion of not, we can give either converse statement.
        System.out.println("x is" + negationText + "larger than 2, therefore");
        System.out.println("x^2 (" +(x*x) + ") is" + negationText + "larger than 4");
        sc.close();
}
```

What mistake did the programmer make?

R/ The programmer did not verify that the proposition he is testing is correct for any number. So, he should have verified that the mathematical property does not hold for some numbers (**which ones?**).

QUIZ QUESTION II

```
public static void main(String[] args) {
        //Let's test some propositional Logic
        //(Adapted from Velleman D. (1994) - How to Prove it: A Structured Approach)
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide an integer \n> ");
        //Assume this operation to obtain user input is safe
        //Assume we can guarantee that he will provide an integer
        int x = sc.nextInt();
        //Consider if the integer is > 2
        //if == true then set the string negationText to the empty string
        //otherwise, set it to "not"
        String negationText = (x>2) ? " " : " not ";
        //Notice that the purpose of our program is to simply output
        //a property based on the following (possibly incorrect) proposition
        //if x > 2 and x is a Real
        //then x^2 > 4,
        //otherwise x^2 < 4.
        //Given a number we output whether this is the case for any given number, thus, with the
        //insertion of not, we can give either converse statement.
        System.out.println("x is" + negationText + "larger than 2, therefore");
        System.out.println("x^2 (" +(x*x) + ") is" + negationText + "larger than 4");
        sc.close();
}
```

How would you fix it?

R/ You could: revise the mathematical property so that it is one that holds for all integers **OR** separate it into cases so that it deals correctly with all possible values.

So, where do bugs come from?

Imagine two islands, A and B, whose inhabitants want to build a bridge between them.

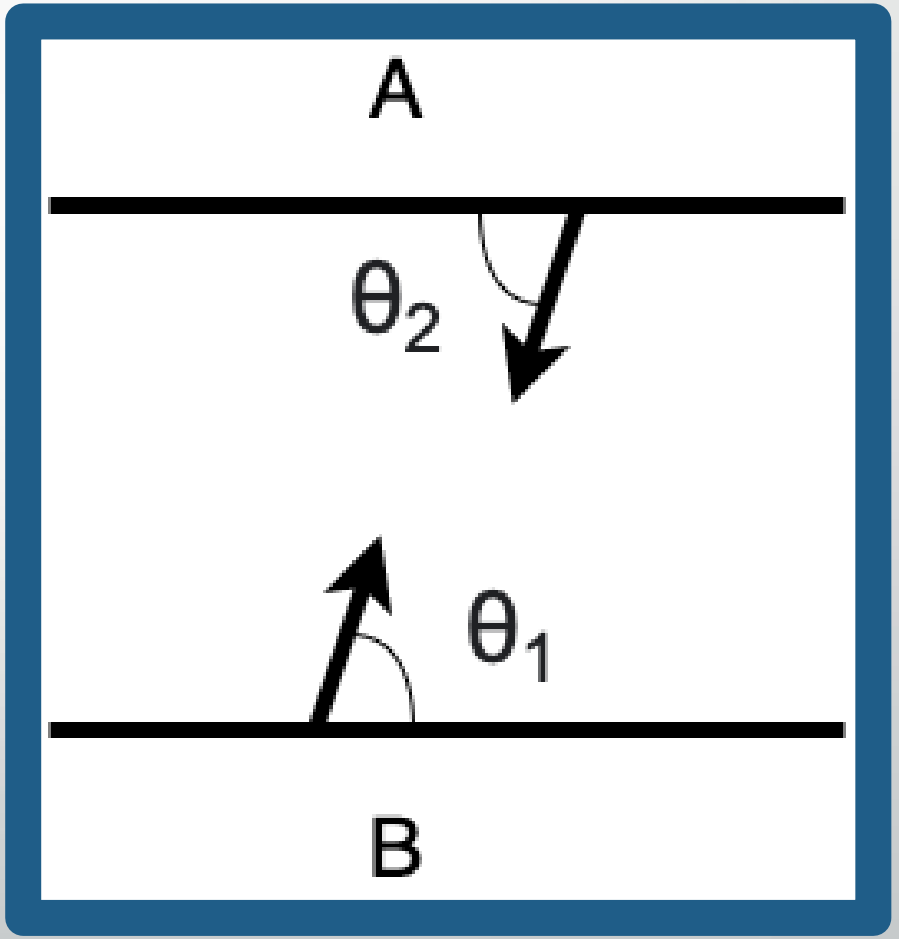
Each side has decided to do half of the bridge.

Given the terrain, they can only begin at the points shown on the diagram, with the bridge at an angle such that both ends will meet in the middle.

Imagine, that for some reason, when the design documents were given to the construction team, each side had a slightly different number for their angle θ…

Conclusion I: Sometimes the information upon which our work is based is incorrect no matter how well we do our job…

Conclusion II: Sometimes it truly is our mistake, and we should always see our work and that of others with a critical eye…



How do we combat bugs, errors, and exceptions?

With unit tests!

In Java we use:

What Junit is?

JUnit is a unit testing framework for Java that allows developers to easily write repeatable tests. All JUnit tests are themselves written as Java Classes, avoiding the need to learn a new language to create the tests.

JUnit 5 – Eclipse

Eclipse has built-in support for JUnit 5 tests.

Using the same method to create a Java class/interface/etc in either folders or at the project level, Eclipse offers the **New JUnit Test Case** wizard to easily create a test class (and add JUnit as a dependency to the project automatically).

To run tests, simply select the file in the project explorer (or folder containing all the tests) and with the context menu select **Run as > JUnit Test.**

JUnit 5 with Gradle or Maven – IntelliJ

IntelliJ has built-in support for JUnit 5 tests through the use of either the Gradle or Maven to build system.

Assuming you are **using Gradle** to build your project you must ensure that the following is in your **build.gradle** file:

```
dependencies {
      //In addition to your other dependencies
      compile 'org.junit.jupiter:junit-jupiter:5.7.0'
}
test {
   useJUnitPlatform()
}
```

The procedure above shows the 'manual' way so that you know what happens behind the scenes and where you set up the testing framework. However, if you just start writing tests, IntelliJ IDEA will automatically detect if the dependency is missing and prompt you to add it.

Assuming you are **using Maven** to build your project you must ensure that the following is in your **dom.xml**:

```
<dependencies>
      <!-- ... -->
      <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.7.0</version>
            <scope>test</scope>
      </dependency>
      <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.7.0</version>
            <scope>test</scope>
      </dependency>
      <!-- ... -->
</dependencies>
```

JUnit 5 - IntelliJ

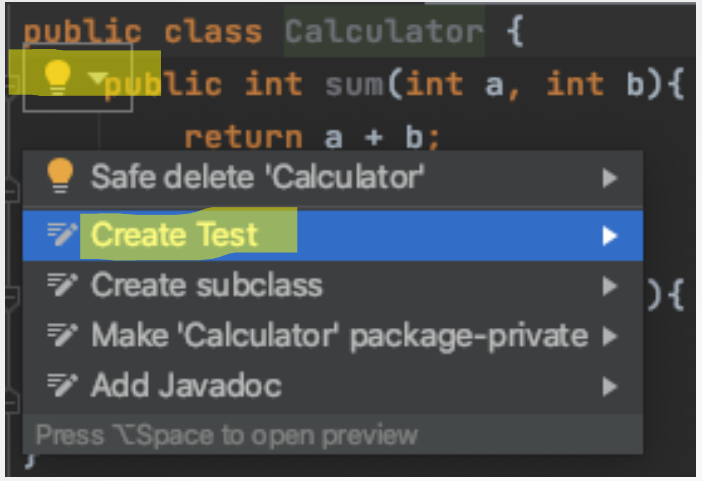
Let's begin by creating a simple Java Class that we want to test:

1

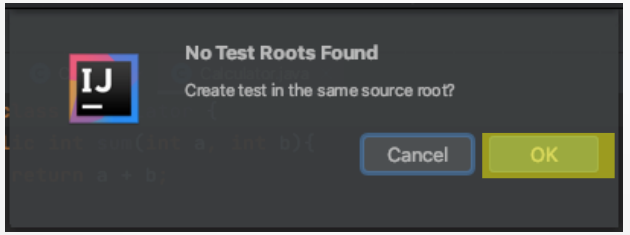
Calculator.java

```
public class Calculator {
    public int sum(int a, int b){
        return a + b;
    }

    public int mul(int a, int b){
        return a * b;
    }
}
```

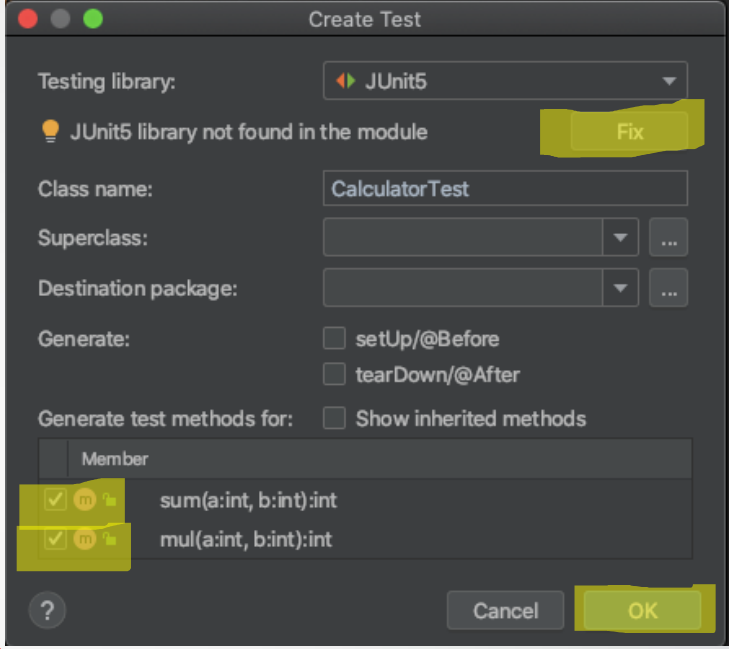
2



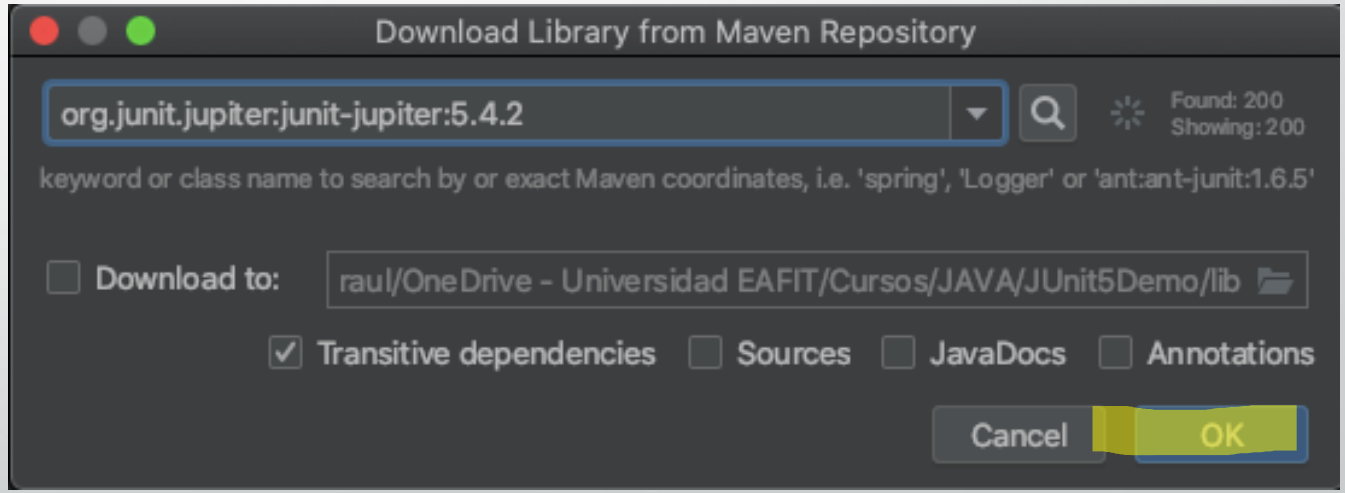
3



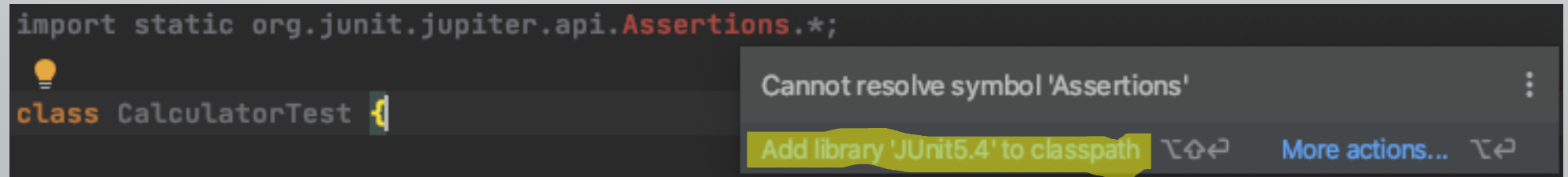
4



5



6



JUnit 5 – An in-depth example with IntelliJ

CalculatorTest.java

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    @DisplayName("Summation Test")
    void sum() {
        var c = new Calculator();
        assertEquals(5, c.sum(2, 3));
    }

    @Test
    @DisplayName("Multiplication Test")
    void mul() {
        var c = new Calculator();
        assertEquals(5, c.mul(2,3));//delta:0.02
    }
}
```

JUnit makes use of Java's annotation system to both denote tests and add properties to it. We must explicitly import these annotations for our tests to work.

Only the **@Test** annotation is obligatory, however. We add the **@DisplayName** annotation so that the results are reported in a clearer manner, and so that when tests fail, we have a clearer idea of what went wrong.

The main way we interact with JUnit is through ***Assertions***. They make explicit how we want our code to behave under the cases we envisaged in our tests. The ***assertEquals()*** method states that we expect that the result of the second parameter (the actual value from the computation) is equal to the first one (our *a priori* known expected value).

JUnit 5 – An in-depth example with IntelliJ

CalculatorTest.java

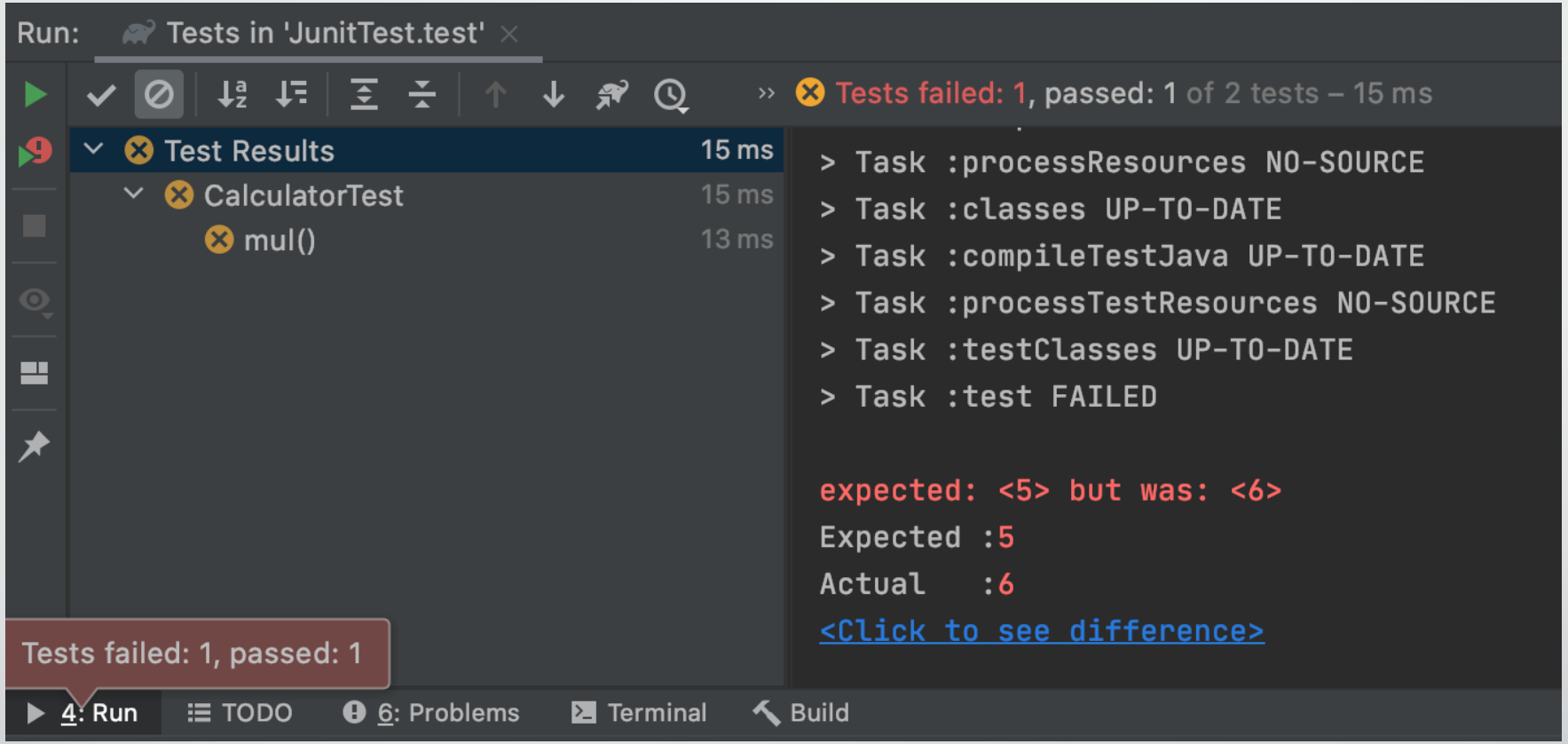
```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    @DisplayName("Summation Test")
    void sum() {
        var c = new Calculator();
        assertEquals(5, c.sum(2, 3));
    }

    @Test
    @DisplayName("Multiplication Test")
    void mul() {
        var c = new Calculator();
        assertEquals(5, c.mul(2,3));
    }
}
```

In general, we should always strive to **make all tests independent from one another**, otherwise the tests might become unmanageable as the application evolves. Here we do this by having a new instance of our **Calculator** in every test.

In this example the first test should pass **but** the second should fail, given the fact that 2*3=6 and not 5. When we run this test suite, we will now get some output indicating where our test failed and why.

To run the tests simply select the folder where the tests have been created and select ***Run All Tests '<project name'*** to open the test runner interface and see the results.

JUnit 5 – An in-depth example with IntelliJ



As expected, when we ran the tests shown in the preceding slides, we observe that our mul() method has failed. Though, in this case, it is due to an (deliberately) **incorrect test** and not a mistake made in the code of our Calculator. Once the test is fixed all tests will pass. If our issue were with our Calculator code, we would need to first reimplement the **mul()** method and then re-run our tests.

JUnit 5 – Some things to keep in mind

Unit-tests are not a silver-bullet; **if the tests you define are incorrect, you will not be able to infer anything valuable from them**.

You should choose your test cases with care: it is important that you **test values that might cause unexpected behaviors in your code**. (**Hint:** what happens if we give our methods the largest possible integer Java has defined for the **int** type? What happens if we pass null parameters? What happens if only one parameters is defined and not the other?)

JUnit is constantly evolving and adding new features: there is a whole world of features to explore in the documentation.

JUnit makes extensive use of newer Java features like lambda expressions (anonymous functions).

CalculatorTest.java

```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
	@ParameterizedTest(name="{0} + {1} = {2}")
	@CsvSource({
		 "1, 2, 3",
		"2, 3, 5",
		"3, 4, 7"
	})
	@DisplayName("Summation Test")
	void sum(int a, int b, int r) {
		var c = new Calculator();
		assertEquals(r, c.sum(a, b));
	}
	@Test
	@DisplayName("Multiply two numbers")
	void mul() {
		assertAll(
			() -> assertEquals(4, Calculator.mul(2, 2)),
			() -> assertEquals(-4,  Calculator.mul(2, -2)),
			() -> assertEquals(4,   Calculator.mul(-2, -2)),
			() -> assertEquals(0,   Calculator.mul(1, 0))); }
}
```

JUnit allows us to perform multiple tests at once in various ways. One of these is by creating **Parametrized Tests** that take input from a **source** and apply each set of arguments successively, effectively creating several tests with a single test declaration.

The **assertAll() method takes a series of assertions in form of lambda expressions** and ensures all of them are checked. This is more convenient than having multiple single assertions because you will always see a granular result rather than the result of the entire test.

For JUnit 5's User Guide please read
https://junit.org/junit5/docs/current/user-guide/

For JUnit 5's API documentation please read
https://junit.org/junit5/docs/current/api/

For the **practical work of this lecture** please follow this tutorial (for IntelliJ but also applicable to Eclipse) from JetBrains :
https://blog.jetbrains.com/idea/2020/09/writing-tests-with-junit-5/
Create a Gradle or Maven project and do not care about the configuration of the corresponding files, if you just start writing tests, IntelliJ IDEA will automatically detect if the dependency is missing and prompt you to add it.

To go further with much more exercises in Eclipse and IntelliJ:
https://www.vogella.com/tutorials/JUnit/article.html