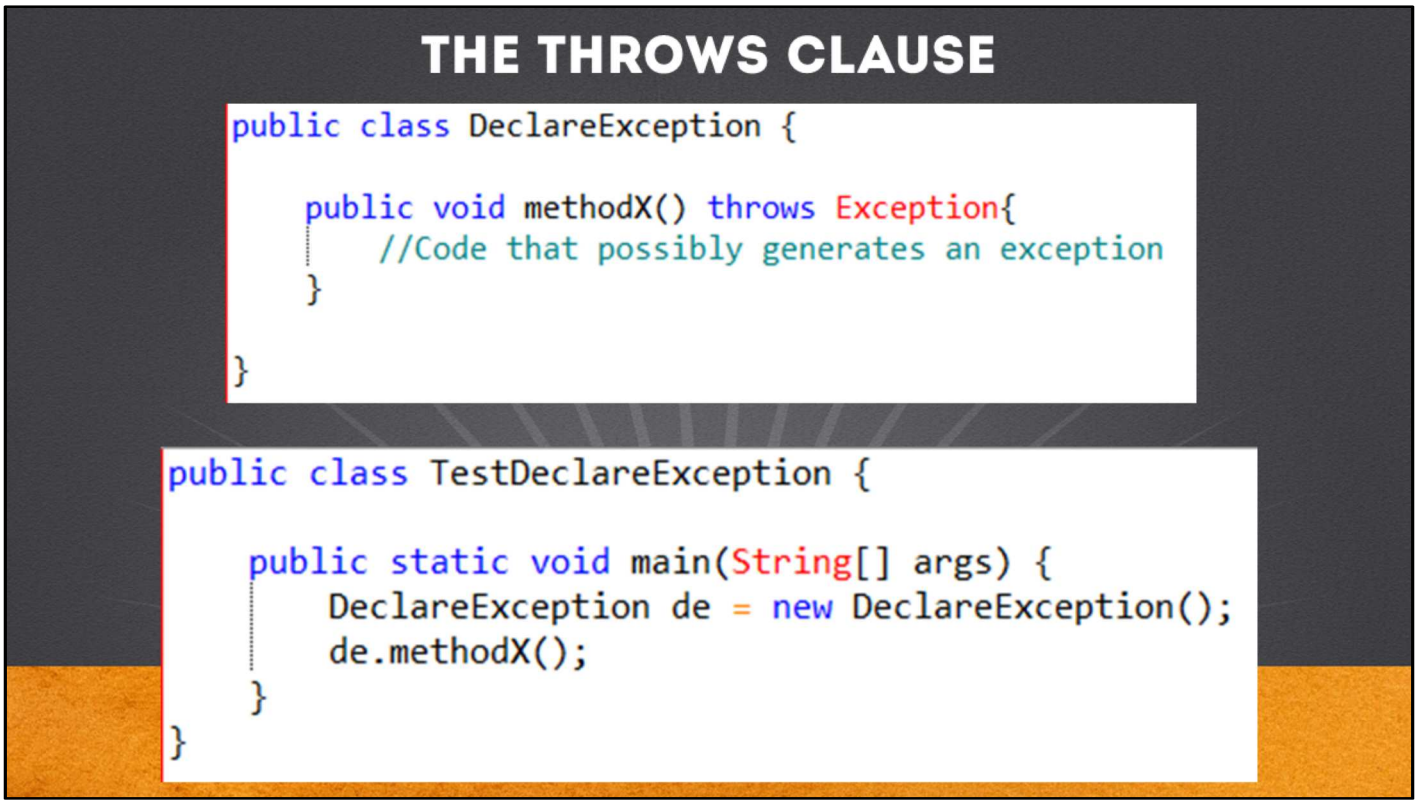
Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson ..

We are going to study more on the subject of exceptions in Java.

Are you ready? Come on!

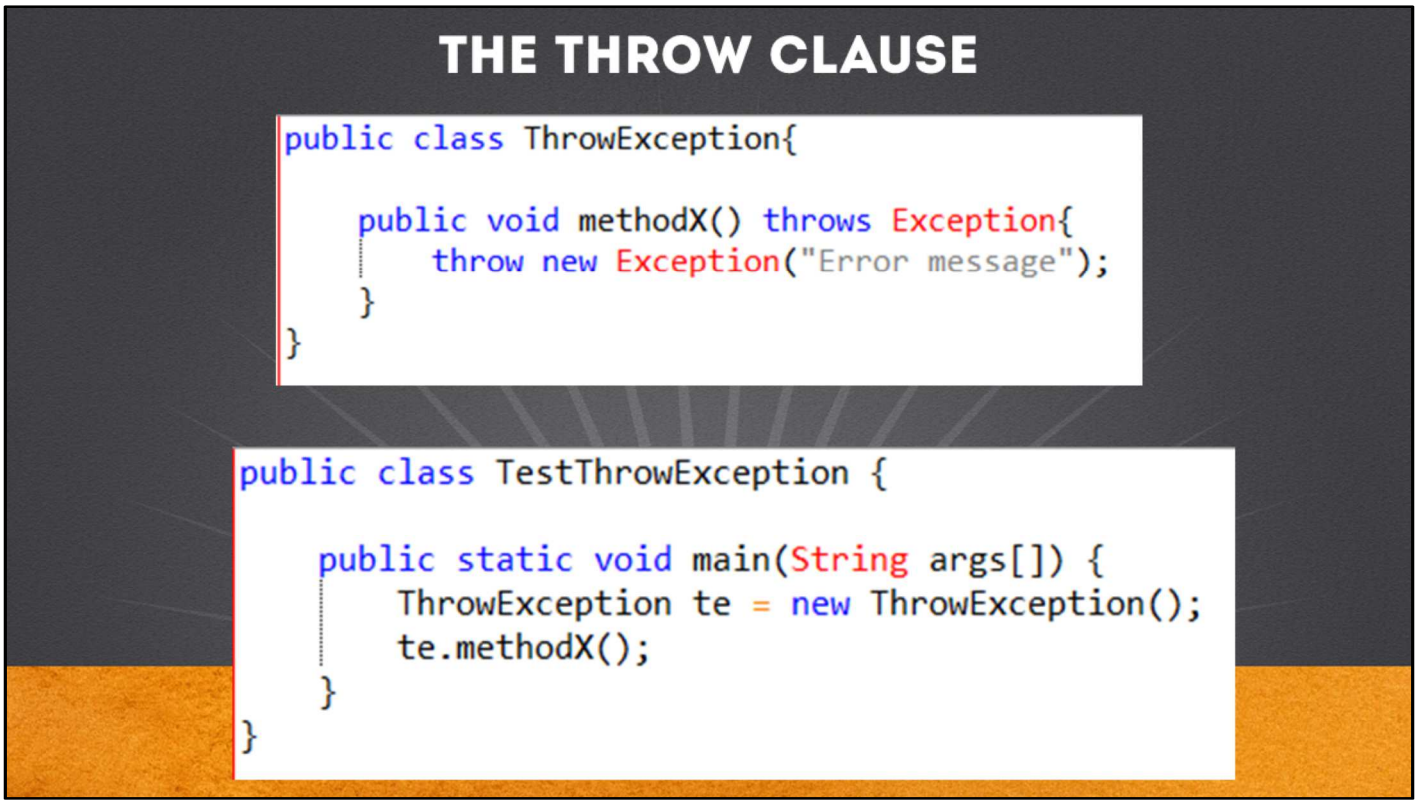


In this lesson we will continue to review more topics about exceptions in Java. Let's now talk about the throws clause, which allows us to specify the type of exception that throws a certain method.

This can be due to the fact that the method did not catch the exception and therefore it will spread to the method that calls it.

An exception class that extends Exception, we are required to declare it in the signature of the method, since this type of exceptions, as when we commented that they should be processed by a try / catch block, if they are not processed by this block, then we are forced to declare them within the method where the method that throws the exception is used.

An exception that extends from RuntimeException is NOT required to declare it in the signature of the method when using a method that throws this type of exception.



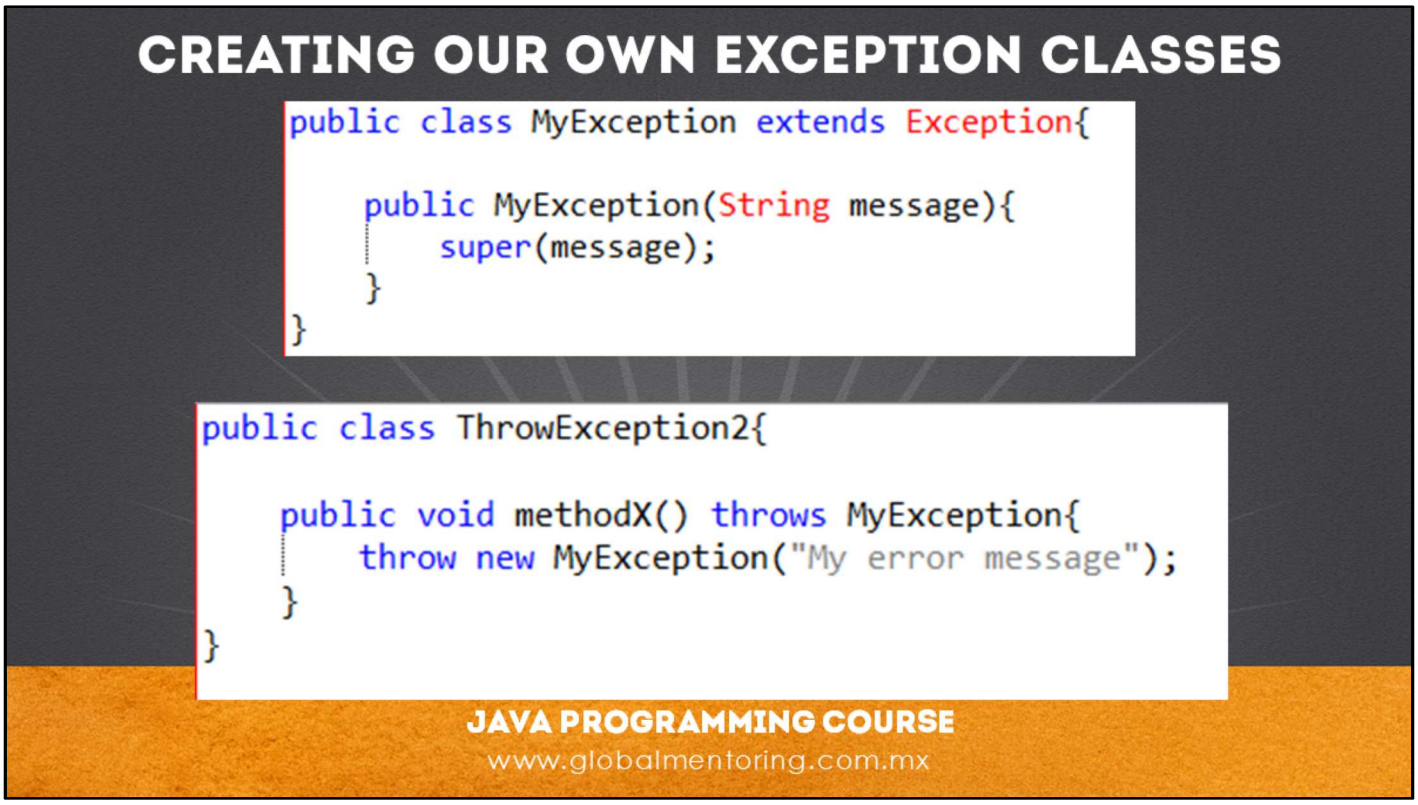
Let's now review the throw clause. This reserved word allows us to explicitly throw an exception.

For example, we can use an exception class that extends the class Exception or any other class of type exception and together with the operator new, create a new object of this class and thus throw an exception explicitly.

In the code we can observe how within the methodX a new exception of type Exception is thrown, using the reserved word throw followed by new and the class of type Exception that we want to instantiate. Normally the exception classes in your constructor have a string that will be the message that will be sent to the stacktrace at the time the problem occurs or the exception is thrown.

Later what we observe is the main method when using methodX requires processing the exception by means of a try / catch block, or to declare in the method signature that methodX can get to throw the Exception type exception, and we see in the signature of the method by means of the word throws Exception.

In fact, with a similar process is how our application can convert exceptions of type Exception (checked exceptions) to type RuntimeException (uncheked exceptions), creating our own classes of type runtime exception and when wrapping an exception of type Exception with a block try / catch, you can relaunch an exception of type runtime exception, this way we do not force the methods that use our methods to declare the type of exception that can be thrown, since they have been converted to type runtime exception. Later we will see how to perform this conversion.

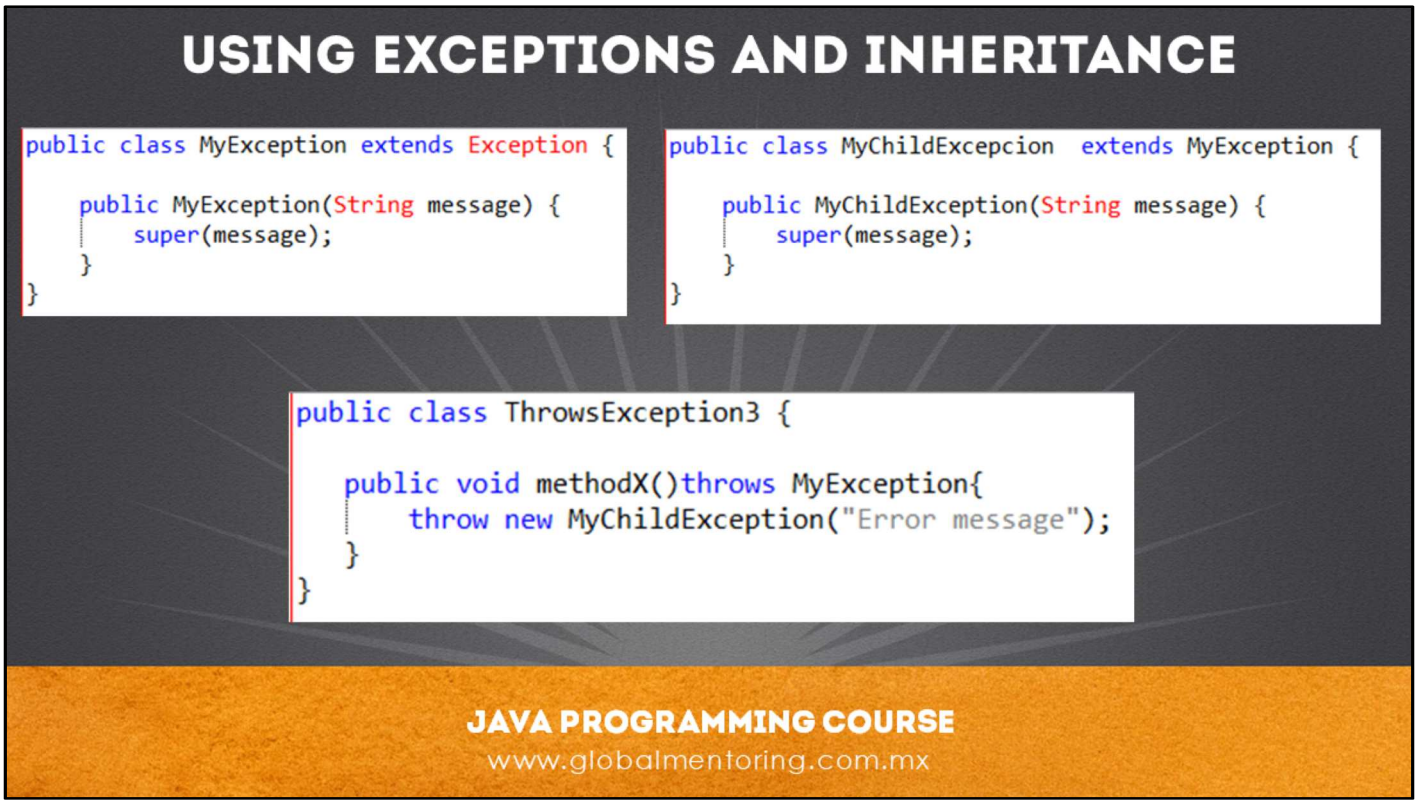


Let's see next how we can create our own exception classes, which can extend from the class Exception, or RuntimeException.

The intention to create our own Exception class is to personalize the error messages that are sent to the user, for example, a Database error, it can be trapped and converted into a readable error message to the user using our application, or As we have said before, we can create exception class that descend from the runtime exception class to not force users to use try / catch blocks or declare in the method signatures the type of exception that will be thrown in case of any exception of the type that we have declared.

As we observed in the code, we are creating a class of type MyException which inherits from the Exception class, however it can extend from any other kind of exception that interests us. Later this class contains a constructor which receives the message that will help us to initialize the message attribute of the parent class, by calling the parent constructor by means of the word super. And with this, our exception is initialized.

Later in the class DropException2 we can see that we throw an exception of type MyException, and since it is of type Excepcion, the methodX is forced to declare in its signature the type of exception that will be thrown in case of error. This is how we will create our own exception class and that is also how we will use them within our Java code.



Let's now review the issue of using exceptions and inheritance. In the code shown we see two classes, one called MyException which extends from the Exception class. Later we declare a class called MyChildExcepcion which is a subclass of MyException, therefore we have an exception class hierarchy that we have created.

On the other hand, in the class ThrowsException3, a subclass called MyChildException is thrown within the methodX, but if we observe the signature of the method, the method declares that an exception is thrown that is a superclass called MyException.

The intention to do this is that the method can internally throw exceptions that are subclasses, but the class that uses this method only processes the most generic exception, and with that we can have a wider range for the handling of exceptions, using precisely the concept of inheritance but now within the definition of classes of type Exception, and like when we work with polymorphism, much of the code in any programming language and also in Java, is to create methods as generic as possible that help us to encompass more possible exceptions in less lines of code, which in the end will save us the maintenance costs of our applications, by having the most generic and compact code possible.

Let's see below the use of exceptions when overloading our methods in Java. Some of the rules that we must respect are:

A method that overwrites a parent method that declares an exception CAN:

- Do not throw an exception, that is, omit the exception in your signature.
- Throw one or more exceptions thrown by the parent method.
- Throw one or more exceptions that are subclasses of the exception thrown.

A method that overwrites a parent method that declares an exception CAN NOT:

- Throw additional exceptions not thrown in the parent method.
- Throw superclasses of the exceptions thrown by the parent method.

In the code shown, we observe a parent class (ThrowExA) and two child classes (ThrowExB and ThrowExc), which overwrite the methodX defined in the parent. What we are clarifying in this code are the exceptions that we can use in signing the methods overwritten by the daughter classes.

On the one hand we observe that the class ThrowExB in the signature of the methodX instead of using the same exception as in the signature of the method of the parent class (MyException), a subclass (MyChildException) is used, in this way it is clear that not necessarily we must use the same kind of exception in the signature of the method, but it can be some class within the hierarchy of exception classes that we are using, being always subclasses, and not superclasses.

On the other hand, we observed in the ThrowExc class that a super class (Exception) of the exception class used in the parent (MyException) is used, and therefore this is not valid and there is a compilation problem in our code. Later we will see an example of this type of exception handling in the signature of the methods that are overwritten.