Contenido

[JAVA 2](#_Toc127905372)

[El JDK 2](#_Toc127905373)

[EL JRE 2](#_Toc127905374)

[Lo que aprendimos 3](#_Toc127905375)

[Para saber más: Type Casting 4](#_Toc127905376)

[ATAJOS 5](#_Toc127905377)

[JAVA – PROGRAMACION ORIENTADA A OBJETOS (POO) 6](#_Toc127905378)

[CLASS: 6](#_Toc127905379)

[OBJETO: 6](#_Toc127905380)

[INSTANCIA 6](#_Toc127905381)

[CONSTRUCTOR: 6](#_Toc127905382)

[Metodo: 7](#_Toc127905383)

[Modificadores de acceso: 7](#_Toc127905384)

[Métodos Getters y Setters 7](#_Toc127905385)

[Generar de manera automática los Setter y los Getter de las variables de la clase. 7](#_Toc127905386)

[HERENCIA 8](#_Toc127905387)

[POLIMORFISMO: 9](#_Toc127905388)

[CLASE BASTRACTA (abstract) 9](#_Toc127905389)

[Modificadores de acceso 11](#_Toc127905390)

[**Public** 12](#_Toc127905391)

[**Protected** 12](#_Toc127905392)

[**Private** 12](#_Toc127905393)

# JAVA

El JDK son las herramientas de desarrollo (como el compilador) y también tiene el JRE incluido.

* El JDK son las herramientas de desarrollo (como el compilador) y también tiene el JRE incluido.

EL JRE es el ambiente para ejecutar una aplicación Java

* Si solo queremos ejecutar una aplicación Java, el JRE (Java Runtime Environment) es suficiente.

El mundo Java está lleno de siglas de 3 o 4 letras comenzando con la letra J. Ya debes conocer las dos más famosas, JRE y JDK. El primero es el ambiente de ejecución, el segundo son las herramientas de desarrollo junto con el ambiente de ejecución. Simplificando, podemos decir que:

* JDK = JRE + Herramientas de desarrollo

Existe una tercera sigla, **JVM** (Java Virtual Machine), que también hemos usado durante el curso. La responsabilidad de la Java Virtual Machine es ejecutar el Bytecode, entonces ¿Cuál es la diferencia entre el JRE y la JVM si ambos ejecutan Bytecode?

La respuesta es simple: El JRE (Nuestro ambiente de ejecución) contiene la JVM, pero también posee varias librerías incluidas. Es decir, para ejecutar una aplicación Java no solo es necesario tener la JVM, también necesitamos de otras librerías.

Entonces, podemos simplificar:

* JRE = JVM + Librerías

Es importante resaltar que no se puede descargar la JVM individualmente. Siempre tendremos que descargar el JRE que tiene la JVM y el conjunto de librerías.

Para imprimir un hola mundo

public class Ejemplo

{

public static void main(String[] args)

{

System.out.println(“Hola Mundo”);

}

# Lo que aprendimos

En esta clase escribiste tu primer código Java y aprendimos:

* Cuál es la diferencia entre JRE y JDK.
* Cómo compilar el código fuente de Java desde la línea de comandos (javac).
* Cómo ejecutar Bytecode en la línea de comando (java).
* Un programa Java debe escribirse dentro de una clase (class).
* Cada instrucción Java debe terminar con ;.
* Para abrir y cerrar un bloque usaremos las llaves {}.
* Un programa Java tiene una entrada que es una función (método) main.
* Para imprimir algo en la consola, usamos la declaración System.out.println ().

# Para saber más: Type Casting

Como se ve en los videos, cuando intentamos poner un valor entero en una variable de tipo double, Java no muestra un error. Sin embargo, cuando intentamos poner un doble en una variable del tipo entero, tenemos un error de compilación.

Esta propiedad se produce porque Java convierte implícitamente de un tipo más pequeño a tipos "más grandes". De entero a double, por ejemplo.

Lo contrario no es cierto porque hay pérdida de datos cuando se realiza la conversión. Resultando en un " type mismatch" que muestra que esta instrucción es de tipos incompatibles.

Para realizar una conversión donde puede haber pérdida de información, es necesario hacer un type casting. Vea las instrucciones a continuación.

int edad = (int) 30.0;

En el caso anterior, es explícito que se realizará la conversión de doublé a entero. Vea cómo funciona el cast implícito y explícito en la tabla a continuación.

| **DE/PARA** | **byte** | **short** | **char** | **int** | **long** | **float** | **double** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| byte | ---- | *Impl.* | (char) | *Impl.* | *Impl.* | *Impl.* | *Impl.* |
| short | (byte) | ---- | (char) | *Impl.* | *Impl.* | *Impl.* | *Impl.* |
| char | (byte) | (short) | ---- | *Impl.* | *Impl.* | *Impl.* | *Impl.* |
| int | (byte) | (short) | (char) | ---- | *Impl.* | *Impl.* | *Impl.* |
| long | (byte) | (short) | (char) | (int) | ---- | *Impl.* | *Impl.* |
| float | (byte) | (short) | (char) | (int) | (long) | ---- | *Impl.* |
| double | (byte) | (short) | (char) | (int) | (long) | (float) | ---- |

Para comparar cada tipo primitivo más claramente, la siguiente tabla muestra el tamaño de cada uno.

| **TIPO** | **TAMAÑO** |
| --- | --- |
| boolean | 1 bit |
| byte | 1 byte |
| short | 2 bytes |
| char | 2 bytes |
| int | 4 bytes |
| float | 4 bytes |
| long | 8 bytes |
| double | 8 bytes |

# ATAJOS

Para colocar el siguiente método, indicamos la combinación de teclas: escribo main + 'Control' + 'espacio'

public static void main(String[] args)

{

System.***out***.println("Hello mundo");

int edad = 16;

int cantidad = 1;

if (edad >= 18)

{

System.***out***.println("Puede ingresar");

{

}

escribo Sysout +'Control' + 'espacio'

System.***out***.println()

ctrl + shift + f, para formatear el código fuente.

# JAVA – PROGRAMACION ORIENTADA A OBJETOS (POO)

La Programación Orientada a objetos permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de software utilizado por muchos programadores DRY (Don't Repeat Yourself), para evitar duplicar el código y crear de esta manera programas eficientes.

¿Cómo llamamos, en orientación a objeto, las características de una clase?

* Atributo

## CLASS:

Es un molde para crear objetos o instancias.

## OBJETO:

Es una instancia de la clase.

INSTANCIA:

Son las representaciones del objeto, utiliza la estructura del objeto para representar ese objeto con datos.

* Es una extensión del contenido del objeto con datos.

CONSTRUCTOR:

Un constructor es un elemento de una clase cuyo identificador coincide con el de la clase correspondiente y que tiene por objetivo obligar a y controlar cómo se inicializa una instancia de una determinada clase, ya que el lenguaje Java no permite que las variables miembros de una nueva instancia queden sin inicializar.

## Metodo:

Un método en Java es un conjunto de instrucciones definidas dentro de una clase, que realizan una determinada tarea y a las que podemos invocar mediante un nombre.

<https://www.aluracursos.com/blog/poo-que-es-la-programacion-orientada-a-objetos>

Modificadores de acceso: public (Hace que sea visible desde cualquier parte del código) or private (Solo es visible con la clase que esta trabajando).

## Métodos Getters y Setters

Hasta ahora sólo hemos creado una clase persona, la cual podemos crear cuantos objetos queramos construyéndolos a través de su constructor. También tenemos un método que visualiza todos los datos del objeto (persona) y uno que dice si es mayor o menor de edad, pero si volvemos a revisar la clase al momento de querer visualizar u obtener un dato de la persona no se puede ya que los atributos están declarados como privados y no se puede acceder a esa información. ¿Cómo es posible obtener la información de algún atributo de una clase?.

Hay métodos que no son de utilidad para obtener o almacenar algún valor de un atributo de una clase, comúnmente son llamados getters y setters.

Los getters (de la palabra inglés get - obtener) indica que podemos tomar algún valor de un atributo y los setters (de la palabra ingles set-poner/fijar) podemos guardar algún valor sobre un atributo. Son importantes al momento de crear una clase objeto ya que de ellos dependen el valor que pueden tomar los atributos o para modificar algún atributo sin necesidad de modificar algún otro atributo.

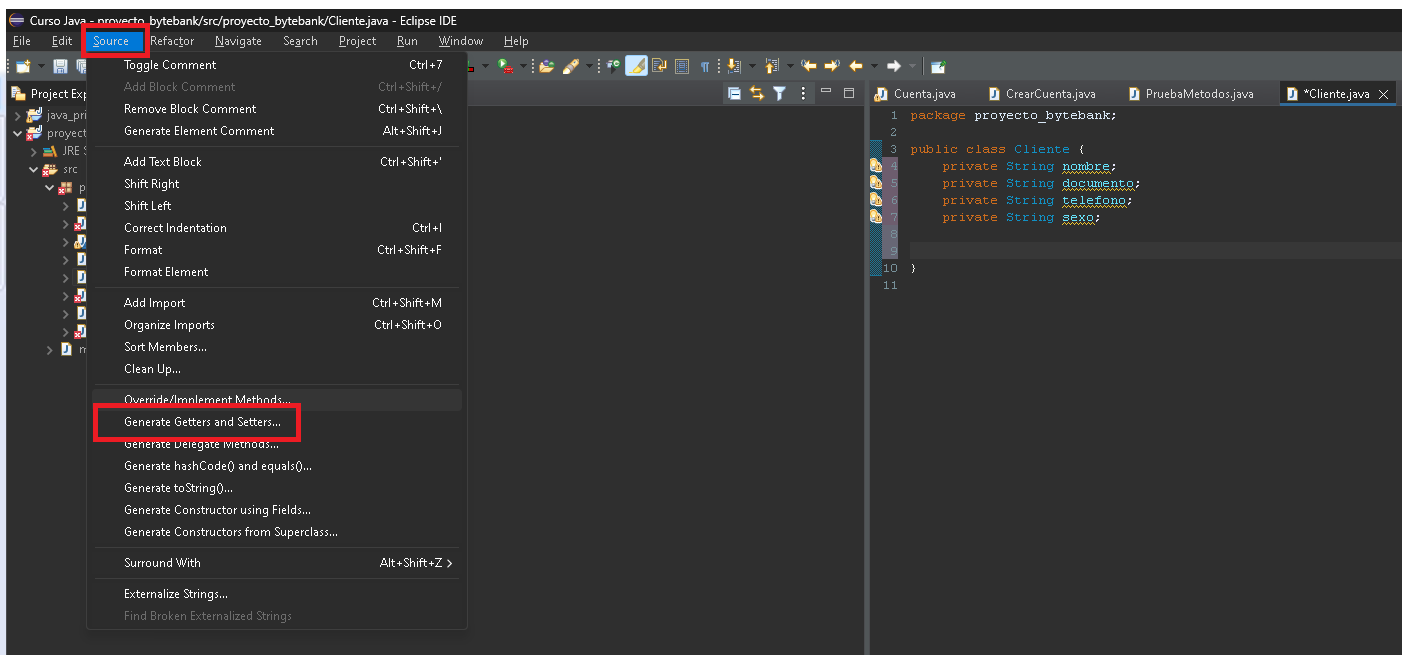
## Generar de manera automática los Setter y los Getter de las variables de la clase.

\*\*Solo aplica para eclipse

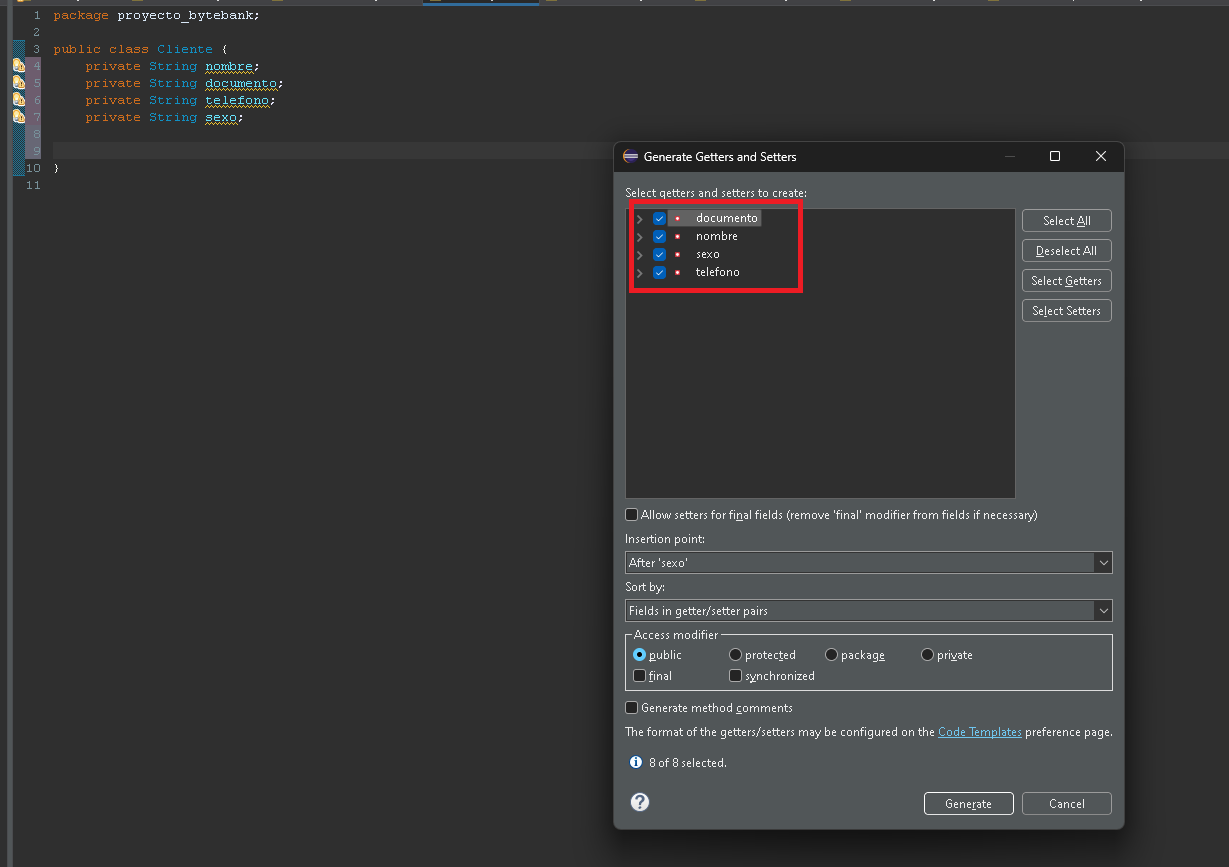
Nos paramos en la clase respectiva

Vamos a SOURCE

Luego vamos a la opción Generate Getter and Setter



Luego seleccionamos las variables a las que queremos generar los Setter y Getter y después damos al botón Generate



## HERENCIA

Se utiliza la palabra reservada “extends” para heredar las variables y métodos de la clase padre.

La herencia **en Java** implica que una superclase o clase base hereda sus funciones y atributos a una subclase o clase derivada. La palabra reservada que nos permite realizar herencia entre clases es extends.

public class Gerente extends Funcionario {}

## POLIMORFISMO:

En Java, el polimorfismo es la capacidad de un objeto para tomar muchas formas. En otras palabras, un objeto puede ser tratado como si fuera una instancia de varias clases diferentes. Esto permite que las clases compartan comportamientos comunes, mientras que todavía mantienen su propia funcionalidad única.

Hay dos tipos de polimorfismo en Java: el polimorfismo de compilación y el polimorfismo de ejecución.

Polimorfismo de compilación: Se refiere al uso de sobrecarga de métodos y constructores. En tiempo de compilación, el compilador selecciona el método o constructor adecuado para su uso, en función de la lista de argumentos utilizados.

Polimorfismo de ejecución: Se refiere al uso de métodos polimórficos que son sobrescritos en una clase derivada. En tiempo de ejecución, el método llamado se decide en función del tipo de objeto al que se refiere la variable de referencia.

## CLASE BASTRACTA (abstract)

Las clases abstractas, como su nombre lo indica, son algo abstracto, no representan algo específico y las podemos usar para crear otras clases. No pueden ser instanciadas, por lo que no podemos crear nuevos objetos con ellas.

Podemos imaginar una clase Animal, con métodos como caminar y comer, como una clase base que podemos heredar para construir otras clases como León o Pájaros. Ambas van a heredar de nuestra clase animal con sus respectivos métodos y tendremos la posibilidad de crear nuestros objetos. De esta manera podemos reducir código duplicado y mejorar la calidad del código.

En Java declaramos una clase abstracta con la palabra reservada abstract.

**public** **abstract** **class** **Animal** {

**public** **Animal**(String value){

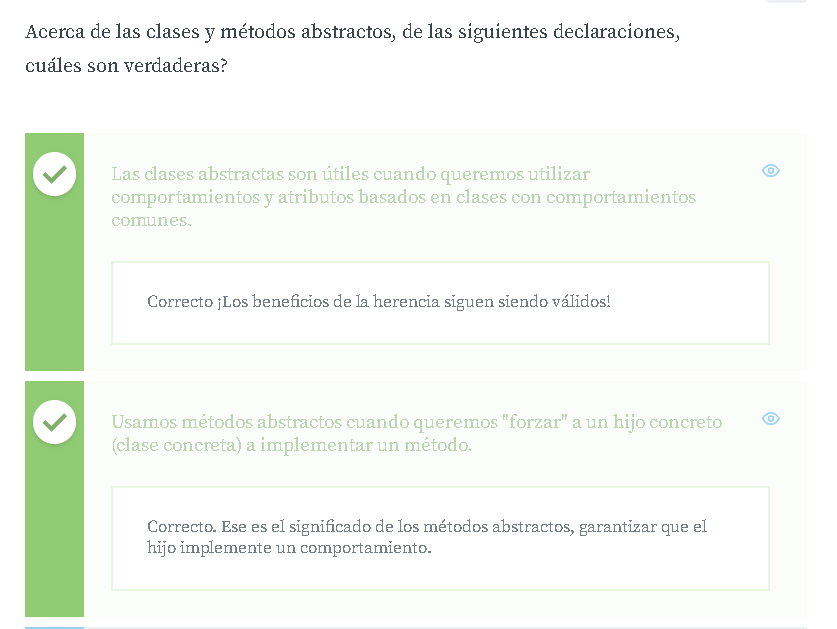
// Constructor

**this**.value = value;

}

**public** **abstract** **void** **sound**()

}



¿Qué es cierto sobre las clases abstractas? Seleccione todas las declaraciones verdaderas:



# **Modificadores de acceso**

Los modificadores de acceso o accesibilidad son algunas palabras claves utilizadas en el lenguaje Java para definir el nivel de accesibilidad que los elementos de una clase (atributos y métodos) e incluso la propia clase puede tener los mismos elementos de otra clase.

## **Public**

Este es el modificador menos **restrictivo** de todos. De esta manera, cualquier componente puede acceder a los miembros de la clase, las clases y las interfaces.

## **Protected**

Al usar este modificador de acceso, los miembros de la clase y las clases son accesibles para otros elementos siempre que estén dentro del mismo package o, si pertenecen a otros packages, siempre que tengan una relación extendida (herencia), es decir, las clases secundarias pueden acceder a los miembros de su clase principal (o clase de abuelos, etc.).

## **Private**

Este es el modificador de acceso **más restrictivo** de todos. Solo se puede acceder a los miembros definidos como privados desde dentro de la clase y desde ningún otro lugar, independientemente del paquete o la herencia.

Hagas clic [aquí](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html) para acceder a la documentación oficial y obtener más información.

# Try Catch and finally

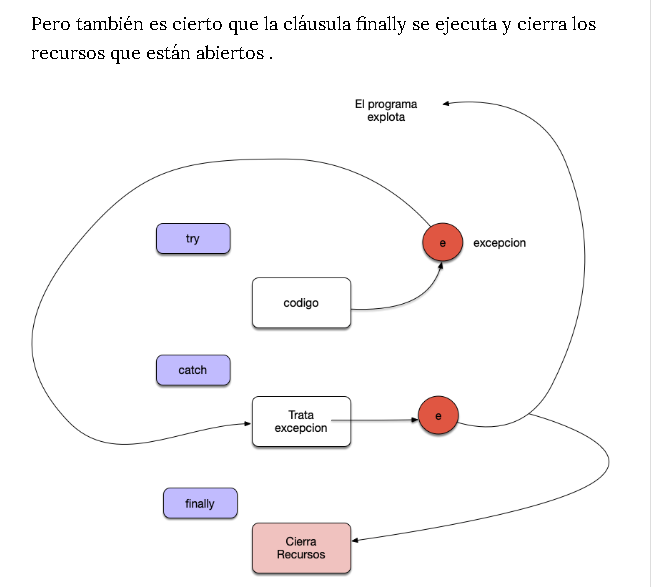
El try catch en programación se utiliza para manejar fragmentos de código que son propensos a fallar, como puede ser: recibir un valor nulo, convertir un tipo de dato a otro o en tu caso, la lectura de un archivo.

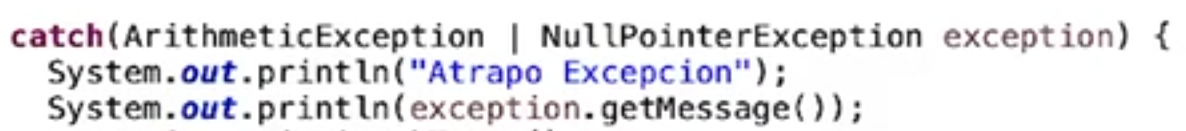
Al leer un archivo que está almacenado en la memoria del dispositivo (interna o externa), se genera un flujo de bits (físicamente corriente) desde dicha memoria a la memoria RAM. Ese flujo puede verse interrumpido por diferentes razones, lo que causaría que la tarea falle y que el programa se detenga. El try catch lo que hace es asegurarse que aunque la tarea que se esté ejecutando falle, el programa se siga ejecutando y no se detenga.

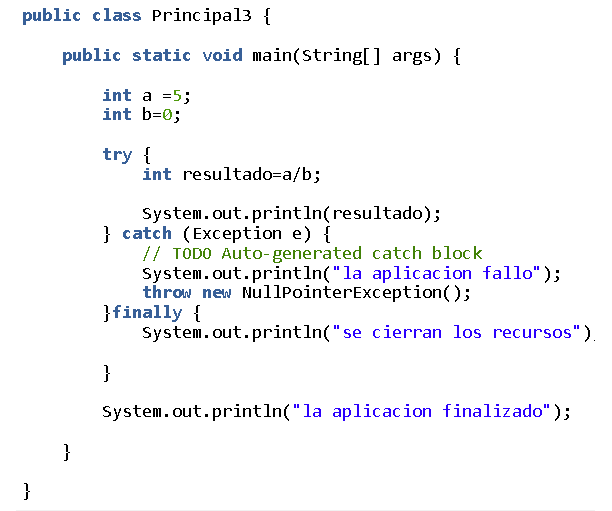
Si el código que está dentro del try falla, se ejecuta el catch y el programa se sigue ejecutando. Dentro del try se debe de colocar el código que es propenso a fallar y dentro del catch el código a ejecutarse si el try falla, como puede ser un mensaje de error.

En muchas ocasiones la gente suele preguntar para qué es necesario usar  java finally . La respuesta es que finally asegura siempre el cierre de los recursos y se ejecuta ocurra lo que ocurra en el programa. Por ejemplo supongamos que se produce una excepción dentro de la clausula catch.

Atrapar dos excepciones.







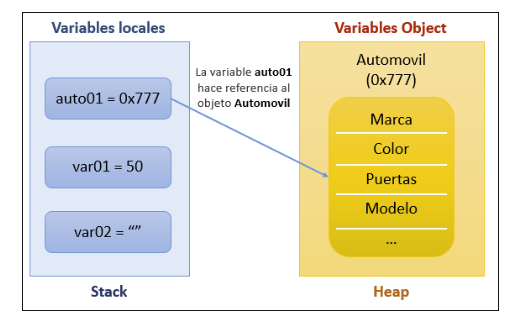
## MEMORIA STACK

La memoria Stack se usa para almacenar las variables locales (cuyo ámbito de acción está limitada solo a la función donde se declaró) y también las llamadas de funciones en Java. Las variables almacenadas en esta memoria por lo general tienen un periodo de vida corto, viven hasta que terminen la función o método en el que se están ejecutando.

## MEMORIA HEAP

La memoria Heap es utilizada para almacenar los objetos (incluyendo sus atributos), los objetos almacenados en este espacio de memoria normalmente tienen un tiempo de duración más prolongado que los almacenados en Stack.

Hasta este punto, ya debes haber recordado que las variables de un programa no almacenan el objeto, más bien guardan la referencia del objeto.



LANZAR UNA EXCEPCION

private static void metodo2() {

System.out.println("Inicio del metodo2");

throw new ArithmeticException();

}

LANZAR UNA EXCEPCION CON MENSAJE PERSONALIZADO.

private static void metodo2() {

System.out.println("Inicio del metodo2");

throw new ArithmeticException("Surgió un error");

}

## Estructuras de excepciones.

Si hiciste el ejercicio ¿será Miguel comprendió la clase ?, recordará lo que aprendimos. Para solucionarlo aún más, enumeramos los temas de esta clase:

* Existe una gran jerarquía de clases que representan excepciones. Por ejemplo, *ArithmeticException* es hija de *RuntimeException*, que hereda de *Exception*, que a su vez es hija de la clase de excepciones más ancestral, *Throwable*. Conocer bien esta jerarquía significa saber cómo usar las excepciones en su aplicación.
* *Throwable* es la clase que debe extenderse para poder lanzar un objeto en la pila (usando la palabra reservada *throw*)
* Es en la clase *Throwable* donde tenemos casi todo el código relacionado con las excepciones, incluyendo *getMessage()* e *printStackTrace ()*. El resto de la jerarquía tiene solo algunas sobrecargas de constructores para comunicar mensajes específicos.
* La jerarquía que comenzó con la clase *Throwable* se divide en excepciones y errores. Las excepciones se utilizan en los códigos de aplicación. Los errores son utilizados exclusivamente por la máquina virtual.
* Las clases que heredan de *Error* se utilizan para informar errores en la máquina virtual. Los desarrolladores de aplicaciones no deben crear errores que hereden de *Error*.
* *StackOverflowError* es un error de máquina virtual para informar que la pila de ejecución no tiene más memoria.
* Las excepciones se dividen en dos categorías amplias: las que el compilador comprueba obligatoriamente y las que no.
* Los primeros se denominan *checked* y se crean por pertenecer a una jerarquía que no pasa *RuntimeException*.
* Los segundos están *unchecked* y se crean como descendientes de *RuntimeException*.

# Haga lo que hicimos en el aula: Finally y try with resources

Hagamos algunas pruebas ahora.

1) Empiece por crear la clase *Conexion*:

public class Conexion {

public Conexion () {

System.out.println("Abriendo conexión");

}

public void leerDatos() {

System.out.println("Recibiendo datos");

// throw new IllegalStateException();

}

public void cerrar() {

System.out.println("Cerrando conexión ");

}

}

2) Ahora, para probar nuestra conexión, necesitaremos crear otra clase, TestConexion. No olvide el método principal:

public class TestConexion {

public static void main(String[] args) {

}

}

3) Dentro del método main, de nuestra clase recién creada, pondremos un bloque try y catch para hacer uso de nuestra conexión:

try{

Conexion con = new Conexion();

con.leerDatos();

con.cerrar();

} catch(IllegalStateException ex){

System.out.println("Surgio un error en la conexión");

}

4) No olvide descomentar la línea respectiva dentro de la clase Conexion.

5) Queremos cerrar nuestra conexión siempre, incluso en caso de errores, por lo que debemos asegurarnos de que siempre se llame al método con.cerrar(). Para ello, tenemos el bloque *finally*. Haz lo siguiente:

Conexion con = null;

try{

con = new Conexion();

con.leerDatos();

con.cerrar();

} catch(IllegalStateException ex){

System.out.println("Surgió un error en la conexión ");

} finally {

con.cerrar();

}

6) Todavía podemos mejorar nuestro código. Para ello, haremos la declaración de conexión entre paréntesis del try, aplicando lo siguiente:

try(Conexion conexion = new Conexion() ){

}

7) Debido a la declaración anterior, necesitamos que nuestra clase Conexion implemente la interfaz AutoCloseable y el método close(). Entonces, lo dejaremos de la siguiente manera:

public class Conexion implements AutoCloseable{

public Conexion() {

System.out.println("Abriendo Conexion");

}

public void leerDatos() {

System.out.println("Recibiendo datos");

throw new IllegalStateException();

}

@Override

public void close() {

System.out.println("Cerrando Conexion");

}

}

8) Ahora, llamaremos al método leerDados(), dentro de nuestro nuevo try, comentando el código anterior.

try(Conexion conexion = new Conexion () ){

conexion.leerDatos();

}

9) Todavía tenemos que hacer nuestro nuevo catch también, que se verá así:

try(Conexion conexion = new Conexion () ){

conexion.leerDatos();

} catch(IllegalStateException ex){

System.out.println("Surgió un error en la conexión ");

}

En el video, usamos la excepción *IllegalStateException*, que es parte de la biblioteca de Java e indica que un objeto tiene un estado invalido. Es posible que haya visto otras excepciones famosas, como *NullPointerException*. Ambos son parte de las excepciones padrones del mundo Java que el desarrollador debe conocer.

Hay otra excepción padrón importante que podríamos haber utilizado en nuestro proyecto. Para ponerlo en contexto, ¿tiene sentido crear una cuenta con una agencia que tiene un valor negativo? Por ejemplo:

Cuenta c = new CuentaCorriente (-111, 222); //¿tiene sentido?

No tiene ningún sentido. Entonces, podríamos verificar los valores en el constructor de la clase. Si el valor es incorrecto, lanzamos una excepción. ¿Cual? La *IllegalArgumentException*, que es parte de las excepciones de la biblioteca de Java:

public abstract class Cuenta {

//atributos omitidos

public Cuenta(int agencia, int numero){

if(agencia < 1) {

throw new IllegalArgumentException("Agencia inválida");

}

if(numero < 1) {

throw new IllegalArgumentException("Número de cuenta inválido");

}

//resto del constructor fue omitido

}

}

*IllegalArgumentException* e *IllegalStateException* son dos excepciones importantes que el desarrollador de Java debería utilizar. En general, cuando tenga sentido, utilice una excepción estándar en lugar de crear la suya propia.

¿Todo bien hasta ahora?