

Semana 02

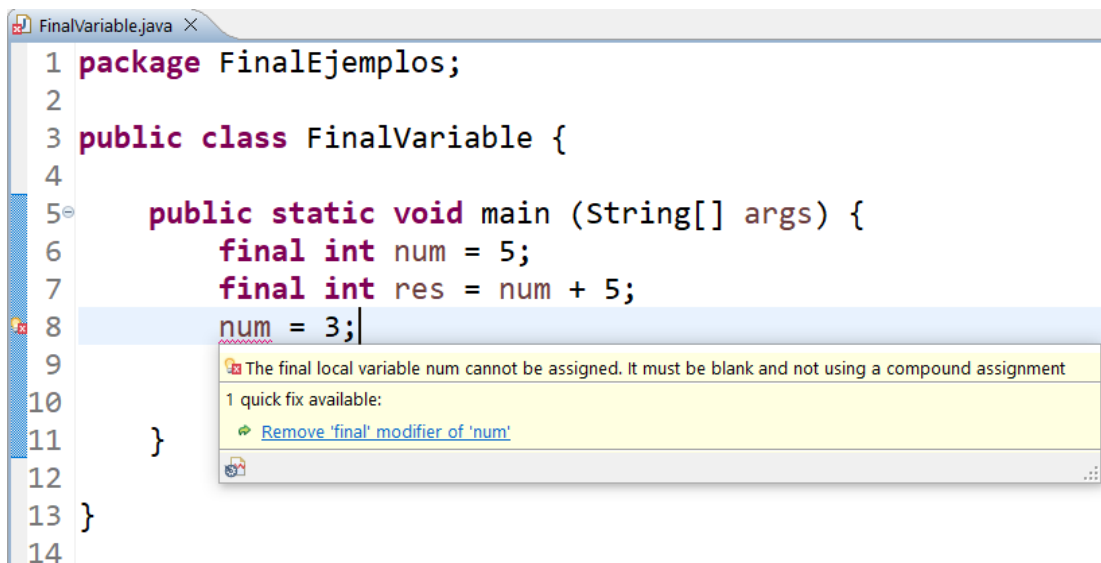
Explica los 4 propósitos del final

¿Qué hace la palabra clave final?

La palabra reservada *final* hace que el compilador genere un error cuando a una variable final se le intenta asignar un nuevo valor. El error de compilación quizá en algún caso evita un error ya que no debería haber intención de asignar un nuevo valor a una variable final.

La palabra reservada *final* se usa en 4 diferentes formas:

- En la declaración de **variables**: permite declarar constantes, no se puede asignar un nuevo valor a la variable una vez inicializada.
- En la declaración de **atributos**: similar a la inicialización de variables, este puede esperar al valor pasado por el constructor para obtener un valor.



```
1 package FinalEjemplos;
2
3 public class FinalVariable {
4
5     public static void main (String[] args) {
6         final int num = 5;
7         final int res = num + 5;
8         num = 3;
9
10    }
11
12 }
13
14 }
```

The final local variable num cannot be assigned. It must be blank and not using a compound assignment

1 quick fix available:

- Remove 'final' modifier of 'num'

- En la declaración de un **método**: en este contexto una clase que herede un método final no puede redefinirlo en la clase hija, no se puede hacer un *override*.

```

1 package Metodo;
2
3 public class Persona {
4     private String nombre;
5     private int edad;
6
7     public Persona(String nombre, int edad) {
8         this.nombre = nombre;
9         this.setEdad(edad);
10    }
11
12    public void imprimirDatosPersonales() {
13        System.out.println("Nombre:" + nombre);
14        System.out.println("Edad:" + getEdad());
15    }
16
17    final public boolean esMayor() {
18        if (getEdad() >= 18)
19            return true;
20        else
21            return false;
22    }
23
24    public int getEdad() {
25        return edad;
26    }
27

```

```

1 package Metodo;
2
3 public class Empleado extends Persona {
4     private int sueldo;
5
6     public Empleado(String nombre, int edad, int sue
7         super(nombre, edad);
8         this.sueldo = sueldo;
9     }
10
11    public void imprimirSueldo() {
12        System.out.println("El sueldo es:" + sueldo)
13    }
14
15    public boolean esMayor() {
16        if (getEdad() >= 6)
17            return true;
18        else
19            return false;
20    }
21 }

```

Cannot override the final method from Persona
1 quick fix available:
Remove 'final' modifier of 'Persona.esMayor()'

- En la declaración de una **clase**: impide extender de la clase.

```

1 package FinalEjemplos;
2
3 public final class FinalClase {
4
5     public static void main(String[] args) {
6
7     }
8
9 }

```

```

1 package FinalEjemplos;
2
3 public class EjemploDeClase extends FinalClase {
4
5 }

```

The type EjemploDeClase cannot subclass the final class FinalClase
1 quick fix available:
Remove 'final' modifier of 'FinalClase'

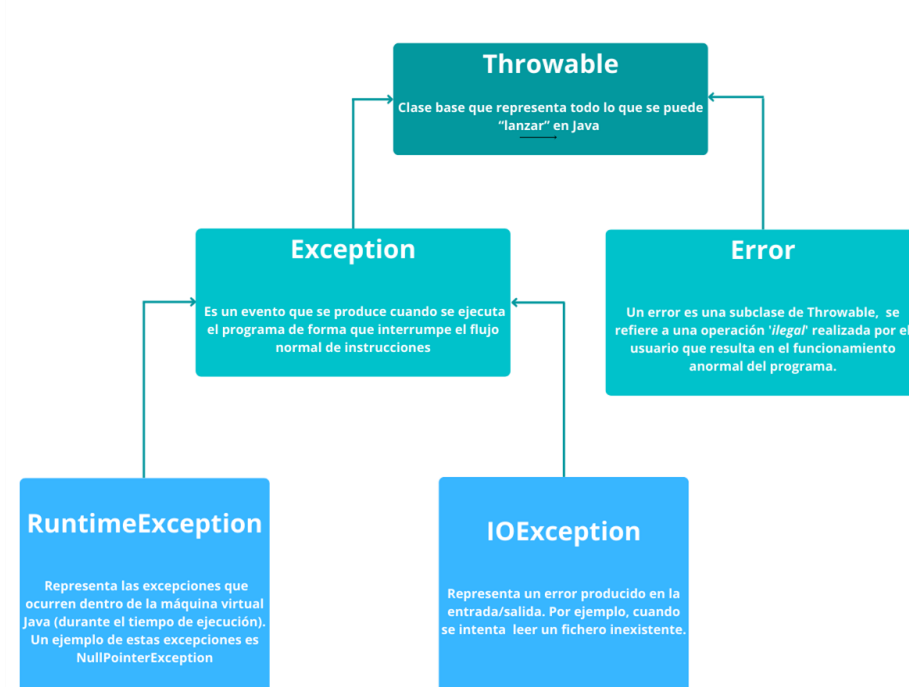
Explica los diferentes tipos de excepciones

Excepción: Las excepciones son situaciones anómalas que pueden ocurrir durante la ejecución de las aplicaciones, como, por ejemplo, acceder a una posición de un vector fuera rango.

```
int[] array = new int[10];  
System.out.println(array[100]);  
ArrayIndexOutOfBoundsException.
```

La definición de excepciones se hace a través de la **herencia**. Las excepciones se representan como clases y existe una jerarquía que representan errores en Java.

Tipos de excepciones en Java



Excepciones más comunes en Java

Excepción	Descripción
<i>FileNotFoundException</i>	Lanza una excepción cuando el fichero no se encuentra.
<i>ClassNotFoundException</i>	Lanza una excepción cuando no existe la clase.
<i>EOFException</i>	Lanza una excepción cuando llega al final del fichero.
<i>ArrayIndexOutOfBoundsException</i>	Lanza una excepción cuando se accede a una posición de un array que no exista.
<i>NumberFormatException</i>	Lanza una excepción cuando se procesa un numero pero este es un dato alfanumérico.
<i>NullPointerException</i>	Lanza una excepción cuando intentando acceder a un miembro de un objeto para el que todavía no hemos reservado memoria.
<i>IOException</i>	Generaliza muchas excepciones anteriores. La ventaja es que no necesitamos controlar cada una de las excepciones.
<i>Exception</i>	Es la clase padre de <i>IOException</i> y de otras clases. Tiene la misma ventaja que <i>IOException</i> .

Manejo de Excepciones (Try / Catch / Finally)

El **manejo de excepciones** en Java es un mecanismo mediante el cual se mantiene el flujo normal de la aplicación. Para hacer esto, se emplea un mecanismo para manejar errores o excepciones en tiempo de ejecución en un programa.

BLOQUE TRY

Todo el código que vaya dentro de esta sentencia será el código sobre el que se intentará capturar el error si se produce y una vez capturado hacer algo con él.

Lo ideal es que no ocurra un error, pero en caso de que ocurra un *bloque try* permite estar preparados para capturarlo y tratarlo.

BLOQUE CATCH

En este bloque se define el conjunto de instrucciones de tratamiento del problema capturado con el bloque try anterior. Es decir, cuando se produce un error o excepción en el código que se encuentra dentro de un bloque try, pasamos directamente a ejecutar el conjunto de sentencias que se tengan en el bloque catch.

BLOQUE FINALLY

El bloque *finally* es un bloque donde se puede definir un conjunto de instrucciones necesarias tanto si se produce error o excepción como si no y que por tanto se ejecuta siempre.

Ejemplo:

```
1 public class Prueba {
2
3     public static void main (String [] args)    {
4
5         try {
6             System.out.println("Intentamos ejecutar el bloque de instrucciones.");
7             System.out.println("Instrucción 1.");
8             int n = Integer.parseInt("M"); //Error forzado.
9             System.out.println("Instrucción 2.");
10            System.out.println("Instrucción 3, etc.");
11        }
12        catch (Exception e) {
13            System.out.println("Instrucciones a ejecutar cuando se produce un error");
14        }
15        finally {
16            System.out.println("Instrucciones a ejecutar finalmente");
17        }
18    }
19 }
20
```

Problems | Javadoc | Declaration | Console

<terminated> Prueba [Java Application] C:\Users\D\Desktop\Generation\ch17\java\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (2 dic. 2022 00:48:59 - 00:48:59) [pid: 16256]

Intentamos ejecutar el bloque de instrucciones:
Instrucción 1.
Instrucciones a ejecutar cuando se produce un error
Instrucciones a ejecutar finalmente tanto si se producen errores como si no.

Diagrama y explica una arquitectura web usando el patrón MVC

Modelo vista controlador (MVC)

Es un *estilo* de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes principales. Se trata de un modelo que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, y sobre multitud de lenguajes y plataformas de desarrollo.

El Modelo que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.

La Vista, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos interacción con éste.

El Controlador, que actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

Las funciones del **MODELO** son:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
- Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
- Lleva un registro de las vistas y controladores del sistema.
- Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero por lotes que actualiza los datos, un temporizador que desencadena una inserción, etc.).

Las funciones del **CONTROLADOR** son:

- Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).
- Contiene reglas de gestión de eventos, del tipo "Si Evento A, entonces Acción a". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener_tiempo_de_entrega (nueva_orden_de_venta)".

Las funciones de la **VISTA** son:

- Recibir datos del modelo y los muestra al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

De forma general el flujo de este proceso se lleva a cabo de la siguiente manera:



El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)



El controlador recibe (por parte de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, a través de un gestor de eventos (handler) o callback.



El controlador accede al modelo, actualizándolo, modificándolo de forma adecuada a la acción solicitada por el usuario.

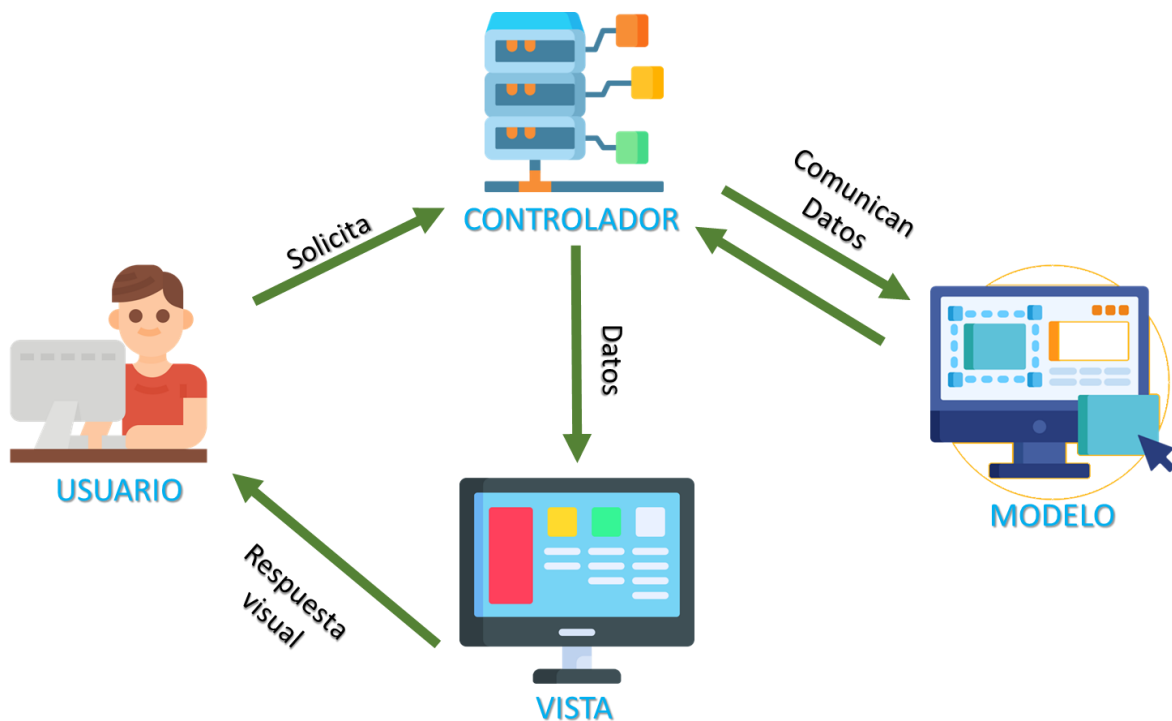


El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario.



La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en él.

El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón *Observador* para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista, aunque puede dar la orden a la vista para que se actualice. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.



Proceso síncrono y asíncrono (Diferencia y ejemplo)

Primeramente, se establecen los conceptos de sincronía y asincronía, así como sus principales diferencias de forma general, **sincronicidad** significa hacer algo al mismo tiempo, por su parte, **asíncrono**, hacer referencia a llevar a cabo acciones de forma no precisamente con un orden de tiempo.

Ejemplos generales:

Proceso síncrono: Sucede en tiempo real, es decir, el emisor y el receptor están conectados y están participando de un intercambio de manera simultánea. Por ejemplo: Dos personas hablan por chat.

Proceso asíncrono: La comunicación asincrónica no sucede en tiempo real porque el emisor y el receptor intercambian información de manera diferida. Por ejemplo: Una persona deja un comentario en un blog.

Programación síncrona y asíncrona.

En un modelo de programación **síncrono**, **las cosas suceden una a la vez**. Cuando se llama a una función que realiza una acción de larga duración, solo retorna cuando la acción ha terminado y puede retornar el resultado. Esto detiene al programa durante el tiempo que tome la acción.

Un modelo **asíncrono** permite que **ocurran varias cosas al mismo tiempo**. Cuando se comienza una acción, el programa continúa ejecutándose. Cuando la acción termina, el programa es informado y tiene acceso al resultado (por ejemplo, los datos leídos del disco).

En el siguiente diagrama, las líneas gruesas representan el tiempo que el programa pasa corriendo normalmente, y las líneas finas representan el tiempo pasado esperando la red. En el modelo síncrono, el tiempo empleado por la red es parte de la línea de tiempo para un hilo de control dado. En el modelo asincrónico, comenzar una acción de red conceptualmente causa una división en la línea del tiempo. El programa que inició la acción continúa ejecutándose, y la acción ocurre junto a el, notificando al programa cuando está termina.

synchronous, single thread of control



synchronous, two threads of control



asynchronous



Diagrama y codifica la inyección de dependencias

La inyección de dependencias es un patrón de diseño que tiene como objetivo **tomar la responsabilidad** de crear las instancias de las clases que otro objeto necesita y suministrárselo para que esta clase los pueda utilizar.

