



SEMANA 02

ACADEMIA JAVA

**ARANXA GUADALUPE MARTÍNEZ
OJEDA**

02 DE DICIEMBRE 2022

1. INYECCIÓN DE DEPENDENCIAS

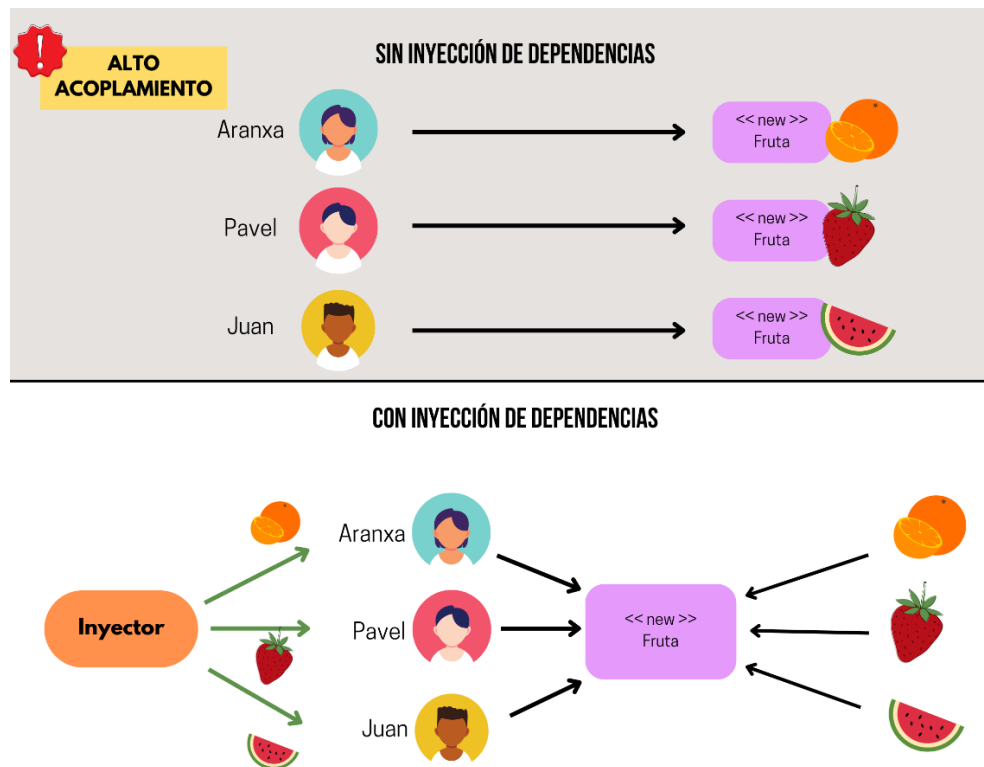


Ilustración 1. Diagrama Inyección de Dependencias

Si siguiendo el siguiente ejemplo, el problema que podemos encontrar es que cada Persona está fuertemente acoplada con la clase Fruta. Si una Persona deseará alguna otra fruta se tendría que crear una nueva clase para poder asignarle esta nueva fruta, esta forma de implementación es muy limitada y no es flexible por que lo que tendríamos que hacer uso de Inyección de Dependencias.

Con Inyección de Dependencias podemos apoyarnos de nuestro "Inyector" el cual nos ayuda a otorgarle a cada Persona una Fruta, en caso de que una desee una nueva Fruta podemos apoyarnos nuevamente del Inyector, ya que no dependemos directamente de la clase Fruta.

2. ARQUITECTURA WEB USANDO MVC

El patrón MVC (Model, View, Controller), es un patrón de arquitectura de software creado en los años 70 y uno de los patrones más conocido en la web. Hace referencia a como se estructura una aplicación.

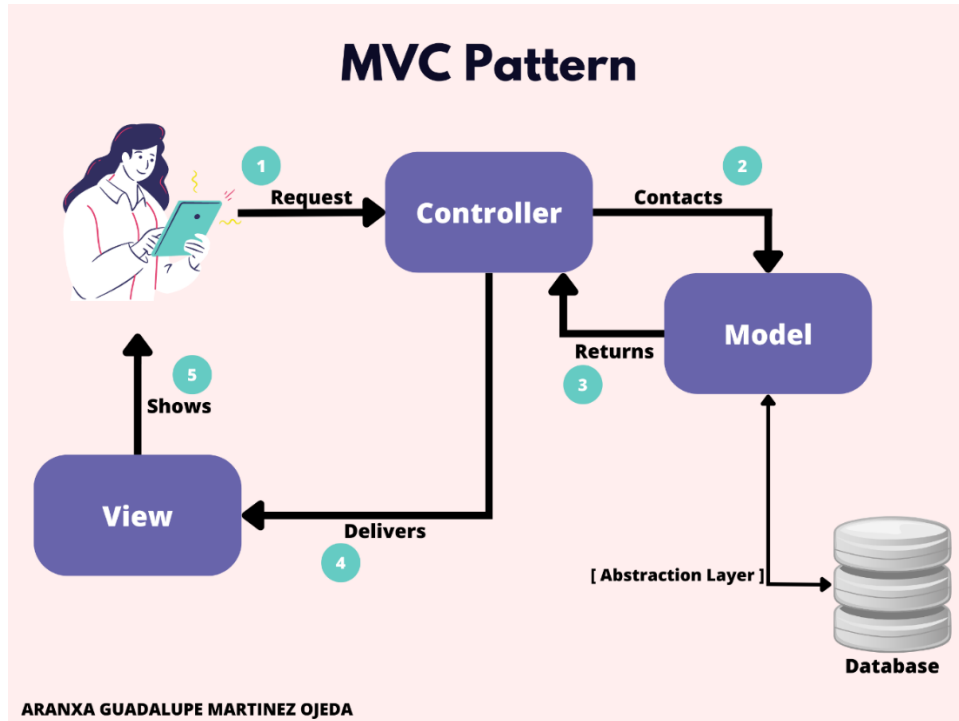


Ilustración 2. Diagrama patrón MVC

Model (Modelo): Representa la lógica de los datos, define estos datos para luego utilizarlos, manipularlos como un CRUD (create, read, update and delete). El modelo no debe de saber nada de la interfaz gráfica, es independiente, su única responsabilidad es la lógica y reglas del negocio.

View (Vista): Tiene todos los elementos visibles al usuario, lo que el usuario puede ver en la pantalla, es decir, la interfaz, que contiene elementos gráficos como botones, imágenes, formularios, etc. La vista se comunica con el controlador para poder acceder a los datos. La vista puede tener lógica de vista, pero no de negocios.

Controller (Controlador): Es el que actúa de intermediario entre el Modelo y la Vista, se encarga de la comunicación entre estos dos componentes, toma los datos pedidos en la vista/peticiones realizadas por el usuario y posteriormente interactúa con el modelo, para así decidir que datos va a mostrar. Aquí se encuentra la parte inteligente de la aplicación.

Recibe los métodos http → GET, POST, PUT, DELETE

Abstraction Layer (Capa de abstracción): Esta es una mejora adicional y es un mecanismo de abstracción con respecto al gesto de base de datos. Es una forma de independizar nuestra aplicación del gestor de base de datos, de modo que, si algún día necesitamos cambiar el gestor de base de datos, solo tendremos que reescribir el código de esa capa de abstracción.

3. TIPOS DE EXCEPCIONES

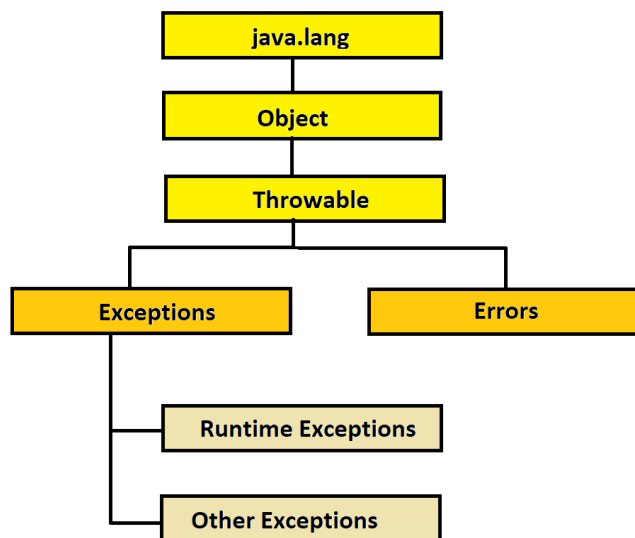


Ilustración 3.-Jerarquía Excepciones

Las excepciones son el medio que ofrecen algunos lenguajes de programación para tratar situaciones anómalas que pueden suceder cuando ejecutamos un programa, cuando estas excepciones suceden durante la ejecución de un programa, el programa aborta y la aplicación termina.

En Java estas excepciones tienen como clase padre a la clase *Throwable*, como vemos en el diagrama anterior. La clase *Throwable* a su vez es una subclase de la clase *Object*.

ERRORS: Subclase de *Throwable* que indica problemas graves que están sucediendo en el programa, no deben de ser capturados. Estos errores ocurren generalmente por condiciones anormales y no pueden ser resueltos por condiciones normales. A continuación, se muestran algunos ejemplos de errores:

Error	Descripción
AbstractMethodError	Cuando una aplicación Java intenta invocar un método abstracto.
StackOverflowError	Un desbordamiento de pila (stack overflow) de Java se produce cuando el tamaño de la memoria requerida por

	la pila del programa Java es mayor de lo que configuró el entorno de ejecución de Java
InternalError	Indicando la ocurrencia de un error interno inesperado en la JVM
VirtualMachineError	Indica que la JVM está estropeada o se ha quedado sin recursos, imprescindibles para seguir funcionando.
OutOfMemoryError	Una excepción OutOfMemoryError es consecuencia de haberse quedado sin espacio en el almacenamiento dinámico de Java o en cualquier área privada de MVS

EXCEPTIONS

Las excepciones son errores que no son críticos, por lo cual pueden ser tratados y continuar la ejecución de la aplicación. Como se muestra en el diagrama de jerarquía anterior hay una subclase especial de excepciones *RuntimeException*, este tipo de excepciones pueden producirse en cualquier parte de código, y por lo tanto no los métodos no tienen que capturar o lanzar explícitamente estas excepciones, es decir, son excepciones **Unchecked**, ejemplos:

Excepción	Ejemplo
ArithmeticException	Sucede si dividimos un numero entre cero <pre>Int a = 50/0;</pre>
NullPointerException	Si tenemos un valor nulo en alguna variable y deseamos realizar cualquier operación en la variable arroja esta exception <pre>String s = null; System.out.println(s.length());</pre>
NumberFormatException	Si el formato de cualquier variable o numero no coincide, puede resultar en esta excepción. Si tenemos una variable de tipo String y deseamos convertirla en int, causara este exception <pre>String s = "abc"; Int i = Integer.parseInt(s);</pre>

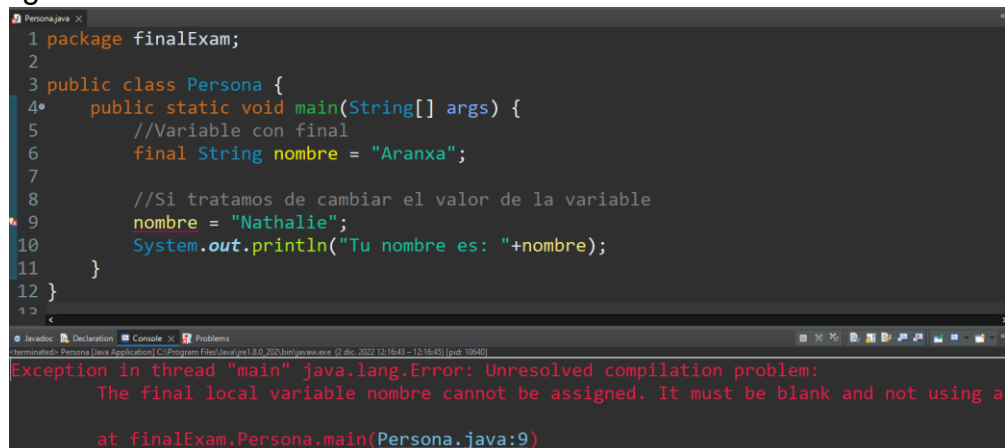
Excepciones Checked

Estas excepciones ocurren al momento de la compilación y deben ser implícitamente capturados y tratados en el código. Las Classes que son directamente heredadas de Throwable (a excepción de RuntimeException y Error) son excepciones checked. Estas se pueden manejar haciendo uso de **try catch** esto te permite procesar y lidiar con la excepción. Otra alternativa es hacer uso de **throws** para lanzar la excepción al método para ser manejada.

4. LOS 4 PROPÓSITOS DEL *FINAL*

a) Final en variables locales

No es necesario asignar un valor cuando una variable este declarada con *final*. La única regla es que se le debe de asignar un valor antes de ser usada. Cuando una variable esta definida con final, no esta permitido asignarle un nuevo valor.

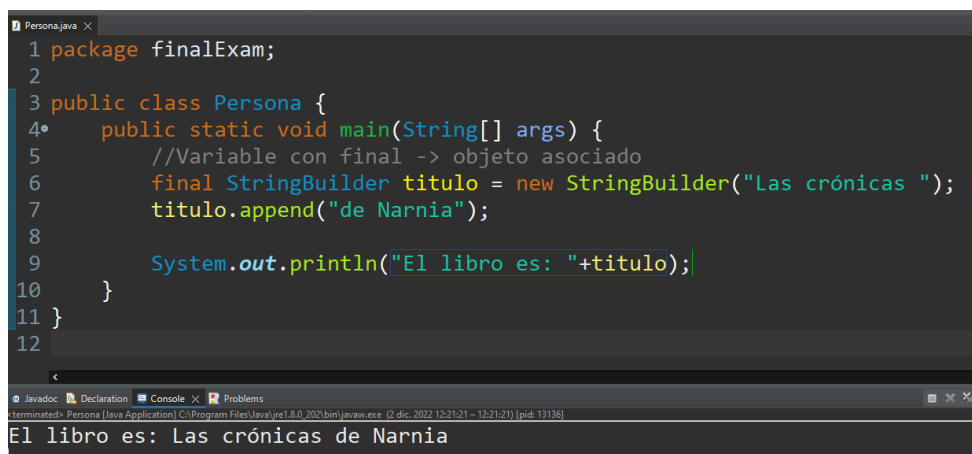


```
1 package finalExam;
2
3 public class Persona {
4     public static void main(String[] args) {
5         //Variable con final
6         final String nombre = "Aranxa";
7
8         //Si tratamos de cambiar el valor de la variable
9         nombre = "Nathalie";
10        System.out.println("Tu nombre es: "+nombre);
11    }
12 }
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The final local variable nombre cannot be assigned. It must be blank and not using a
at finalExam.Persona.main(Persona.java:9)

Ilustración 4. Ejemplo Variable Final

Si una variable de referencia esta marcada como final, esto no significa que el objeto a la que esta asociada no pueda ser modificado.



```
1 package finalExam;
2
3 public class Persona {
4     public static void main(String[] args) {
5         //Variable con final -> objeto asociado
6         final StringBuilder titulo = new StringBuilder("Las crónicas ");
7         titulo.append("de Narnia");
8
9         System.out.println("El libro es: "+titulo);
10    }
11 }
12
```

El libro es: Las crónicas de Narnia

Ilustración 5. Modificando Objeto asociado a variable final

b) Final en variables de instancia y estáticas

Si una variable de instancia esta declarada con *final*, entonces se le debe de asignar un valor cuando es declarada o cuando el objeto es inicializado. Al igual que una variable local, cuando se son marcadas con final su valor no puede ser cambiado.

```
1 package finalExam;
2
3 public class Persona {
4     final int edad; //Se debe de asignar un valor
5
6     public static void main(String[] args) {
7         final int manzanas; //No es necesario asignar valor al
8                             //declararlo pero si antes de usarlo
9     }
10 }
11 |
```

Ilustración 6. Diferencia en variable final de instancia y variable final local

c) Final en Métodos

Los métodos que contengan *final* no pueden ser sobrescritos (Override) por una subclase

```
1 package finalExam;
2
3 public class Operacion {
4     int x;
5     int y;
6
7     public Operacion(int x, int y) {
8         this.x = x;
9         this.y = y;
10    }
11
12    public final int resultado() {
13        return x+y;
14    }
15 }
```

Ilustración 7. Clase Operacion con método Resultado final

Creamos otra clase “Suma” que hereda a la clase “Operación”, al tratar de sobrescribir el método resultado, nos manda un error ya que este es un método final

```

1 package finalExam;
2
3 public class Suma extends Operacion{
4
5     public Suma(int x, int y) {
6         super(x, y);
7     }
8
9     @Override //No se puede sobrescribir
10    public final int resultado() {
11        return x+y;
12    }
13 }

```

Ilustración 8. Clase hija tratando de sobrescribir método resultado

d) Final en Clases

Cuando una clase tiene final, esta no se puede heredar (extended)

```

1 package finalExam;
2
3 final public class Operacion {
4     int x;
5     int y;
6
7     public Operacion(int x, int y) {
8         this.x = x;
9         this.y = y;
10    }
11
12    public int resultado() {
13        return x+y;
14    }

```

Ilustración 9. Clase Final Operacion

```

1 package finalExam;
2
3 //No puede heredar de Operacion
4 public class Suma extends Operacion{
5
6     public Suma(int x, int y) {
7         super(x, y);
8     }
9
10    @Override
11    public int resultado() {
12        return x+y;
13    }
14 }

```

Ilustración 10. Suma tratando de heredar Operacion

5. DIFERENCIA ENTRE LOS PROCESOS SÍNCRONOS Y ASÍNCRONOS

En un proceso síncrono, en la ejecución de un programa cada tarea se ejecuta una vez que ha terminado la tarea anterior, haciéndolo de manera secuencial.

Mientras que, en un proceso síncrono, una tarea no tiene que esperar a que se termine de ejecutar otra tarea para continuar.

Síncrono → Un gran ejemplo de un proceso síncrono en nuestro día a día, es en la fila de compra para el super. Una vez tienes lista todas tus compras de la semana, tiene que esperar a que la cajera acabe de atender a las personas delante de ti, para que así eventualmente sea tu turno y la cajera pueda atenderte y realizar tus compras de manera exitosa

Asíncrono → Otro gran ejemplo para un proceso asíncrono es cuando vas a un restaurante con toda tu familia, cada uno ordena lo que más les gusta y el mesero manda la orden al Chef, aquí cada platillo es entregado conforme el chef los termine, es decir, los platillos no salen por el orden en el que fueron anotados por el mesero, si no que van saliendo conforme el tiempo y la complejidad de cada platillo.

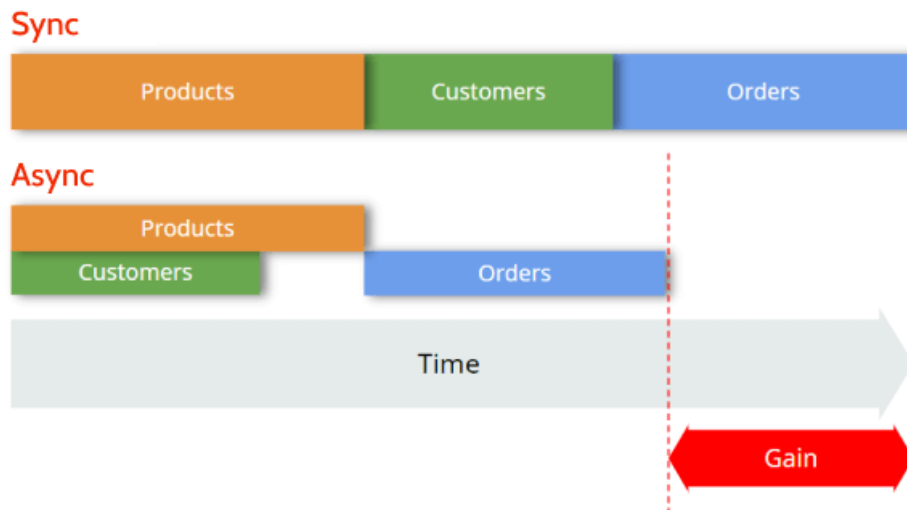


Ilustración 11. Gráfica que muestra el tiempo entre un proceso síncrono y asíncrono