

# Inyección de dependencias

<b>Inyección de dependencias</b>	<b>1</b>
¿Qué aprenderás?	2
Introducción	2
¿Qué es inyección de dependencias?	3
Ejemplo de inyección de dependencias	3
Ahora bien, ¿de qué sirve esto?	8



**¡Comencemos!**

## ¿Qué aprenderás?

- Conocer el concepto de inyección de dependencias.
- Aplicar inyección de dependencias.

## Introducción

En este capítulo conoceremos el concepto de inyección de dependencias, esto nos facilitará la vida como programadores al reutilizar código y dividir las tareas que realiza un programa, veremos la importancia del uso de inyección de dependencias en una aplicación, ejemplos de esto y cómo podemos utilizarla.



¡Comencemos!

## ¿Qué es inyección de dependencias?

La inyección de dependencias es un patrón de diseño orientado a objetos que permite dividir y repartir las responsabilidades de un programa.

Repartir las responsabilidades de un programa, se refiere a que un programa posee diferentes funciones, y si bien están para el uso del programa, no se engloban dentro de una sola gran función. Éstas se ordenan de tal manera que favorezca el entendimiento del programa y la reutilización del código.

## Ejemplo de inyección de dependencias

Imaginemos que tenemos un sistema de pedidos de comida, pero necesitamos registrar los pedidos, para luego cobrarlos y enviarlos, en esta representación sólo imprimimos texto que representará las tareas que realiza el sistema.

El archivo de *ServicioComidaDomicilio* tiene tres tareas, la primera es registrar el pedido y la segunda es cobrar el pedido para luego enviarlo.

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_uno;
public class ServicioComidaDomicilio {
    public void enviar() {
        System.out.println("Registrar pedido");
    }
}
```

```
        System.out.println("Cobrar Pedido");  
        System.out.println("Enviar pedido");  
    }  
}
```

Luego, dentro del método *Main*, realizamos la llamada al objeto *ServicioComidaDomicilio*:

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_uno;  
public class Main {  
    public static void main(String[] args) {  
        ServicioComidaDomicilio servicio = new ServicioComidaDomicilio();  
        servicio.enviar();  
    }  
}
```

Nuestro resultado se verá de la siguiente manera:

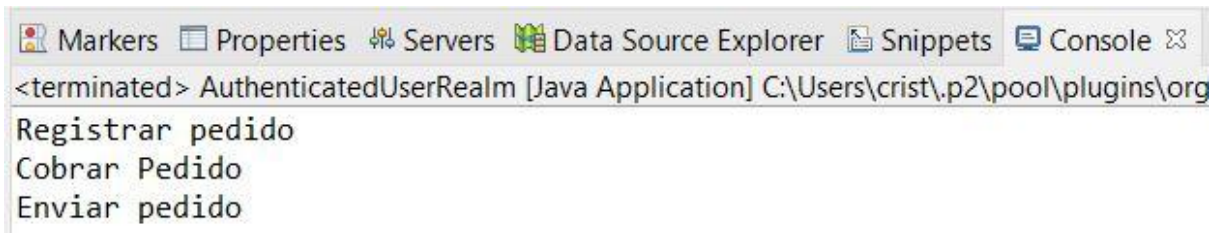


Imagen 1. Resultado.  
Fuente: Desafío Latam

Ahora bien, separando los servicios, se delegan responsabilidades, vale decir, cada servicio se especializa en una función (concepto: divide y vencerás) y haciendo que cada uno se comporte de forma independiente.

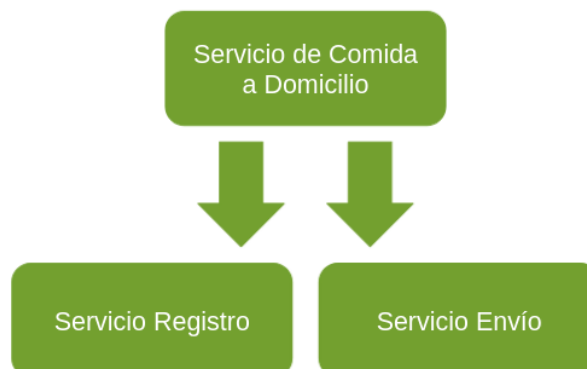


Imagen 2. Separando los servicios.  
Fuente: Desafío Latam

Creamos el servicio para registrar el pedido (*ServicioRegistroPedido*):

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_dos;
public class ServicioRegistroPedido {
    public void registrarPedido() {
        System.out.println("Registrar Pedido");
    }
}
```

Creamos el servicio para cobrar el pedido (*ServicioCobroPedido*):

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_dos;
public class ServicioCobroPedido {
    public void cobrarPedido() {
        System.out.println("Cobrar Pedido");
    }
}
```

Creamos el servicio para enviar el pedido (*ServicioEnvioPedido*):

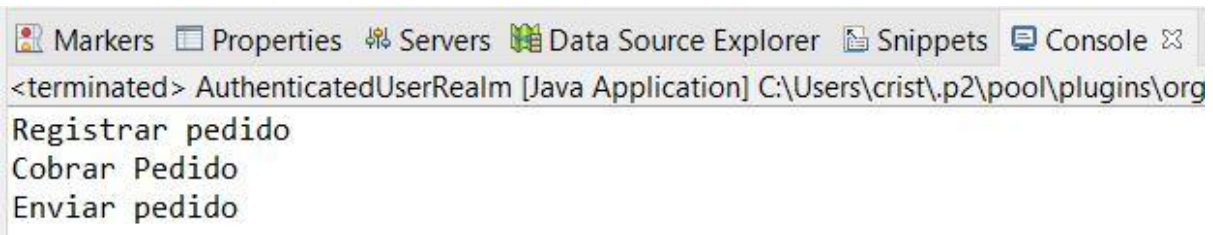
```
package com.desafiolatam.inyeccion_dependencias_ejemplo_dos;
public class ServicioEnvioPedido {
    public void enviarPedido() {
        System.out.println("Enviar Pedido");
    }
}
```

Se definen los objetos en el constructor de la clase *ServicioComidaDomicilio*, obteniendo así que el servicio de comida a domicilio (*ServicioComidaDomicilio*) dependa de otros tres servicios:

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_dos;
public class ServicioComidaDomicilio {
    ServicioRegistroPedido servicio_registro;
    ServicioCobroPedido servicio_cobro;
    ServicioEnvioPedido servicio_envio;
    public ServicioComidaDomicilio() {
        this.servicio_registro = new ServicioRegistroPedido();
        this.servicio_cobro = new ServicioCobroPedido();
        this.servicio_envio = new ServicioEnvioPedido();
    }
    public void enviar() {
        servicio_registro.registrarPedido();
    }
}
```

```
servicio_cobro.cobrarPedido();  
servicio_envio.enviarPedido();  
}  
}
```

Obteniendo como resultado lo mismo, ya que solo se externalizan servicios y el método Main no es modificado.



```
<terminated> AuthenticatedUserRealm [Java Application] C:\Users\crist\p2\pool\plugins\org  
Registrar pedido  
Cobrar Pedido  
Enviar pedido
```

Imagen 3. El mismo resultado anterior.  
Fuente: Desafío Latam

Con lo anterior, conseguimos que nuestro servicio de comida a domicilio dependa de dos servicios que funcionan de forma paralela e independientes, pero, se puede realizar esta misma operación inyectando las dependencias al servicio de comida a domicilio, así él no las define en el constructor del servicio.

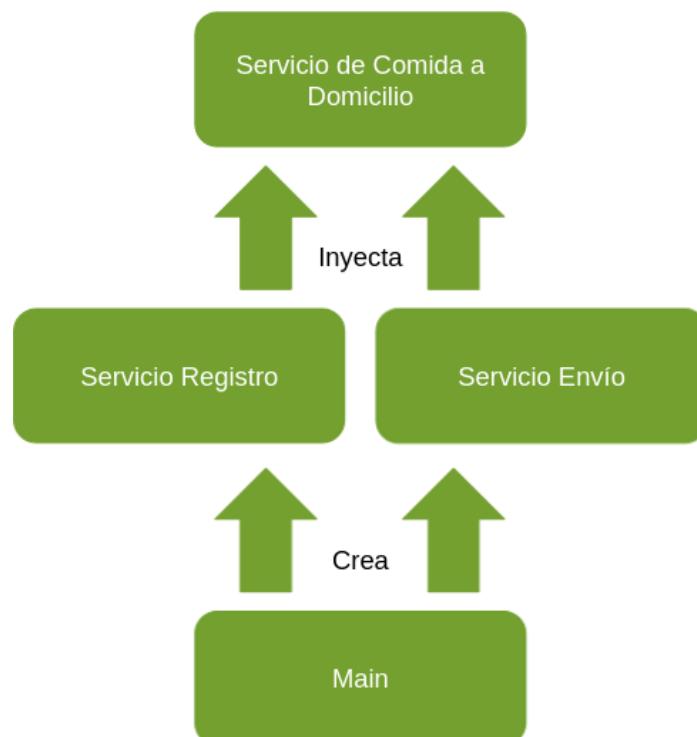


Imagen 4. Modificación diagrama Servicio de Comida a Domicilio.  
Fuente: Desafío Latam

Modificando el servicio de comida a domicilio (*ServicioComidaDomicilio*), obtenemos que el servicio si depende de los otros tres servicios, pero él no los genera, sólo los utiliza.

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_tres;
public class ServicioComidaDomicilio {
    ServicioRegistroPedido servicio_registro;
    ServicioCobroPedido servicio_cobro;
    ServicioEnvioPedido servicio_envio;
    public ServicioComidaDomicilio(ServicioRegistroPedido
servicio_registro, ServicioCobroPedido servicio_cobro,
ServicioEnvioPedido servicio_envio) {
        this.servicio_registro = servicio_registro;
        this.servicio_cobro = servicio_cobro;
        this.servicio_envio = servicio_envio;
    }
    public void enviar() {
        servicio_registro.registrarPedido();
        servicio_cobro.cobrarPedido();
        servicio_envio.enviarPedido();
    }
}
```

Los servicios de registro (*ServicioRegistroPedido*), el de envío (*ServicioEnvioPedido*) y el de cobro (*ServicioCobroPedido*) son definidos en el método *Main* y son utilizados por el servicio de comida a domicilio (*ServicioComidaDomicilio*).

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_tres;
public class Main {
    public static void main(String[] args) {
        ServicioComidaDomicilio servicio = new ServicioComidaDomicilio(
new ServicioRegistroPedido(), new ServicioCobroPedido(), new
ServicioEnvioPedido());
        servicio.enviar();
    }
}
```

Entonces, el servicio principal ya no se encarga de definir sus dependencias, sino que lo hace el método *Main*.

## Ahora bien, ¿de qué sirve esto?

Esto nos permite la extensibilidad de los servicios. La extensibilidad, es la capacidad de un programa para soportar nuevas funcionalidades o elementos sin alterar la funcionalidad del servicio (o alterar mínimamente).

Continuando con el ejemplo, supongamos que ahora se requiere verificar el pedido antes de enviarlo, como la verificación no será un servicio extra, extendemos nuestro servicio de envío (*ServicioEnvioPedido*) sin modificar la funcionalidad de este, creando así un nuevo servicio (*ServicioEnvioVerificoPedido*) que extiende los atributos y métodos del servicio anterior.

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_tres;
public class ServicioEnvioVerificoPedido extends ServicioEnvioPedido {
    public void enviarPedido() {
        System.out.println("Verifico Pedido");
        super.enviarPedido();
    }
}
```

Modificando solo la definición en el método *Main*:

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_tres;
public class Main {
    public static void main(String[] args) {
        ServicioComidaDomicilio servicio = new ServicioComidaDomicilio(
new ServicioRegistroPedido(), new ServicioCobroPedido(), new
ServicioEnvioVerificoPedido());
        servicio.enviar();
    }
}
```

Obteniendo como resultado, finalmente:

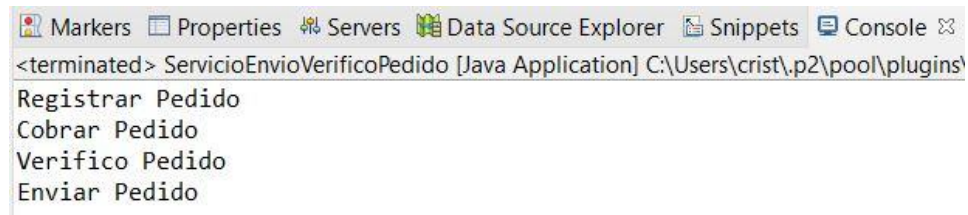


Imagen 5. Resultado final.  
Fuente: Desafío Latam

Esto último, nos permitió modificar el comportamiento de la aplicación gracias al concepto de inyección de dependencias. Entonces, la inyección de dependencias nos permite inyectar clases y funcionalidades de manera transversal a medida que la aplicación crece.

Otra manera para implementar el concepto inyección de dependencias, es a través de interfaces. Una interfaz es un objeto abstracto, vale decir, una interfaz es un conjunto de atributos, métodos y funciones que especifica lo que se debe hacer y no cómo se debe hacer, las clases que implementen la interfaz será la que describa la lógica del método. Vamos al ejemplo:

Creamos la interfaz que posee dos métodos, *enviarPedido()* y *verificarPedido()*:

```
package com.inyeccion_dependencias_ejemplo_cinco;
public interface IServicioEnvioPedido {
    public void enviarPedido();
    public void verificarPedido();
}
```

Implementando la interfaz, generando la clase *ServicioEnvioPedido*:

```
package com.inyeccion_dependencias_ejemplo_cinco;
public class ServicioEnvioPedido implements IServicioEnvioPedido{
    public void enviarPedido() {
        System.out.println("Enviar Pedido");
    }
    public void verificarPedido() {
        System.out.println("Verifico pedido");
    }
}
```



En el *ServicioComidaDomicilio*, asignamos la interfaz como servicio de envío (*IServicioEnvioPedido*), haciendo que el servicio de comida a domicilio reconozca métodos y atributos provenientes desde la interfaz. O sea, el servicio de comida a domicilio sabe lo que va a hacer y no cómo lo va a hacer.

```
package com.desafiolatam.inyeccion_dependencias_ejemplo_cinco;

public class ServicioComidaDomicilio {
    ServicioRegistroPedido servicio_registro;
    ServicioCobroPedido servicio_cobro;
    IServicioEnvioPedido servicio_envio;

    public ServicioComidaDomicilio(ServicioRegistroPedido
servicio_registro, ServicioCobroPedido servicio_cobro,
                                IServicioEnvioPedido servicio_envio) {
        this.servicio_registro = servicio_registro;
        this.servicio_cobro = servicio_cobro;
        this.servicio_envio = servicio_envio;
    }

    public void enviar() {
        servicio_registro.registrarPedido();
        servicio_cobro.cobrarPedido();
        servicio_envio.verificarPedido();
        servicio_envio.enviarPedido();
    }
}
```

Una vez más, se corrige el método *Main* para que haga defina los objetos dependientes del servicio:

```
package com.inyeccion_dependencias_ejemplo_cinco;
public class Main {
    public static void main(String[] args) {
        ServicioComidaDomicilio servicio= new ServicioComidaDomicilio(
            new ServicioRegistroPedido(),
            new ServicioCobroPedido(),
            new ServicioEnvioPedido()
        );
        servicio.enviar();
    }
}
```

Y se obtiene como resultado:

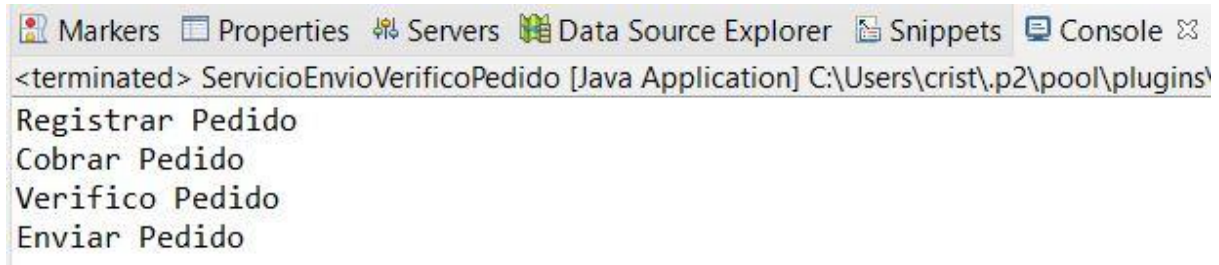


Imagen 6. Resultado mostrado en pantalla.

Fuente: Desafío Latam

Entonces, las interfaces nos permiten modificar métodos y atributos de las clases de manera independiente sin alterar las funcionalidades de los objetos, obteniendo así los mismos resultados.

Ya que sabemos que es inyección de dependencias y como se utiliza en un proyecto. Spring y muchos otros frameworks utilizan el patrón de diseño de inyección de dependencias.