



# Inyección de dependencias

La inyección de dependencias (DI) es un patrón de diseño que se utiliza para implementar la inversión de control (IoC), lo que significa que el control sobre cómo se obtienen las dependencias de una clase es invertido y delegado a otro componente.

Se entiende por dependencia cualquier objeto que una clase necesita para realizar su función. Por ejemplo, en caso de una clase `carro`, se necesita una instancia de `motor` para funcionar, entonces se puede decir que `motor` es una dependencia de `carro`. Entonces la inyección de dependencias ocurre cuando en lugar de que la clase `carro` cree su propia instancia de `motor`, se le proporcione una instancia de `motor` desde el exterior de la clase `carro`.

## Beneficios

- Desacoplamiento → Las clases no necesitan saber cómo crear sus dependencias, lo que facilita el cambio de implementación y mejora la mantenibilidad del código.
- Facilidad de pruebas → Al inyectar las dependencias, es más sencillo hacer pruebas unitarias ya que se pueden inyectar objetos simulados (mocks) en lugar de las dependencias reales.

## Tipos de inyección

1. Inyección a través de constructor → Las dependencias se pasan a la clase a través de su constructor.
2. Inyección a través de un setter → Las dependencias se establecen mediante métodos setter.
3. Inyección de campo → Las dependencias se inyectan a través de un atributo.

## Ejemplo

A continuación se propone como ejemplo un sistema de prestamos de libros, en el sistema se tiene un registro de miembros del club de lectura con sus datos y los datos del libro que se les presta semanalmente.

### Sin inyección de dependencias

El código que se muestra debajo corresponde a la clase para instanciar los libros, cada libro tiene como atributos el nombre, su ISBN y una matricula propia del club.

```
package V0;
```

```
public class Libro {  
    private String titulo;  
    private String isbn;  
    private String matricula;  
  
    Libro(String titulo, String isbn, String matricula){  
        this.titulo = titulo;  
        this.isbn = isbn;  
        this.matricula = matricula;  
    }  
  
    public String getTitulo(){
```

```

        return this.titulo;
    }
}

```

Por otra parte se tiene el código correspondiente a los miembros, cada miembro tiene asociado su nombre y el libro que se le presta semanalmente. Sin embargo, como se nota en la definición de los atributos, existe un alto acoplamiento pues el miembro toma la responsabilidad de crear la instancia de la dependencia libro pedido.

```

package V0;

public class Miembro {
    private String nombre;
    //Alto acoplamiento
    private Libro libroPedido = new Libro("El principito", "9783140464079", "9735ALSK");

    Miembro(String nombre){
        this.nombre = nombre;
    }

    public void leerLibro(){
        System.out.println("Hola, soy " + this.nombre +
            " y voy a leer " + this.libroPedido.getTitulo());
    }
}

```

Como resultado del alto acoplamiento a todos los miembros se les asigna el mismo libro, lo cual representa una gran rigidez y una abstracción muy pobre, pues el funcionamiento del programa no se apegas al comportamiento real esperado.

```

package V0;

public class Principal {
    public static void main(String[] args){
        Miembro m1 = new Miembro("Antonio");//Instanciando a un nuevo miembro
        m1.leerLibro();//Antonio comienza a leer el libro
        Miembro m2 = new Miembro("Alberto");//Instanciando a un nuevo miembro
        m2.leerLibro();//Antonio comienza a leer el libro
    }
}

```

```

Hola, soy Antonio y voy a leer El principito
Hola, soy Alberto y voy a leer El principito

Process finished with exit code 0

```

## Inyección por campo

Los cambios necesarios para realizar la inyección por variable o campo se llevan a cabo en la clase Miembro. En la definición del atributo Libro se reemplaza el constructor del objeto únicamente por la declaración de la variable de referencia `libroPedido`.

```

package campo;

public class Miembro {
    private String nombre;

    //Inyeccion por campo/variable
    Libro libroPedido;

    Miembro(String nombre){
        this.nombre = nombre;
    }

    public void leerLibro(){
        System.out.println("Hola, soy " + this.nombre +
            " y voy a leer " + this.libroPedido.getTitulo());
    }
}

```

Con este cambio ahora la inicialización del libro queda pendiente. Para poder asignar un libro se crea la clase Inyector cuya tarea es asignar un libro al miembro que se le indique. La asignación se realiza a través de la escritura directa sobre el atributo del miembro.

```

package campo;

public class Inyector {
    static void asignarLibro(Miembro miembro, Libro libro){
        miembro.libroPedido = libro; //Asignación de libro
    }
}

```

Ahora en el programa principal se realiza la instancia de miembros y libros para posteriormente asignarlos entre si y lograr un menor acoplamiento. Ahora se le puede asignar un libro diferente a cada miembro.

```

package campo;

public class Principal {
    public static void main(String[] args){
        //Instanciando miembros
        Miembro m1 = new Miembro("Antonio");
        Miembro m2 = new Miembro("Alberto");
        //Instanciando libros
        Libro l1 = new Libro("El principito", "978-607-99498-0-8", "ASXD9865");
        Libro l2 = new Libro("1984", "978-6073116336", "PSKD2846");
        //Asignando libro a cada miembro
        Inyector.asignarLibro(m1,l2);
        Inyector.asignarLibro(m2,l1);

        m1.leerLibro();
        m2.leerLibro();
    }
}

```

```
}  
}
```

```
Hola, soy Antonio y voy a leer 1984  
Hola, soy Alberto y voy a leer El principito  
  
Process finished with exit code 0
```

## Inyección por setter

Esta vez las modificaciones se realizan sobre el inyector y la clase miembro. En la clase miembro se realiza el encapsulamiento del atributo libro pedido, de manera que ahora el inyector se ve obligado a utilizar el setter de dicho atributo.

```
package setter;  
  
public class Miembro {  
    private String nombre;  
  
    //Inyeccion por setter  
    private Libro libroPedido;  
  
    Miembro(String nombre){  
        this.nombre = nombre;  
    }  
  
    public void leerLibro(){  
        System.out.println("Hola, soy " + this.nombre +  
            " y voy a leer " + this.libroPedido.getTitulo());  
    }  
  
    public void setLibroPedido(Libro libroPedido){  
        this.libroPedido = libroPedido;  
    }  
}  
  
public class Inyector {  
    static void asignarLibro(Miembro miembro, Libro libro){  
        miembro.setLibroPedido(libro);  
    }  
}
```

## Inyección por constructor

En la inyección por constructor se mantiene la visibilidad de la clase miembro, sin embargo, se agrega la posibilidad de asignar el atributo libro prestado desde que se inicializan los miembros.

```
package constructor;  
  
public class Miembro {  
    private String nombre;
```

```
//Inyeccion por constructor
private Libro libroPedido;

Miembro(String nombre, Libro libroPedido){
    this.nombre = nombre;
    this.libroPedido = libroPedido; // Ahora el constructor espera un libro
}

public void leerLibro(){
    System.out.println("Hola, soy " + this.nombre +
        " y voy a leer " + this.libroPedido.getTitulo());
}

public void setLibroPedido(Libro libroPedido){
    this.libroPedido = libroPedido;
}
}
```

El método asignar libro del inyector ahora es el que tiene la responsabilidad de dar de alta al miembro y asignarla un libro.

```
package constructor;

public class Inyector {
    static Miembro asignarLibro(String nombre, Libro libro){
        //Instancia del miembro y asignación de libro
        return new Miembro("nombre",libro);
    }
}
```

En el programa principal ya no es necesario invocar al constructor de miembros, ahora instanciamos nuestros libros disponibles para posteriormente asignarlos a través del inyector y a la vez instanciar a nuestros lectores.

```
package constructor;


public class Principal {
    public static void main(String[] args){
        //Instanciando libros
        Libro l1 = new Libro("El principito", "978-607-99498-0-8","ASXD9865");
        Libro l2 = new Libro("1984", "978-6073116336","PSKD2846");
        //Instanciando y asignando libro a cada miembro
        Miembro m1 = Inyector.asignarLibro("Antonio",l2);
        Miembro m2 = Inyector.asignarLibro("Alberto",l1);

        m1.leerLibro();
        m2.leerLibro();
    }
}
```

## Referencias

## Dependency Injection :: Spring Framework

Dependency injection (DI) is a process whereby objects define their dependencies (that is, the other objects with which they work) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after

 <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>

### 36. Principio Inyección de Dependencias. DI (Dependency Injection)

Cualquier aplicación no trivial esta compuesta de dos o más clases que colaboran para lograr un objetivo de la lógica del negocio. Tradicionalmente cada objeto es responsable de obtener sus propias referencias de los objetos que colaboran con él. Esto puede llevarnos a acoplamientos

 [https://www.youtube.com/watch?v=\\_93CJUHMmFs](https://www.youtube.com/watch?v=_93CJUHMmFs)

