IoC (Inversion of control)

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

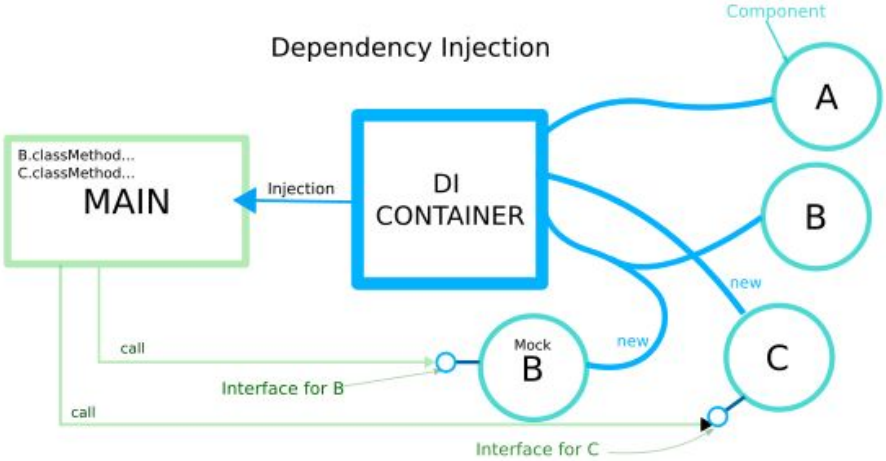
Contenedor de Inversión de Control

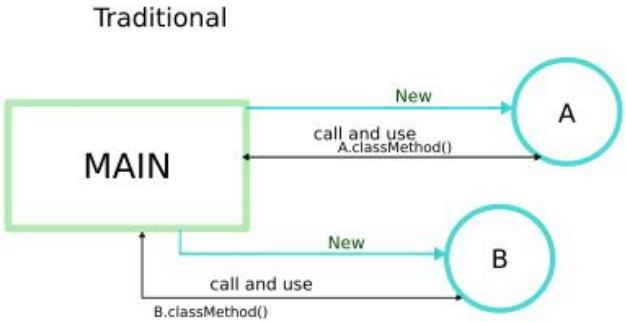
Spring se basa en el principio de **Inversión de Control (IoC)** o Patrón Hollywood («No nos llames, nosotros le llamaremos») consiste en:
Un Contenedor que maneja objetos por tí.

El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, **el contenedor hace los "new" de las clases java** para que no los realicemos nosotros, de ahí proviene el nombre de qué se invierte el control.

Debido a la naturaleza del IoC, el contenedor define el ciclo de vida de los objetos y resuelve las dependencias entre los objetos (**inyección de dependencia**).

Contenedor de Inversión de Control





Contenedor de Inversión de Control

El contenedor de Inversión de control es responsable de crear instancias, configurar y ensamblar los objetos(beans).

El contenedor obtiene sus instrucciones sobre qué objetos debe instanciar, configurar e inyectar leyendo los metadatos de configuración. Los metadatos de configuración se representan a través de las **anotaciones**.

Ventajas

- Ampliar/Modificar la funcionalidad del sistema sin necesidad de modificar las clases.
- Proporciona modularidad (aparte de la que nos proporciona nativamente el lenguaje).
- Evita la dependencia entre clases.

¿Cómo inyectar las dependencias?

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

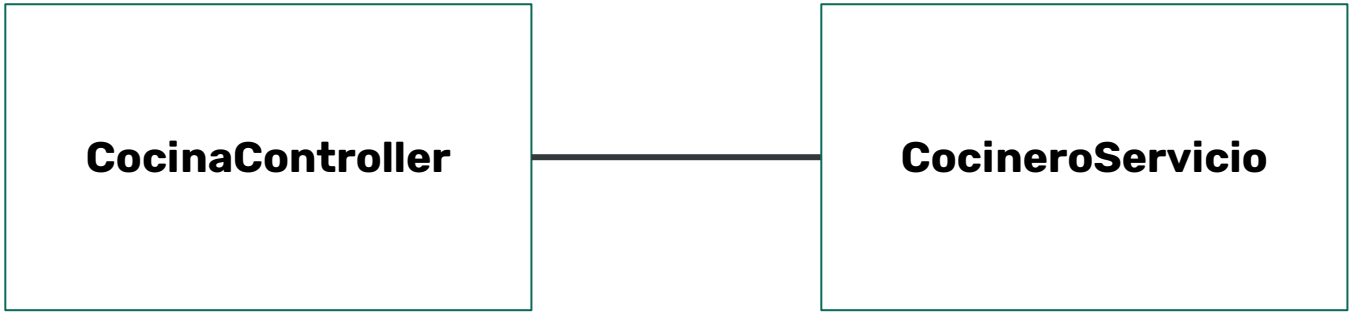
El rol de las interfaces en la inyección de dependencia

Antes de ver las diferentes formas de realizar una inyección de dependencia es esencial que conozcamos una buena práctica que debemos tener presente sin importar cual de las tres formas apliquemos.

La inyección de dependencias puede realizarse referenciando directamente las clases de dichas dependencias.

Para nuestro ejemplo de cocinero esto sería tener un servicio llamado CocineroServicio donde tengamos toda la lógica necesaria para que el cocinero pueda preparar los diferentes platos, y cada vez que necesitemos implementar la inyeccion de dependencia deberíamos referenciar/llamar dicha clase (CocineroServicio).

En desarrollo de software este problema se lo llama acoplamiento, es decir, la clase CocinaController está fuertemente acoplada a la clase CocineroServicio y no es una buena práctica porque, si en un futuro cambiamos de cocinero o agregamos uno nuevo, el mantenimiento de la clase Cocina se vuelve complicado.



Para mejorar nuestro diseño podemos crear una Interfaz ‘’CocineroService’’ y en el caso de que agregamos diferentes implementaciones solo será necesario agregar dicha implementación sin modificar CocinaController.

¿Cuáles son los tres métodos?

Ahora avancemos para conocer los tres métodos posibles para inyectar las dependencias.

- Por constructor
- Por método setters
- Directo en propiedad

Por constructor

Este método es el más recomendado, pues te daría la visibilidad necesaria de cuantas inyecciones tiene tu clase y es más fácil de hacer debug.

```
@Controller
public class CocinaController {
    private CocineroService unCocineroService;

    /**
     * Constructor de la clase CocinaController
     */
    @Autowired // Anotacion opcional desde version 4.3
    public CocinaController(CocineroService cocineroService) {
        this.unCocineroService = cocineroService;
    }

    public String prepararPlato(String ingredientes,String menu) {
        this.unCocineroService.setIngredientes(ingredientes);
        this.unCocineroService.setMenu(menu);
        String platoListo = this.unCocineroService.getPlatoListo();
        return platoListo;
    }
}
```

Creamos la propiedad con la dependencia de CocineroService que luego la asignaremos por medio del constructor de la clase CocinaController

Por método setters

Las dependencias de la clase se inyectarán a través de métodos setter.

```
@Controller
public class CocinaController {
    private CocineroService unCocineroService;

    @Autowired
    public void setCocineroService (CocineroService cocineroService) {
        this.unCocineroService = cocineroService;
    }

    public String prepararPlato(String ingredientes,String menu) {
        this.unCocineroService.setIngredientes(ingredientes);
        this.unCocineroService.setMenu(menu);
        String platoListo = this.unCocineroService.getPlatoListo();
        return platoListo;
    }
}
```

Creamos la propiedad con la dependencia de CocineroService que luego la asignaremos por medio del método set

Directo en propiedad

Muy rápido y efectivo con clases muy pequeñas.

```
@Controller
public class CocinaController {

    @Autowired
    private CocineroService uncocineroService;

   public String prepararPlato (String ingredientes, String
menu) {
    }

       unCocineroService.setIngredientes(ingredientes);
       unCocineroService.setMenu(menu);
       String platoListo = unCocineroService.getPlatoListo();
        return platoListo;
    }
}
```

Directamente declaramos la propiedad con la dependencia de CocineroService con la @Autowired

DigitalHouse>