



salesianos

TRIANA

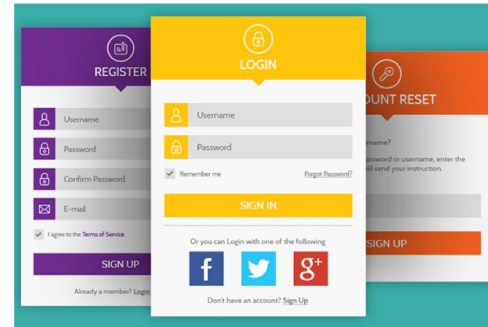
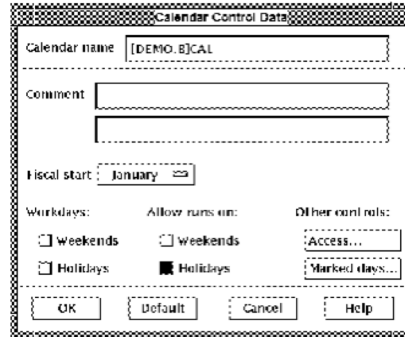
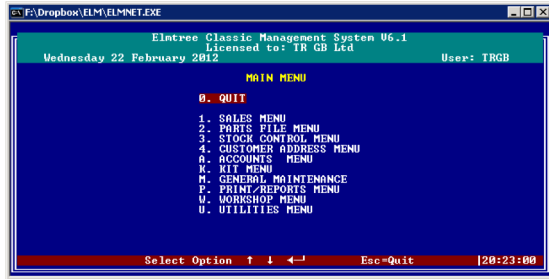


REPASO DE SPRING CORE

INVERSIÓN DE CONTROL E INYECCIÓN DE DEPENDENCIAS

INVERSIÓN DE CONTROL (IoC)

- ▶ Principio de diseño (o patrón)
- ▶ El objetivo es conseguir *desacoplar* objetos.



Principio de Hollywood

No nos llames.

Nosotros te llamaremos a tí.



INVERSIÓN DE CONTROL (IoC)

- ▶ Martin Fowler
- ▶ *Dejar que sea otro el que controle el flujo del programa* (por ejemplo, un framework)

```
#ruby
puts 'What is your name?'
name = gets
→ process_name(name)
puts 'What is your quest?'
quest = gets
→ process_quest(quest)
```

Ejemplos propuestos por Martin Fowler

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)} ←
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)} ←
Tk.mainloop()
```

Ralph Johnson and Brian Foote

Journal of Object-Oriented Programming

Junio/Julio 1988

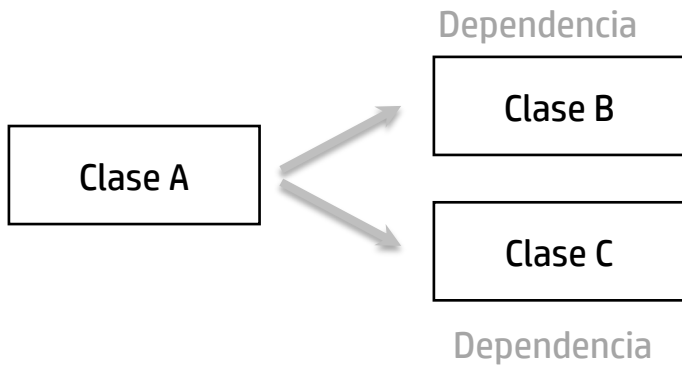
*Una característica importante de un **framework** es que los **métodos definidos por el usuario** para adaptar el mismo a menudo **serán llamados desde el framework**, en lugar de desde el código de aplicación del usuario. El framework a veces **desempeña el papel de programa principal** en la coordinación y secuenciación de actividad de la aplicación. Esta **inversión de control** proporciona al framework la posibilidad de servir como un **esqueleto extensible**. El usuario proporciona métodos que adaptan los algoritmos genéricos.*

ALGUNOS EJEMPLOS DE INVERSIÓN DE CONTROL

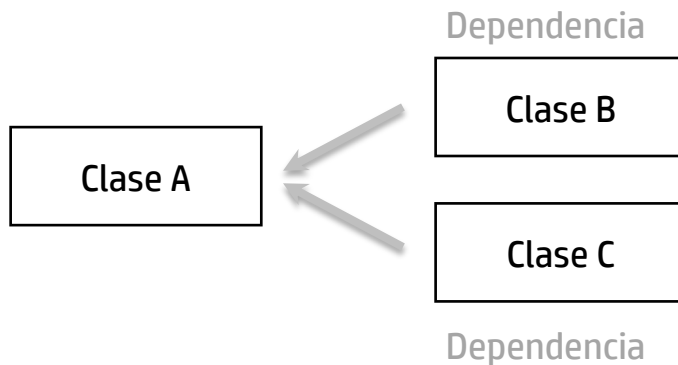
- ▶ Suscripción o manejo de eventos (.NET, Java, ...)
- ▶ Session Bean (EJB): `ejbRemove`, `ejbPassivate`, `ejbActivate`, ...
- ▶ JUnit: `setUp`, `tearDown`, ...
- ▶ Inyección de dependencias: es solo una forma de inversión de control.
- ▶

INYECCIÓN DE DEPENDENCIAS

- ▶ Es una forma de inversión de control.



Modelo tradicional



Inversión de Control
con inyección de
dependencias

MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS

```
public class MovieLister {  
    public Movie[] moviesDirectedBy(String arg)  
    {  
        List<Movie> allMovies = finder.findAll();  
  
        for (Iterator it = allMovies.iterator(); it.hasNext();)   
        {  
            Movie movie = (Movie) it.next();  
            if (!movie.getDirector().equals(arg)) it.remove();  
        }  
  
        return (Movie[]) allMovies.toArray(new  
                                           Movie[allMovies.size()])  
    }  
}
```

MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS

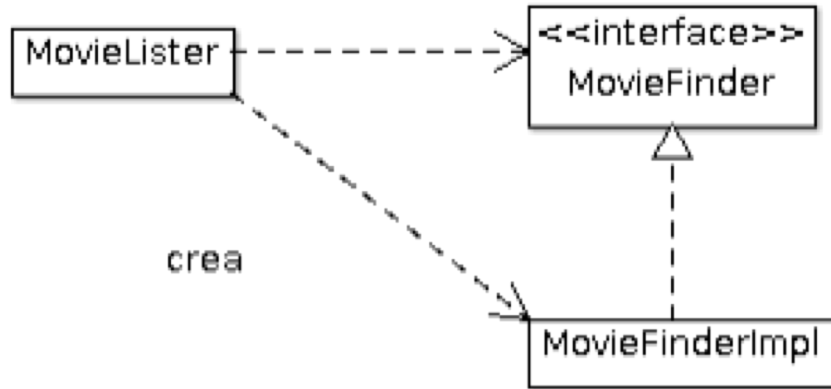
```
public interface MovieFinder
{
    List<Movie> findAll();
}

public class MovieLister {

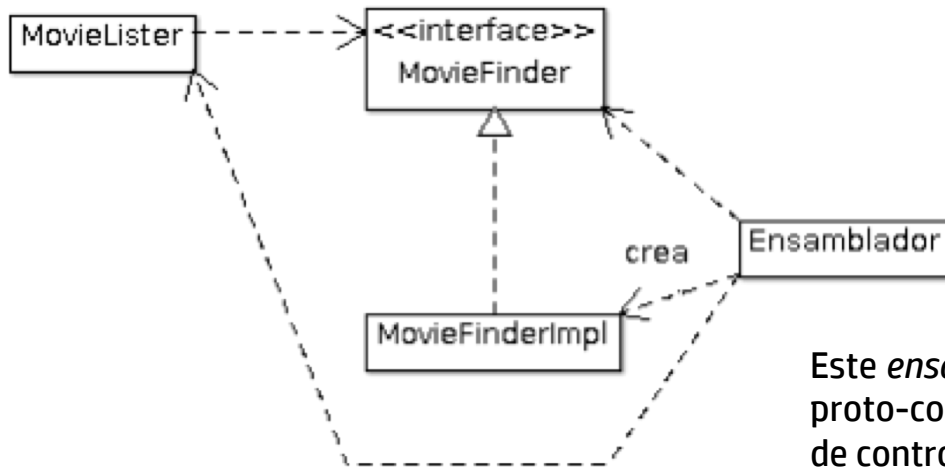
    private MovieFinder finder;

    public MovieLister() {
        finder = new CSVMovieFinder("movies.txt");
    }
    //...
}
```

MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS



MOTIVACIÓN PARA EL USO DE INYECCIÓN DE DEPENDENCIAS



Este *ensamblador* es nuestro proto-contenedor de inversión de control.

EJEMPLO DE INYECCIÓN DE DEPENDENCIAS CON SPRING

```
class MovieLister {  
    private MovieFinder finder;  
  
    public void setFinder(MovieFinder finder) {  
        this.finder = finder;  
    }  
}  
  
class CSVMovieFinder implements MovieFinder {  
  
    public void setFilename(String filename) {  
        this.filename = filename;  
    }  
}
```

EJEMPLO DE INYECCIÓN DE DEPENDENCIAS CON SPRING

```
<beans>
  <bean id="MovieLister" class="spring.MovieLister">
    <property name="finder">
      <ref local="MovieFinder"/>
    </property>
  </bean>
  <bean id="MovieFinder" class="spring.CSVMovieFinder">
    <property name="filename">
      <value>movies.txt</value>
    </property>
  </bean>
</beans>
```

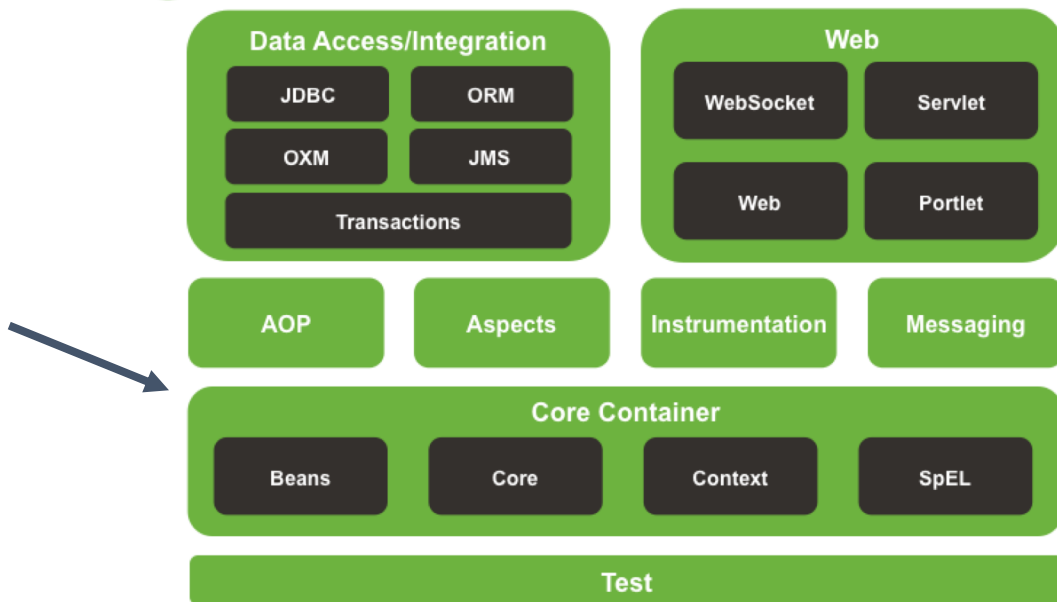
CONTENEDOR DE INVERSIÓN DE CONTROL



NOS SITUAMOS



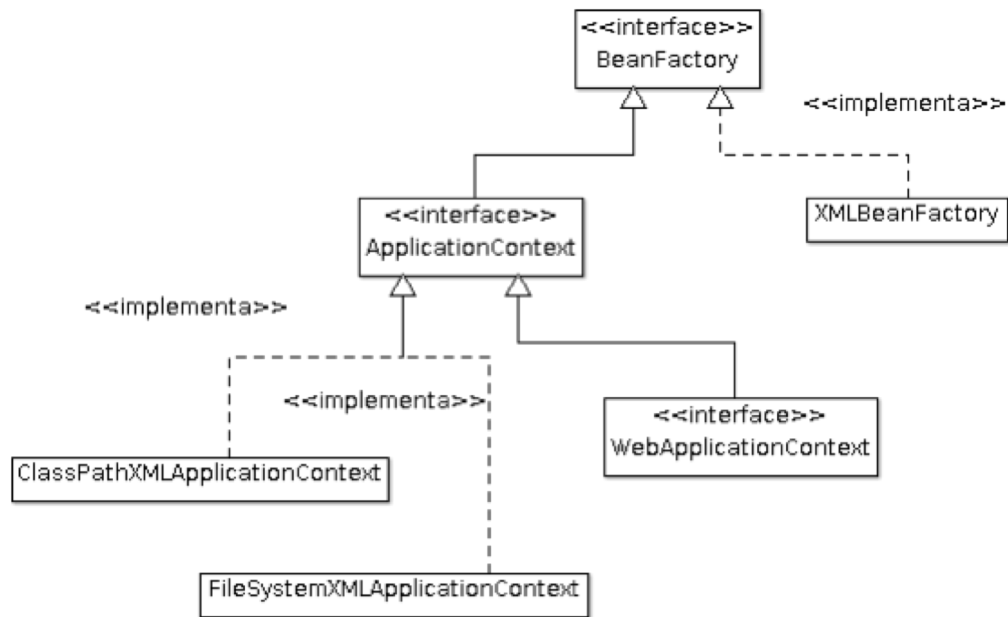
Spring Framework Runtime



LA BASE DEL *IoC* CONTAINER

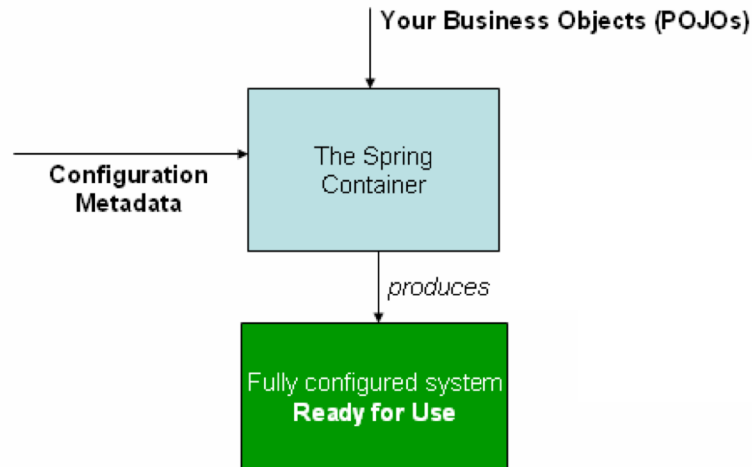
- ▶ Paquetes
 - ▶ `org.springframework.beans`
 - ▶ `org.springframework.context`
- ▶ Los elementos más básicos
 - ▶ BeanFactory: lo elemental para poder manejar cualquier bean objeto.
 - ▶ ApplicationContext: superset del anterior. Añade AOP, manejo de recursos, internacionalización, contextos específicos, ...

LA BASE DEL IoC CONTAINER



BEANS

- ▶ Se trata de un objeto (cualquiera) gestionado por nuestro contenedor de inversión de control.



Podríamos decir
que es como un
objeto
empoderado.

CONFIGURACIÓN DEL CONTENEDOR DE INVERSIÓN DE CONTROL



JAVACONFIG

- ▶ Spring soporta la configuración vía código Java.
- ▶ Nos permite prescindir por completo de XML.
- ▶ Podemos combinar el uso de JavaConfig con las anotaciones trabajadas en el bloque anterior.

ANOTACIONES CLAVE

- ▶ **@Configuration**
 - ▶ A nivel de clase
 - ▶ Indica que una clase va a definir uno o más `@Bean`
- ▶ **@Bean.**
 - ▶ A nivel de método
 - ▶ Equivalente a `<bean ... />`

JAVACONFIG BÁSICO

@Configuration

```
public class AppConfig {
```

@Bean

```
    public Saludator saludator() {  
        return new Saludator();  
    }
```

```
}
```

INSTANCIACIÓN DEL CONTENEDOR

- ▶ Ahora usamos `AnnotationConfigApplicationContext`
- ▶ Recibe como argumento la/s clase/s que tienen alguna configuración.

```
public static void main(String[] args) {  
    ApplicationContext appContext = new  
        AnnotationConfigApplicationContext(AppConfig.class);  
    Saludador saludador = appContext.getBean(Saludador.class);  
    //...  
}
```


INSTANCIACIÓN DEL CONTENEDOR

- ▶ Podemos usar el constructor vacío y registrar las clases.

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx =  
        new AnnotationConfigApplicationContext();  
    ctx.register(AppConfig.class, OtherConfig.class);  
    ctx.register(AdditionalConfig.class);  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

ESCANEIO DE COMPONENTES

- ▶ Idéntico comportamiento que en XML
- ▶ `@ComponentScan(basePackages=...)`
- ▶ También programáticamente

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx =  
        new AnnotationConfigApplicationContext();  
    ctx.scan("com.acme");  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
}
```

ESTEREOTIPOS

- ▶ **@Component**
 - ▶ Estereotipo básico
 - ▶ Los demás son derivados de él.
- ▶ **@Service**: orientado a las clases servicio, lógica de negocio, ...
- ▶ **@Repository**: clases de acceso a datos (DAO)
- ▶ **@Controller**: clases que sirven para gestionar las peticiones recibidas.

INYECCIÓN AUTOMÁTICA O AUTOCABLEADO



INYECCIÓN AUTOMÁTICA

- ▶ Spring permite la inyección *automática* entre beans que *se necesitan*.
- ▶ Busca candidatos dentro del contexto.
- ▶ Ventajas
 - ▶ Reduce la configuración necesaria
 - ▶ Útil durante el desarrollo. Permite requerir objetos sin configurarlo explícitamente.

TIPOS DE AUTOWIRED

- ▶ **no:** sin autocableado
- ▶ **byName:** en función del nombre de la propiedad requerida.
- ▶ **byType:** en función del tipo de la propiedad requerida. Si hay más de un bean de este tipo, se produce excepción.
- ▶ **constructor:** análogo a *byType*, pero para argumentos del constructor.

INCONVENIENTES DEL AUTOCABLEADO

- ▶ Es útil si se usa siempre en un proyecto.
- ▶ En otro caso, puede ser confuso.
- ▶ No se pueden *autoinyectar* tipos primitivos o String.
- ▶ Menos exacto que la inyección explícita
- ▶ Posible ambigüedad en inyección *byType*.

INCONVENIENTES DEL AUTOCABLEADO: ¿QUÉ HACER?

- ▶ No usar el autocableado :(
- ▶ Manejar el autocableado a través de anotaciones (lo estudiaremos más adelante).
- ▶ Utilizar `autowired-candidate=false` en los beans más conflictivos.
- ▶ Utilizar `primary=true` en las opciones principales.

USO DE @AUTOWIRED

- ▶ Busca un bean adecuado y lo inyecta en la dependencia.
- ▶ Se realiza un autocableado *byType*

¿DÓNDE PUEDO USAR @AUTOWIRED?

- ▶ Método setter

```
@Autowired  
public void setPelículaDao(PelículaDao películaDao) {...}
```

- ▶ Definición de la propiedad

```
@Autowired  
private PelículaDao películaDao;
```

- ▶ Constructor

```
@Autowired  
public PelículaService(PelículaDao películaDao) {...}
```

USO DE @AUTOWIRED

- ▶ Se pueden mezclar los 3 tipos de uso de autowired.
 - ▶ En la propiedad es muy cómodo.
 - ▶ Si el método setter tiene alguna “lógica especial”, sería adecuado.
 - ▶ Para atributos *final*, usamos el constructor.

@AUTOWIRED DE VARIOS OBJETOS DE DIFERENTE TIPO.

- ▶ No hay limitación en el número de argumentos de un método anotado con @Autowired.

```
public class MovieRecommender {  
    private MovieCatalog movieCatalog;  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(MovieCatalog movieCatalog,  
        CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
    // ...  
}
```

USO DE @AUTOWIRED PARA VARIOS OBJETOS DEL MISMO TIPO

- ▶ Podemos obtener todos los beans de un mismo tipo
 - ▶ Array
 - ▶ Colección: List, Set, Map

```
public class PeliculaDaoImplMemory implements PeliculaDao {  
    @Autowired  
    private Set<CatalogoPelículas> catalogosPelículas;  
}
```

@AUTOWIRED NO SATISFECHO

- ▶ Si @Autowired no encuentra ningún bean candidato produce excepción.
- ▶ Podemos modificar este comportamiento para que deje la dependencia sin satisfacer, pero sin excepción:
 - ▶ @Autowired(required=false)
 - ▶ @Nullable (Spring 5)
 - ▶ Optional<?> (Java 8)



USO DE @REQUIRED

USO DE @REQUIRED

- ▶ Nos permite indicar que una propiedad debe ser necesariamente inyectada.
- ▶ No indica cómo debe realizarse la inyección
 - ▶ Explícita
 - ▶ Autowired
 - ▶ ...
- ▶ Si no se satisface, produce una excepción.
- ▶ Permite evitar NPE



USO DE @PRIMARY Y @QUALIFIER

USO DE @PRIMARY

- ▶ Ante varios beans de un tipo, es el primer candidato (*primus inter pares*)
- ▶ A nivel de clase (@Component y derivados)
- ▶ A nivel de método (@Bean)

USO DE @PRIMARY

```
@Primary
@Component
public class HibernateFooRepository extends FooRepository
{
    public HibernateFooRepository(
        SessionFactory sessionFactory) {
        // ...
    }
}
```

USO DE @QUALIFIER

- ▶ Nos permite afinar mucho más el autocableado.
- ▶ Podemos *seleccionar* que bean específico (de entre varios de un tipo) queremos inyectar.
- ▶ Mecanismo extensible.

USO DE @QUALIFIER

- ▶ El mecanismo más sencillo es usar el nombre del bean.

```
public class PeliculaDaoImplMemory
                                implements PeliculaDao {

    @Autowired
    @Qualifier("catalogoClasicas")
    private CatalogoPeliculas catalogoPeliculas;

    //...

}
```

USO DE @QUALIFIER

- ▶ También podemos usar @Qualifier a nivel de argumento de un método.

```
public class MovieRecommender {  
  
    @Autowired  
    public void prepare(  
        @Qualifier("main") MovieCatalog movieCatalog,  
        CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
    // ...  
}
```