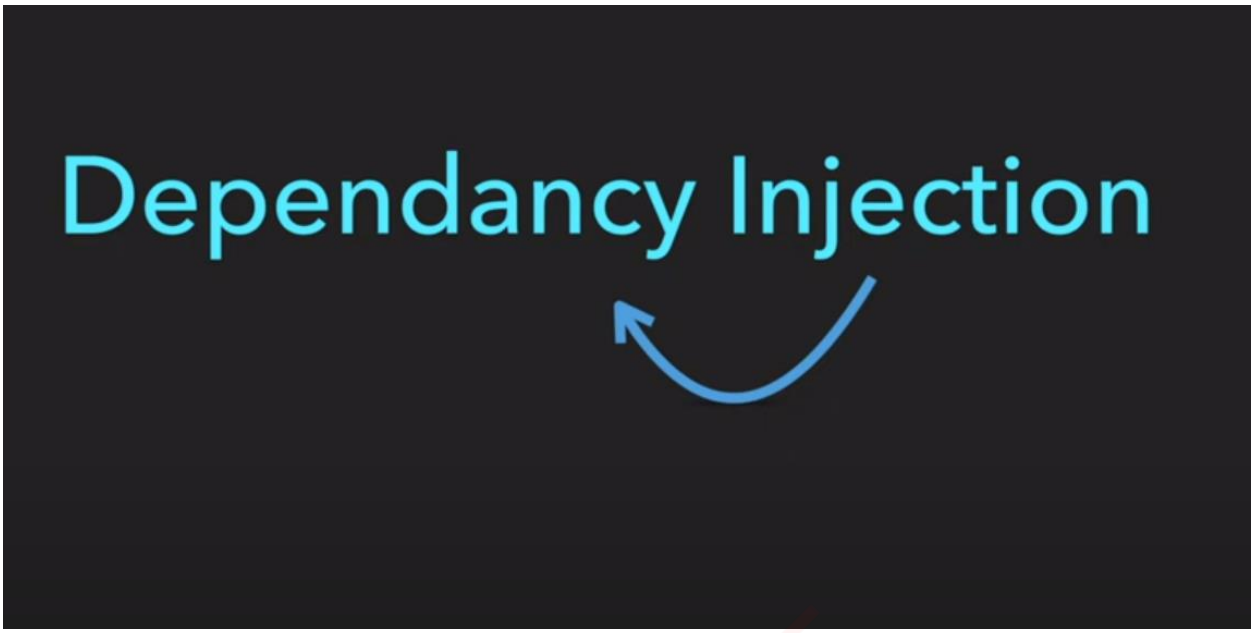

Refresher

Create three classes (Student, University and Main) and find out a hole)

1st: try it without a field

```
package com.itsc.di;

public class Student {
	public void displayStudentInfo() {
		System.out.println("student info here");
	}
}
```

```
package com.itsc.di;

public class Main {
	public static void main(String[] args) {
		Student student = new Student();
		student.displayStudentInfo();

	}
}
```


Use XML based configuration and apply IoC principle on the above case:

What if there is a property/field?

```
package com.itsc.di;

public class Student {
	private University university;

	public void setUniversity(University university) {
		this.university = university;
	}

	public void displayStudentInfo() {
		System.out.println(university.getName() + " university");
	}
}
```

```
package com.itsc.di;

public class University {
	private String name;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

```
package com.itsc.di;

public class Main {
	public static void main(String[] args) {
		Student student = new Student();
		University university = new University();
		university.setName("Addis Ababa");
		student.setUniversity(university);

		student.displayStudentInfo();

	}
}
```


What have you noticed? Tight coupling? Dependency Injection? Absence of Inversion of Control principle?

Dependency Injection

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 it is constructed or returned from a factory method.

The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes

Code is cleaner with the DI principle, and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies. As a result, your classes become easier to test, particularly when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants: **Constructor-based dependency injection** and **Setter-based dependency injection**.

Setter-based DI

Setter-based DI is accomplished by the container calling **setter methods** on your beans after invoking a **no-argument constructor**

The DI will be injected with the help of setter and/or getter methods.

Example:

Let's have a **Student** class with **studentName** as a property (Dependencies in form of **literals**):

```
package com.itsc.di;

public class Student {
	private String studentName;

	//generate setters and getters for studentName below
	// use shortcuts to generate the methods
	….
	…
	public void displayStudentInfo() {
		System.out.println("The student name is: " + studentName);
	}
}
```

Let's have a launcher class (**Main** class)

```
package com.itsc.di;

public class Main {
	public static void main(String[] args) {
		Student student = new Student();
		student.setStudentName("John");
		student.displayStudentInfo();
	}
}
```

In this code, let's use **Inversion of Control** principles.

```
package com.itsc.di;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
	public static void main(String[] args) {

		//load the xML configuration

		ApplicationContext context = new
				ClassPathXmlApplicationContext("config.xml");

		//retrieve the student bean
		Student student = context.getBean("student", Student.class);

		student.displayStudentInfo();
	}
}
```

XML Configuration File (config.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

      <!--  define Student bean -->
      <bean id="student" class="com.itsc.di.Student">
            <property name="studentName" value="John"></property>
      </bean>
</beans>
```

Let's discuss about the following snippet of the config.xml part

```
      <bean id="student" class="com.itsc.di.Student">
            <property name="studentName" value="John"></property>
      </bean>
```

- `<bean>`: The `<bean>` element is used to define a bean in the DI container. In this case, it creates a bean of the `Student` class.
- `id="student"`: The `id` attribute specifies the unique identifier for the bean within the container. In this case, the bean is assigned the ID "student".
- `class="com.itsc.di.Student"`: The `class` attribute specifies the fully qualified class name of the bean that will be instantiated. In this case, it refers to the `Student` class.
- `<property>`: The `<property>` element is used to set the values of properties (fields) in the bean.
- `name="studentName"`: The `name` attribute specifies the name of the property in the `Student` class that will be set. In this case, it refers to the `studentName` property.
- `value="John"`: The `value` attribute specifies the value that will be assigned to the `studentName` property. In this case, the value "John" will be set.

So, when the **DI container** initializes, it will create an instance of the **Student class** and set the **studentName** property to "**John**". This allows the Student object to be created with its dependencies automatically injected, providing the necessary data for the object to work correctly.

In the provided XML configuration, the DI container will use the **setter method** of the Student class to set the value of the **studentName** property to "John".

The **DI container** analyzes the **bean definition** and identifies the properties that need to be set. It does this by matching **the property names** in the **bean definition** with the corresponding **setter methods** in the class.

That is why you need to create a setter method for the property of the class:

```
public void setStudentName(String studentName) {
		this.studentName = studentName;
	}
```

When the **DI container** creates an instance of the **Student class**, it will invoke the **setStudentName** method and pass the value "John" as an argument. This sets the **studentName** property of the Student object to "John".

What if there are dependencies in form of objects:

```
public class Student {
	private String studentName;
	private University university;
```

From the above code snippet, the **Student** class has a dependency on the **University** class.

A **dependency** in software development refers to a relationship between classes or components where one class depends on another to fulfill its functionality.

In this case, the **Student** class has a **property** named **university** of type **University**, indicating that it relies on the **University** class to provide the necessary functionality or data related to the university associated with the student.

The **University** class represents an external component or resource that the **Student** class depends on. It is responsible for providing the necessary information about the university, such as its name, location, or any other relevant details. Therefore, **University** is the **dependency** in this scenario.

So let's modify the **Student** class as:

```
package com.itsc.di;

public class Student {
	private String studentName;
	private University university;


	//generate setters and getters for studentName and university
	// use the shortcuts
	……..
	……..
	public void displayStudentInfo() {
	  so.printf("%s is admitted to %s", studentName, university.getName());
	}
}
```

Let's create the **University** class now

```
package com.itsc.di;

public class University {
	private String name;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Modify the config.xml file accordingly:

```
<!-- define the University bean -->

<bean id="aau" class="com.itsc.di.University">
      <property name="name" value="Addis Ababa University" />
</bean>

<!-- define the Student bean -->
<bean id="student" class="com.itsc.di.Student">
      <property name="studentName" value="John" />
      <property name="university" ref="aau" />
</bean>
```

The <**property**> element in XML can have a **ref** attribute, which is used to **reference** another **bean** within the **DI container**.

When using the **ref** attribute, the value provided refers to the **ID** of another bean in the container. This allows the DI container to resolve the dependency and inject the referenced bean into the property of the current bean being configured.

……

Let's modify the **Main** class to see the effect:

```
//load the xML configuration
ApplicationContext context = new
                              ClassPathXmlApplicationContext("config.xml");

//retrieve the student bean
Student student = context.getBean("student", Student.class);

//use the Student object
System.out.println("Student Name: " + student.getStudentName());
System.out.println("University: " + student.getUniversity().getName());

//student.displayStudentInfo();
```

This demonstrates that the **Student** object is successfully created with its properties set through **setter-based dependency injection**, and the associated **University** object is correctly **injected** as a **dependency**.

Constructor-based Dependency Injection

Example:
Let's create a **Student** class with two properties: studentName & university

Ceteris paribus in the **Student** class, add the following **constructor**:

```
public Student(String studentName, University university) {
	this.studentName = studentName;
	this.university = university;
}
```

The only part you need to modify for the code to work using constructor-based DI is the **config.xml** file:

```
 <!-- define the University bean -->

<bean id="aau" class="com.itsc.di.University">
	<property name="name" value="Addis Ababa University" />
</bean>

<!-- define the Student bean -->

<bean id="student" class="com.itsc.di.Student">
	<constructor-arg name="studentName" value="John" />
	<constructor-arg name="university" ref="aau" />
</bean>
```

In this XML configuration, we define the student bean. We use the **<constructor-arg>** element to specify the arguments for the **constructor** of the **Student** class.

The **value** attribute is used for literal values, and the ref attribute is used to refer to another bean by its id.

In this configuration, the **<constructor-arg>** elements include the **name** attribute, which corresponds to the **parameter names** of the **Student constructor**. By specifying the parameter names, you can disambiguate the values being passed to the constructor.

So, the first **<constructor-arg>** with **name="studentName"** sets the **studentName** argument of the Student constructor, and the second **<constructor-arg>** with name="university" sets the university argument to reference the **aau** bean of type **University**.

What if **University** class has its own constructor?

```
package com.itsc.di;

public class University {
	private String name;

	public University(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Just modify the **config**.xml file to entertain the changes so that the bean definition of the **University** class has a **constructor-arg** element

```
<!-- define the University bean -->
	<bean id="aau" class="com.itsc.di.University">
		<constructor-arg name="name" value="Addis Ababa University" />
	</bean>
```

The advantage of using “ref” to nested beans

Assume we’ve a **Teacher** class as below:

```
package com.itsc.di;

public class Teacher {
	private String teacherName;
	private University university;

	// generate constructor here
	……

	// generate setters and getters here
	……
```

```
	public void displayStudentInfo() {
		System.out.printf("%s is assigned to %s", teacherName,
university.getName());
	}
}
```

Now when you want to define beans in **config**.xml, you will no longer repeat the definition of **University** bean, if **ref** is used:

```
<!-- define the University bean -->
	<bean id="aau" class="com.itsc.di.University">
		<constructor-arg name="name" value="Addis Ababa University" />
	</bean>
	<!-- define the Student bean -->
	<bean id="student" class="com.itsc.di.Student">
		<constructor-arg name="studentName" value="John" />
		<constructor-arg name="university" ref="aau" />
	</bean>

	<!-- define the Teacher bean -->
	<bean id="teacher" class="com.itsc.di.Teacher">
		<constructor-arg name="teacherName" value="Haile" />
		<constructor-arg name="university" ref="aau" />
	</bean>
```

You see, **University** is a common dependency for both **Student** and **Teacher objects.**

Example:

```
package com.itsc;

public class Student {
	private String studentName;

	public Student(String studentName) {
		this.studentName = studentName;
	}



	public void setStudentName(String studentName) {
		this.studentName = studentName;
	}


}
```

```
package com.itsc;

public class School {
	private Student student;

	public School(Student student) {
		this.student = student;
	}

	public void setStudent(Student student) {
		this.student = student;
	}

	public void displayStudentInfo() {
		System.out.println("Student Name: " +
student.getStudentName());
	}
}
```

```
<!--  Student Bean -->
	  <bean id = "studentBean" class="com.itsc.Student">
```

```
        <constructor-arg name="studentName" value="John" />
    </bean>

    <!--  School Bean -->
    <bean id = "schoolBean" class="com.itsc.School">
        <constructor-arg name="student"
ref="studentBean"></constructor-arg>
    </bean>
```

```
public class Main {
    public static void main(String[] args) {
        var context = new
            ClassPathXmlApplicationContext("beansConfig.xml");
        School school =
                context.getBean("schoolBean", School.class);

        school.displayStudentInfo();
    }
}
```

A combination of constructor injection and setter injection

Let's modify the `School` class to demonstrate this:

```
package com.itsc;

public class School {
	private Student student;

	private String location;

	public School(Student student) {
		this.student = student;
	}

	public void setLocation(String location) {
		this.location = location;
	}

	public Student getStudent() {
		return student;
	}

	public void setStudent(Student student) {
		this.student = student;
	}

	public void displayStudentInfo() {
		Sout(student.getStudentName() + " lives at " + location);
	}
}
```

Leave “Student” class intact.

In this modified `School` class, we've added a `location` field along with its setter method.

Now, let's update the XML configuration to inject the `location` using setter injection and `Student` using constructor injection:

```
<!--  Student Bean -->
<bean id = "studentBean" class="com.itsc.Student">
  <constructor-arg name="studentName" value="John" />
</bean>

<!--  School Bean -->
<bean id = "schoolBean" class="com.itsc.School">
  <constructor-arg name="student" ref="studentBean" />
  <property name = "location" value = "Addis Abbaba" />
</bean>
```

Now, the `School` bean is being instantiated with the `Student` dependency using constructor injection (`<constructor-arg ref="studentBean" />`) and the `location` property is set using setter injection (`<property name="location" value="Addis Ababa" />`).

When you retrieve the `School` bean in your application, it will have both dependencies injected through different methods, demonstrating the combination of constructor-based and setter-based dependency injection.

More Example: Employee Management System

Step 1: Define Employee and Department Classes

Create `Employee.java`:

```
package com.itsc;

public class Employee {
	private String name;
	private int employeeId;

	public Employee(String name, int employeeId) {
		this.name = name;
		this.employeeId = employeeId;
	}

	public String getName() {
		return name;
	}

	public int getEmployeeId() {
		return employeeId;
	}


}
```

Create **`Department.java`**:

```
package com.itsc;

import java.util.List;

public class Department {
	private String name;
	private List<Employee> employees;

	public Department(String name, List<Employee> employees) {
		this.name = name;
		this.employees = employees;
	}
	public String getName() {
		return name;
	}
	public List<Employee> getEmployees() {
		return employees;
	}


}
```

Step 2: Implement EmployeeManagementService Class

Create `EmployeeManagementService.java`:

Now, let's create a service responsible for managing employees and departments. It'll use Dependency Injection to receive the necessary components, such as a list of employees and departments.

```
package com.itsc;

import java.util.List;

public class EMgmtService {
     private List<Employee> employees;
     private List<Department> departments;

     public EMgmtService(
                         List<Employee> employees,
                         List<Department> departments) {
          this.employees = employees;
          this.departments = departments;
     }

     public void displayEmployees() {
          System.out.println("Employees: ");
          for (Employee employee : employees) {
               Sout("Name: " + employee.getName() + ", ID: "
                         + employee.getEmployeeId());
          }
     }

     public void displayDepartments() {
          System.out.println("Departments: ");
          for (Department department : departments) {
               Sout("Department: " + department.getName());
               System.out.println("Employees: ");
               for (Employee emp : department.getEmployees()) {
                    System.out.println("- " + emp.getName());
               }
               System.out.println();
          }
     }


}
```

Step 3: Create applicationContext.xml

```
 <!-- define employee beans -->
<bean id="emp1" class="com.itsc.Employee">
      <constructor-arg value="John" />
      <constructor-arg value="1" />
</bean>

<bean id="emp2" class="com.itsc.Employee">
      <constructor-arg value="Alice" />
      <constructor-arg value="2" />
</bean>

<!-- define department beans -->
<bean id="dept1" class="com.itsc.Department">
      <constructor-arg value="Engineering" />
      <constructor-arg>
            <list>
                  <ref bean="emp1"/>
            </list>
      </constructor-arg>
</bean>

<bean id="dept2" class="com.itsc.Department">
      <constructor-arg value="Philosophy" />
      <constructor-arg>
            <list>
                  <ref bean="emp2"/>
            </list>
      </constructor-arg>
</bean>


<!-- define EmployeeManagementService bean with DI -->
<bean id = "empmgmtservice" class="com.itsc.EMgmtService">
      <constructor-arg>
            <list>
                  <ref bean = "emp1"/>
                  <ref bean = "emp2"/>
            </list>
      </constructor-arg>

      <constructor-arg>
            <list>
                  <ref bean = "dept1"/>
                  <ref bean = "dept2"/>
            </list>
      </constructor-arg>
</bean>
```

Step 4: Implement MainApp

Create `MainApp.java`:

```
package com.itsc;


public class MainApp {
     public static void main(String[] args) {
          var context = new
                         ClassPathXmlApplicationContext(
                             "applicationContext.xml");

          EMgmtService service =
                         context.getBean(
                                        "empmgmtservice",
                                         EMgmtService.class);

          service.displayDepartments();
     }
}
```