Spring basic programs with java

Setter Injection:

In setter injection, dependencies are provided to the class through setter methods after the object is created. This approach allows for optional dependencies and provides flexibility in setting dependencies after object creation.

Program:

(managar class)

```
package com.slokam;


public class manager {


	private Teamlead t1;



	public void setT1(Teamlead t1) {
		this.t1 = t1;
	}



	public void workdone() {


		System.out.println("manager work is started");
		t1.workdone();
		System.out.println("manager work is completed");
	}
}
```

```
package com.slokam;                                          (Teamlead class)


public class Teamlead {


	private deveploer dev1;
	public void setDev1(deveploer dev1) {
		this.dev1 = dev1;
	}



	public void workdone() {

		System.out.println("Teamlead work is started");
		dev1.workdone();
		System.out.println("Teamlead work is completed");
	}

}
package com.slokam;                                   (deveploer class)

public class deveploer {

	public void workdone() {

		System.out.println("deveploer work is started");
		System.out.println("deveploer work is completed");
	}

}
package com.slokam;                                   (Test class)

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

                manager m = new manager();

                Teamlead t = new Teamlead();
                m.setT1(t);

                deveploer d = new deveploer();
                t.setDev1(d);

                m.workdone();

        }

}
```

Constructor Injection:

In constructor injection, dependencies are provided to the class through its constructor. This is the most recommended form of dependency injection because it makes the dependencies required for an object to be in a valid state explicit.

Program:

(managar class)

```
package com.cisco;

public class managar {

        private teamlead t1;

        public managar(teamlead t1) {
                this.t1 = t1;
        }
```

```
    public void workdo() {

        System.out.println("manager work is started");
        t1.workdo();
        System.out.println("manager work is completed");
    }

}
```

(teamlead class)

```
package com.cisco;

public class teamlead {

    private deveploer dev;

    public teamlead(deveploer dev) {
        this.dev = dev;
    }

    public void workdo() {

        System.out.println("teamlead work is started");
        dev.workdo();
        System.out.println("teamlead work is completed");
    }

}
```

(Devepoler class)

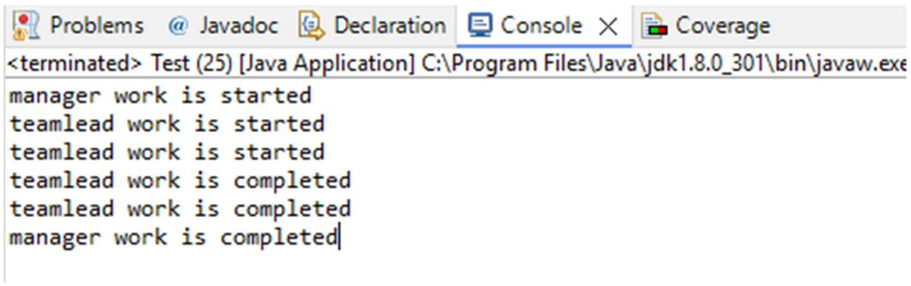
```
package com.cisco;
```

```
public class deveploer {



public void workdo() {

                    System.out.println("teamlead work is started");
                    System.out.println("teamlead work is completed");
          }


}
```

(Test class)

```
package com.cisco;

public class Test {
          public static void main(String[] args) {
                    deveploer dev = new deveploer();
                    teamlead t = new teamlead(dev);
                    managar m = new managar(t);
                    m.workdo();
          }
}
```

Output:



```
<terminated> Test (25) [Java Application] C:\Program Files\Java\jdk1.8.0_301\bin\javaw.exe
manager work is started
teamlead work is started
teamlead work is started
teamlead work is completed
teamlead work is completed
manager work is completed
```

"Is-a" Relationship:

The **"is-a"** relationship represents **inheritance**. When one class is a specialized form of another class, it's an **"is-a"** relationship. This means that the subclass can be treated as an instance of the superclass, and it inherits the behaviors and properties of the superclass.

In Java, this is implemented using the extends keyword for class inheritance or implements for implementing an interface.

Example: "Is-a" Relationship

Consider an example with Animal and Dog. A Dog **is an** Animal, so we can use inheritance here.

Program:

```
// Superclass
public class Animal {
    public void makeSound() {
        System.out.println("Animal sound");
    }
}

// Subclass representing the "is-a" relationship
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

// Main class to demonstrate the "is-a" relationship
public class MainApplication {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Dog is an Animal
        animal.makeSound(); // Output: Bark
    }
}
```

Here:

- **Dog is an Animal** (i.e., "is-a" relationship).
- We can substitute Dog wherever an Animal is expected, following the **Liskov Substitution Principle** (a core concept in OOP).
- Dog inherits the properties and behaviors of Animal, so it can be treated as an Animal.

"Has-a" Relationship

The **"has-a"** relationship represents **composition** (or aggregation). In this relationship, a class contains an instance of another class, rather than inheriting from it. This implies that one class **owns** or **has** a reference to another class as a part of its data.

In Java, this is implemented by simply declaring an instance variable of another class within the class. It's a powerful way to build complex objects from simpler ones while keeping the class hierarchy shallow.

Example: "Has-a" Relationship

Consider an example where a Car **has-a** Engine. A Car object is composed of an Engine object, but a Car is not an Engine.

Program:

```
// Engine class

public class Engine {

    public void start() {

        System.out.println("Engine started");

    }

}


// Car class demonstrating the "has-a" relationship with Engine

public class Car {

    private Engine engine; // Car "has-a" Engine


    // Constructor to inject an Engine

    public Car(Engine engine) {

        this.engine = engine;

    }
    public void startCar() {
        engine.start();
        System.out.println("Car started");
    }
}

// Main class to demonstrate the "has-a" relationship
public class MainApplication {
    public static void main(String[] args) {
        Engine engine = new Engine(); // Create an Engine
        Car car = new Car(engine); // Inject the Engine into the Car

        car.startCar(); // Output: Engine started
                        //         Car started
    }
}
```

Here:

- **Car has an Engine** (i.e., "has-a" relationship).
- Car has an instance of Engine as a field, which means Car can use Engine's functionality but does not inherit from Engine.
- This composition allows Car to functionally contain an Engine without making Car a subtype of Engine.

Spring:

What is Spring?

Spring Framework is an open-source, lightweight, and powerful framework for building Java-based applications. It simplifies the development process by providing tools, libraries, and patterns to create scalable, robust, and maintainable applications.

Originally created to address the complexities of enterprise Java (like J2EE), Spring has evolved into a comprehensive ecosystem for all types of applications, including web, microservices, and cloud-based systems.

Annotation

Annotation	Purpose
@Component	Marks a class as a Spring bean (automatically detected).
@Autowired	Injects dependencies automatically.
@Configuration	Declares a class as a configuration source.
@ComponentScan	Scans specified packages for Spring components.
@Bean	Declares a bean using a method.
@Qualifier	Specifies which bean to inject when there are multiple options.

1. @Component

Purpose: Tells Spring, "This is a class I want you to manage for me."

Use Case: Use it for any general-purpose class.

Example:

```
@Component
public class MyService {
   public void doSomething() {
       System.out.println("Doing something...");
   }
}
```

2. @Autowired

- Purpose: Automatically connects one class to another that it depends on.
- Use Case: Saves you from writing new or manually wiring objects.
- Example:

```
@Component
public class MyController {
   @Autowired
   private MyService myService;

   public void handleRequest() {
      myService.doSomething();
   }
}
```

---

3. @Configuration

- Purpose: Indicates that a class contains setup instructions for Spring (like bean definitions).
- Use Case: Use it to configure the application without XML files.
- Example:

```
@Configuration
public class AppConfig {
   @Bean
   public MyService myService() {
      return new MyService();
   }
}
```

---

4. @ComponentScan

- Purpose: Tells Spring where to look for @Component classes.
- Use Case: Use it when your @Component classes are in a different package.
- Example:

```
@ComponentScan(basePackages = "com.example.services")
public class AppConfig { }
```

---

5. @Bean

- Purpose: Creates and returns an object (bean) for Spring to manage.
- Use Case: Use it for classes that you can't or don't want to annotate with @Component.
- Example:

```
@Configuration
public class AppConfig {
   @Bean
   public MyService myService() {
      return new MyService();
   }
}
```

---

6. @Qualifier

- Purpose: Specifies which bean to use when there are multiple beans of the same type.
- Use Case: Avoids confusion when Spring finds two or more possible beans.
- Example:

```
@Component("firstService")
public class FirstService { }

@Component("secondService")
public class SecondService { }
@Component
public class MyController {
   @Autowired
   @Qualifier("secondService")
   private SecondService service;
}
```

Sample Program with Annotation:

```
package com.slokam;                                    (manager class)

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

// user find only used component
@Component
public class manager {

	@Autowired
	private teamlead t1;

	public void dowork() {

		System.out.println("manager work is started");
		t1.workdone();
		System.out.println("manager work is completed");

	}

}
package com.slokam;                                    (teamlead classs)

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class teamlead {
	@Autowired
	private deveploer dev;

	public void workdone() {

		System.out.println("teamlead work is started");
		dev.dowork();
		System.out.println("teamlead work is completed");

	}

}
package com.slokam;                                    (deveploer class)

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class deveploer {


	public void dowork() {

		System.out.println("deveploer work is started");

		System.out.println("deveploer work is completed");
	}

}
```

```
package com.slokam;                                   (ourconfinguration class)

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.slokam")
public class ourConfiguration {



}
package com.slokam;                                   (Test class)

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Test {

	public static void main(String[] args) {

		ApplicationContext on = new AnnotationConfigApplicationContext(ourConfiguration.class);

		manager m = on.getBean(manager.class);

		m.dowork();

	}

}
```

Spring Boot Basic Information Related to Application

what type of application we are using?

1. Standalone Application:
   A standalone application is an application that runs on a local system.
   Example: A program without any user interface (UI) interaction.
2. Web Application:
   A web application includes a user interface, business logic, and a database.
   Example: Applications that integrate UI interfaces with back-end business logic and databases.
3. Mobile Application:
   A mobile application includes a mobile-specific user interface, business logic, and a database.
   Example: Zepto
4. Back-end Application
   A back-end application contains only business logic and a database.

Key Point for Web Application:

→Not for end-user but for front end application or not application

→It returns Json or xml

→Backend application allows multiple types of requests and gives different types of response.

→To Handle any request, we need to provide request handle(method) that contains logic and returns Json data.

→Any Back-end application contains multiple request handlers.

→Application should be in server for example tomcat.

Java object is converted into Json format data

1. Datatypes:
   - String: Welcome to Hyderabad...!
   - Integer: 121
   - Double: 56.36
   - Boolean: true
2. Datatypes []:
   - String []: ["sai","madhu","akshu","raju"]
   - Int []: [32,56,45,96,74,85]
   - Double []: [56.35,85.25,75.45,42.36,65.98,98.24]
   - Char []: "ABCDEF"
3. Object:
   - Object can take any kind of values like int, String, Boolean, char, Double …etc.
   - Object can take per-defined class and user-defined class.
   - Object: teja → For Per-defined class
   - Object: emp Object → user-defined class

     : {"sid":1,"sname":"madhavi","sage":25,"smarks":75.36}
4. Object []:
   - Object [] can take any kind of data like int, String, Boolean, Char, Double…. etc.
   - Object can take per-defined class and user-defined class.
   - Object []: ["sai",53,86.36,"A"]
   - Object[]: ["teja",63.23,{"sid":1,"sname":"madhavi","sage":25,"smarks":75.36}]
5. Employeepojo:
   - Employeepojo return type should be employee data.
   - It consist of the employee data. Example: Eid, Ename, Esal, Edegs, Elocation.
   - Employeepojo: {"sid":1,"sname":"madhavi","sage":25,"smarks":75.36}

6. Employeepojo []:
   - Employee[]: [{"sid":1,"sname":"madhavi","sage":25,"smarks":75.36}, {"sid":2,"sname":"aditya","sage":26,"smarks":85.36}, {"sid":3,"sname":"Raghu","sage":28,"smarks":98.36}]

7. List<Employeepojo> using user-defined class:
    - List<Employeepojo>:
      [{"sid":1,"sname":"madhavi","sage":25,"smarks":75.36},
      {"sid":2,"sname":"aditya","sage":26,"smarks":85.36},
      {"sid":3,"sname":"Raghu","sage":28,"smarks":98.36}]
    - List<String>: →using pre-defined class:
      List<String>: ["Suresh"]
8. Set<Employeepojo> using user-defined class:
    - Set<Employeepojo>:
      [{"sid":2,"sname":"aditya","sage":26,"smarks":85.36},
      {"sid":3,"sname":"Raghu","sage":28,"smarks":98.36},
      {"sid":1,"sname":"madhavi","sage":25,"smarks":75.36}]
    - Set<String>: →using pre-defined class:
      Set<String>: ["Siva","sai","Teja"]
9. Map<Key, value>: Using user-defined class:
    - Map<Employeepojo Integer>:
      {"studentpojo [sid=1, sname=madhavi, sage=25, smarks=75.36]":50,
      "studentpojo [sid=3, sname=Raghu, sage=28, smarks=98.36]":52,
      "studentpojo [sid=2, sname=aditya, sage=26, smarks=85.36]":51}
    - Map<String Integer> : → Using per-defined class:
      {"swathi":103,"adhi":102,"sai":101}