selsoft

build better, deliver faster

Enterprise Application Development with Spring

Chapter 2: Dependency Injection

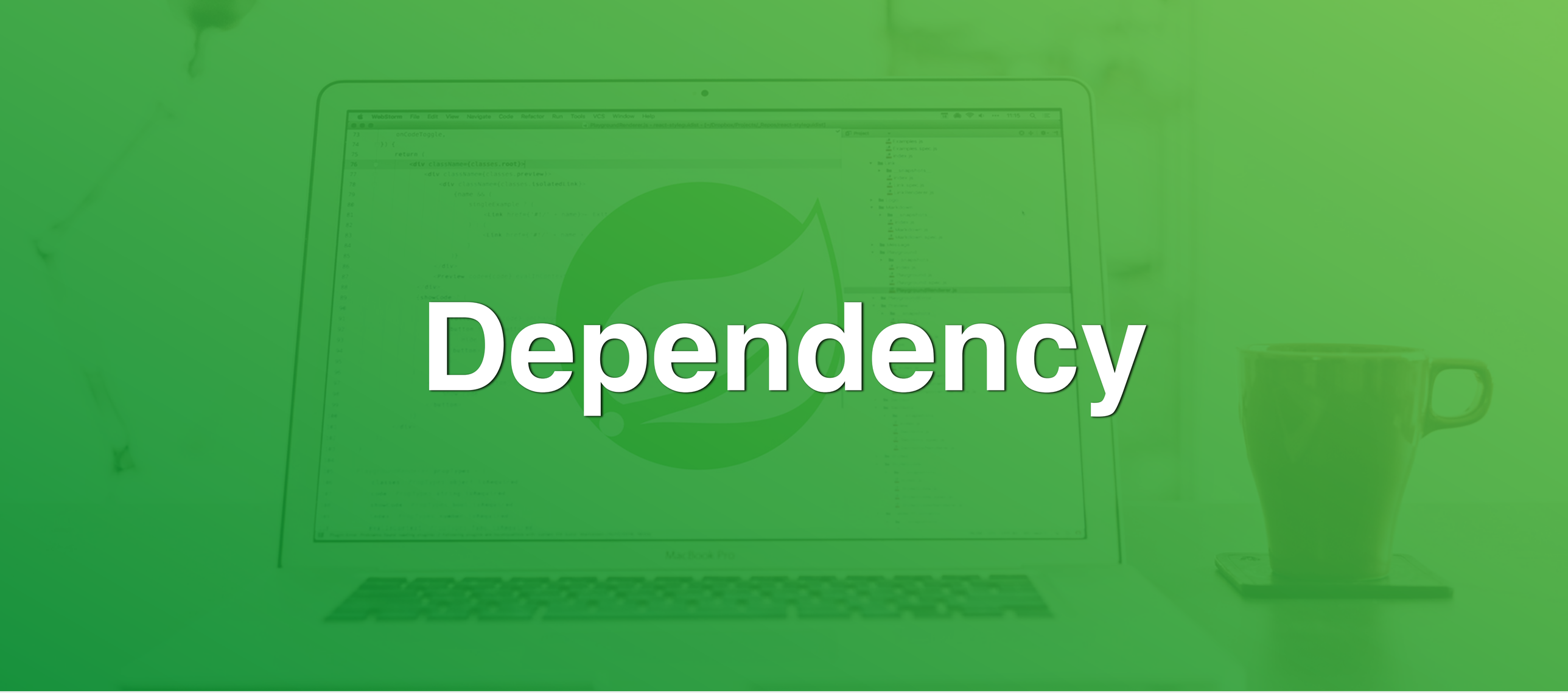
Instructor

Akın Kaldıroğlu

Expert for Agile Software Development and Java

Topics

- Dependency
- Dependency Injection
- Examples



Dependency

selsoft

build better, deliver faster

Coupling and Dependency - I

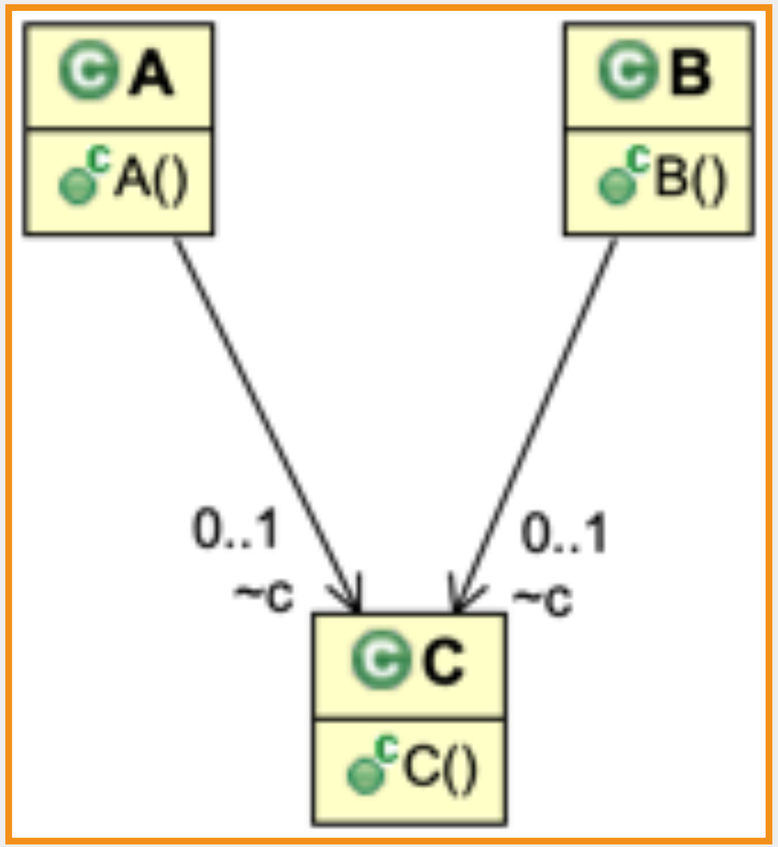
- In terms of relationships among objects, coupling and dependency look similar but they have some differences.
  - They are both translated into Turkish as **bağımlılık**.
- **Coupling** is a name for a general relationship among objects while **dependency** is a more specific relationship among two objects where modification on one object may require modification on other.
- On the other hand coupling is the degree of dependency among two objects; for example lesser dependency means lower coupling.

Coupling and Dependency - II

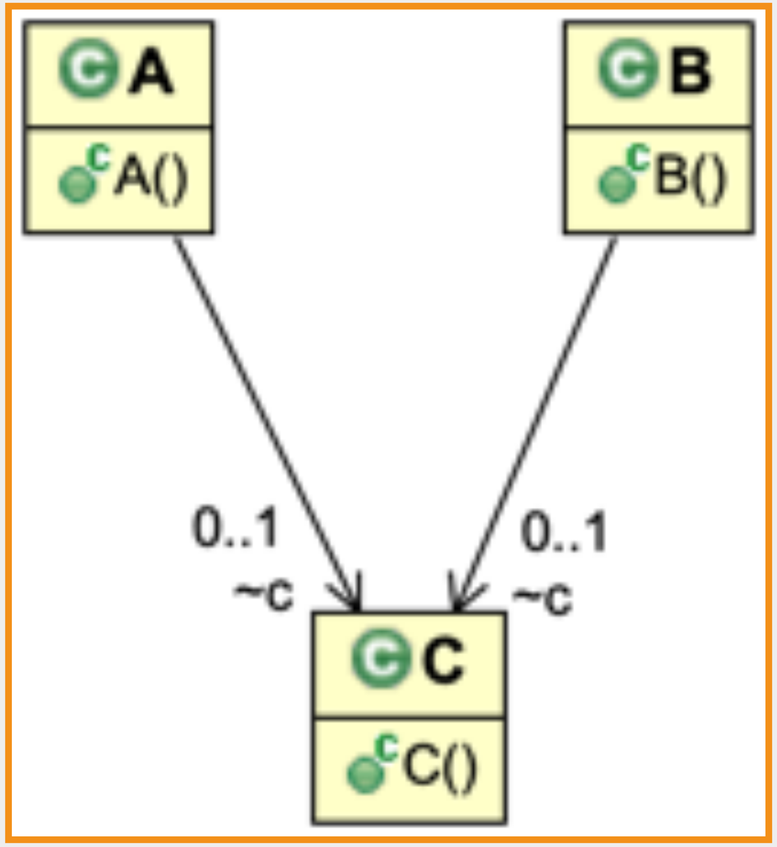
- So dependency is a direct coupling among two objects.
- In this sense in Turkish we can say dependency is bağımlılık but coupling is bağımlılığın derecesi.
- For example, many objects may have a coupling among themselves just because they use the same collection, service or library.
  - This does not create a dependency among those objects but they all have dependency on the shared element.

Example

- In the example the objects of **A** and **B** use an object of **C** so type **A** and **B** have a dependency on type **C**.
  - Or it is said that **A** (or **B**) depends on **C** or **A** (or **B**) is dependent on **C**.
- **A** and **B** are clients of service **C**.
- In terms of coupling it is between **A** and **B** through **C**.



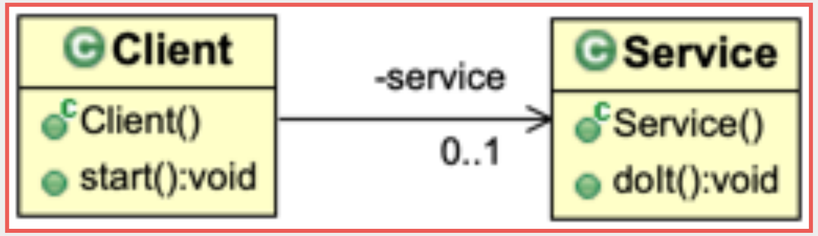
Example



- Due to the dependency, a change in **`C`** may require a change in **`A`** and **`B`**
  - But a a change in **`A`** doesn't require a change in **`B`** or vice versa.

Dependency - I

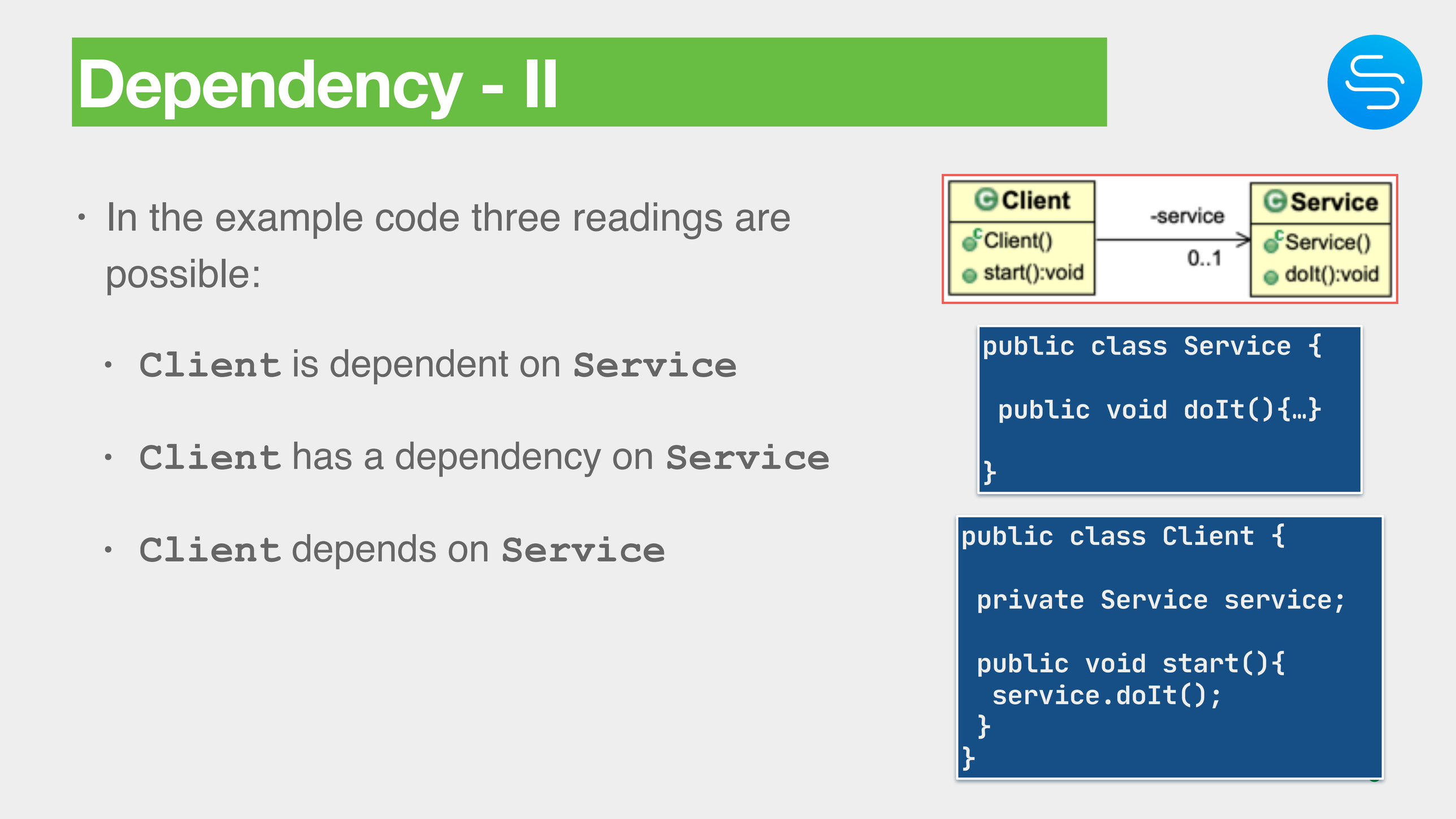
- Dependency defines a relationship between a **service** (or **supplier**) and a **client** (or **source**) where the service provides or supplies services while the client consumes them.
- The client is the dependent end and the service is the independent end.
- The client needs the service in order to operate.




```
public class Service {

 public void doIt(){…}

}
```

```
public class Client {

 private Service service;

 public void start(){
  service.doIt();
 }
}
```

Dependency - II

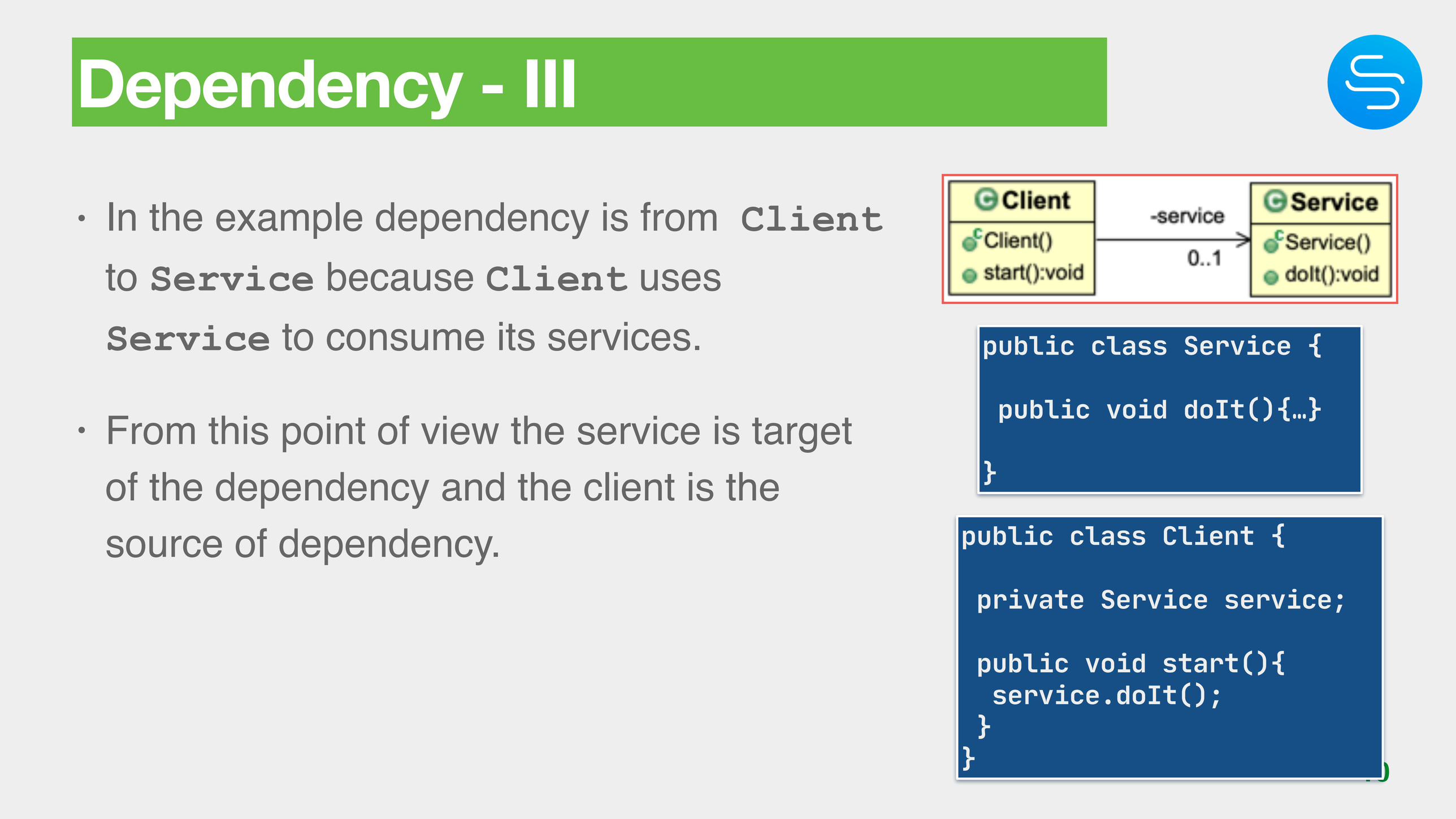


- In the example code three readings are possible:
  - **`Client`** is dependent on **`Service`**
  - **`Client`** has a dependency on **`Service`**
  - **`Client`** depends on **`Service`**


```
public class Service {

 public void doIt(){…}

}
```

```
public class Client {

 private Service service;

 public void start(){
  service.doIt();
 }
}
```

Dependency - III



- In the example dependency is from `Client` to `Service` because `Client` uses `Service` to consume its services.
- From this point of view the service is target of the dependency and the client is the source of dependency.


```
public class Service {

 public void doIt(){…}

}
```

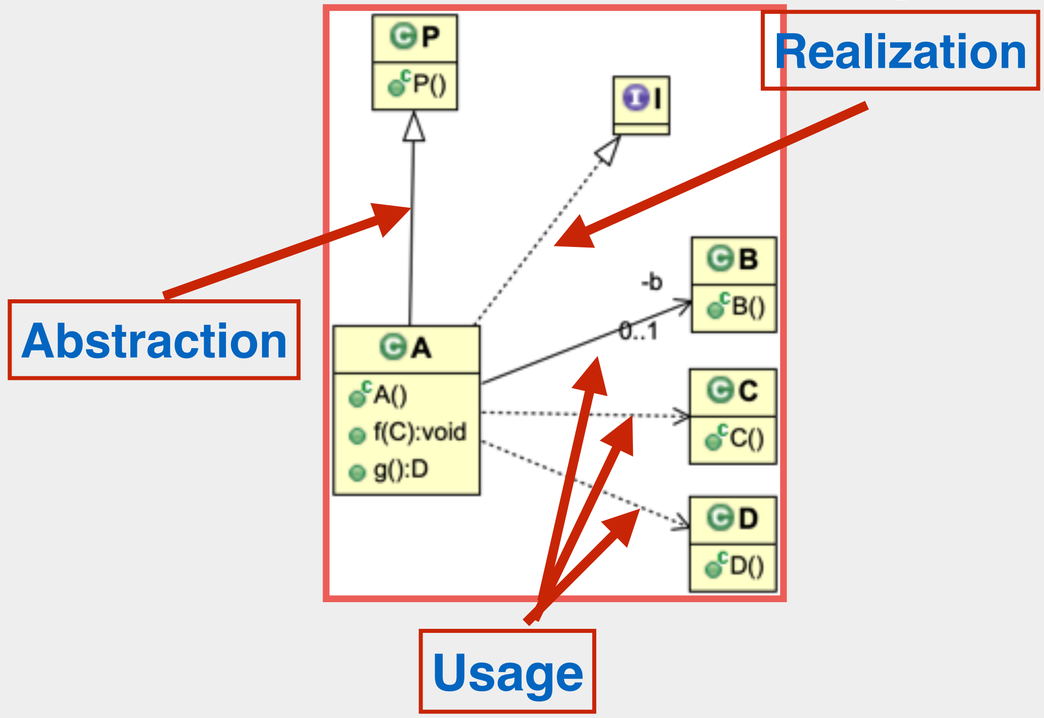
```
public class Client {

 private Service service;

 public void start(){
  service.doIt();
 }
}
```

Dependency Types

- There are mainly three types of dependency:
  - **Abstraction**: The client and the supplier represent the same concept at different levels of abstraction or from different viewpoints.
  - **Realization**: Realization is a specialized Abstraction dependency between two sets of objects, one representing a specification (the supplier) and the other representing an implementation of that specification (the client).
  - **Usage**: The client requires the service for its full implementation or operation.

Example

- **A** inherits from **P** and **I**,
  - Class **A** extends class **P** and implements interface **I**.
- The object of **A** owns an object of **B**,
- The object of **A** receives an object of **C** as a parameter to its method **f()**,
- The object of **A** creates and returns an object of **D**.




```
public class A extends P implements I{
  private B b;

  public void f(C c) {…}

  public D g() { return new D();}
}
```

Abstraction and Realization - I

- Abstraction and realization dependencies are commonly called **is-a** relationship.
- The mechanism to implement **is-a** dependency is called **inheritance**, **sub-typing** or **sub-classing**.
- In abstraction the client or the child provides more specific version of the supplier i.e. the parent while the parent represents more generic version.
- In realization the client or the child provides a concrete implementation to the behavior specified abstractly in the parent.

Abstraction and Realization - II

- Abstraction is called **implementation inheritance** while realization is called **interface inheritance**.

Usage Dependency - I

- In usage dependency the client is incomplete without the service so the client needs the service to operate.
- But this need can be either for definition or implementation of the client.
  - For example the client needs the service only for its implementation of a method such that the client creates the service then returns it.

```
public class A extends P implements I{
  private B b;

  public D g() { return new D();}
}
```

Usage Dependency - II

- If the client needs the service as its part i.e. as an instance variable then the dependency is for definition.

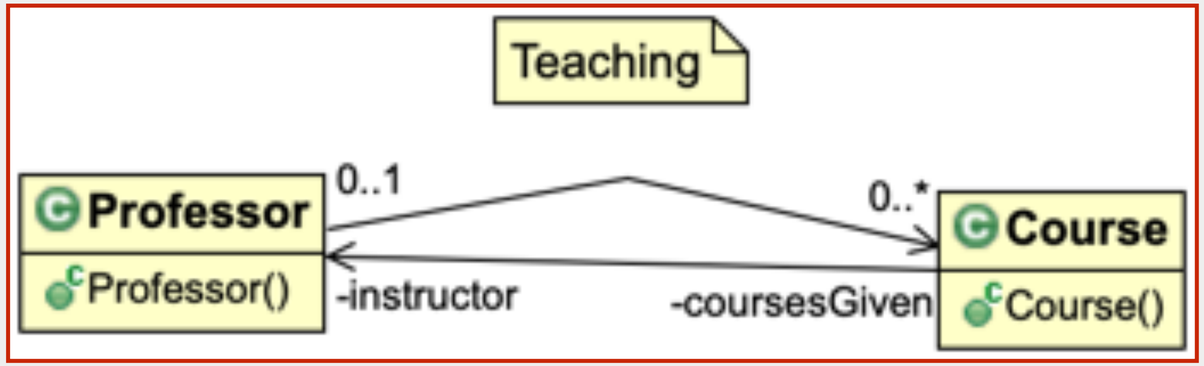
```
public class A extends P implements I{
  private B b;

  public void f(C c) { b.u(c); }

}
```

Usage Dependency - III

- This is a view point from the implementation or detail perspective of the client.
- From the interface perspective we can not differentiate if a dependency is either for definition or implementation.

Association - I

- The dependency for definition is also called **association** (**ilişki**).
- In association there is a notion of **ownership** where the client owns the service (or the client objects own the service objects) or the service is a part of the definition of the client.

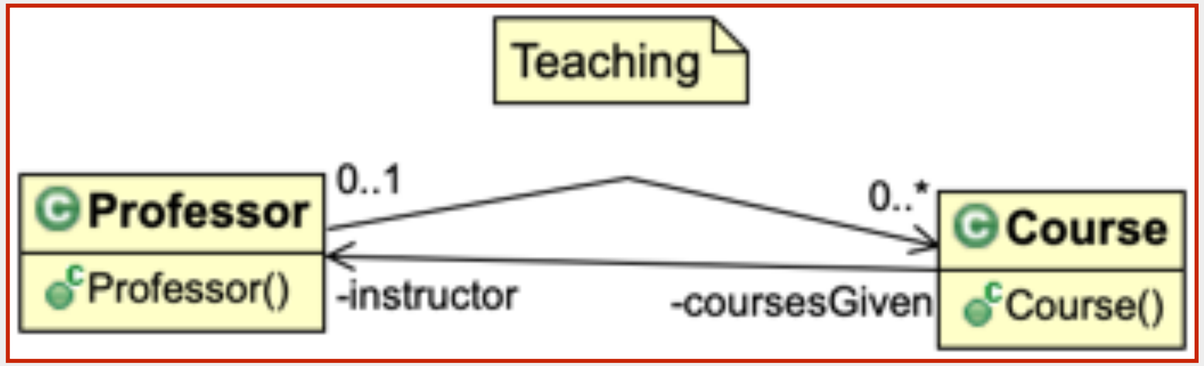



```
public class Professor {

 private Course[] coursesGiven;

}
```

```
public class Course {

 private Professor instructor;

}
```

Association - II

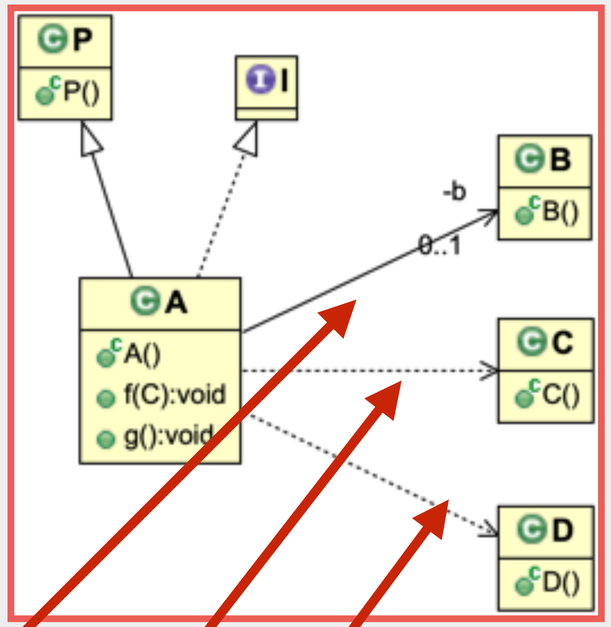
- Association as a relationship among the obejcts of two classes has four properties:
  - association name
  - role names of both sides
  - multiplicity (optional, mandatory, single-value, multivalued, 1-1, 1-M, M-N)
  - navigability (uni-directional, bi-directional)



- Association is also called **has-a** relationship or dependency.

Example

- Here are examples of usage dependencies:
  - The object of **A** owns an object of **B**,
  - The object of **A** receives an object of **C** as parameter to its method **f()**,
  - The object of **A** creates an object of **D**.
- In the first one there is an association or has-a dependency between **A** and **B** but for other two it is just a dependency from **A** to **B**.




```
public class A extends P implements I{
  private B b;

  public void f(C c) {…}

  public D g() { return new D();}
}
```

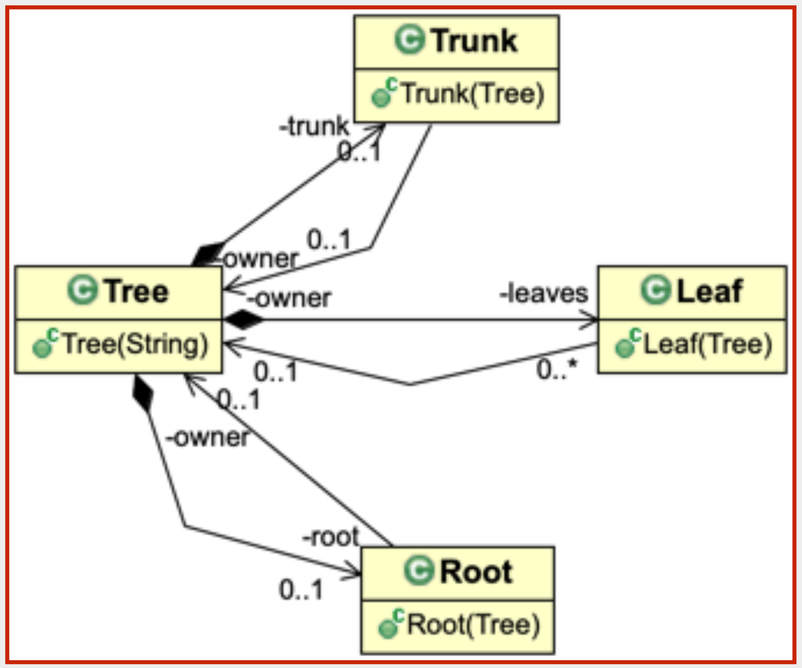
Association Types

- There are two forms of association where the semantic of the relationship changes in terms of ownership:
  - **Aggregation**: In aggregation an object is used to group other objects.
  - **Composition**: Composition is a strong form of aggregation that requires a part object be included in at most one composite object at a time.
    - If a composite object is deleted, all of its part instances that are objects are deleted with it.

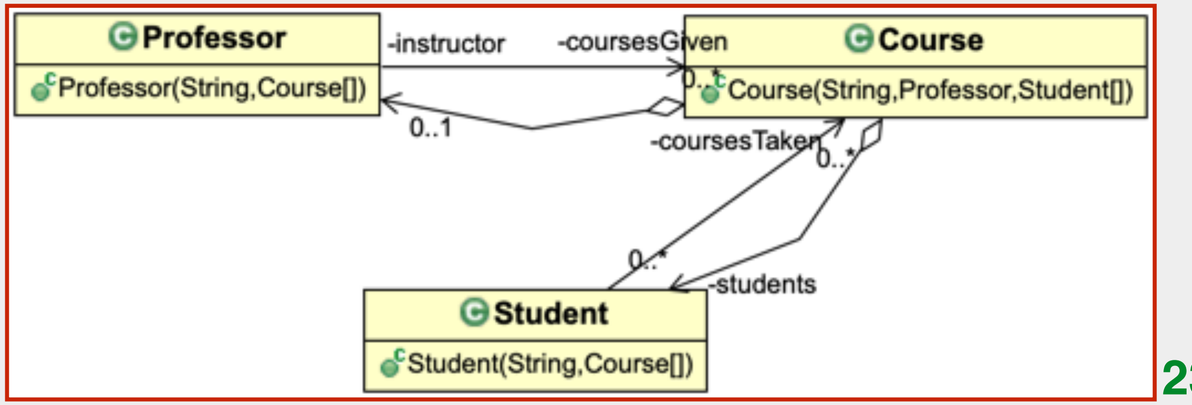
Aggregation vs. Composition - I

- In aggregation the relationship among the objects are somehow looser than in composition.
- Aggregation is a part-of relationship whereas composition emphasizes exclusive ownership.
  - Computer is an aggregation of CPU, RAM, Disk, Display, etc. for example.
  - University is a composite of schools for example so schools don’t exist if the university does not exist.

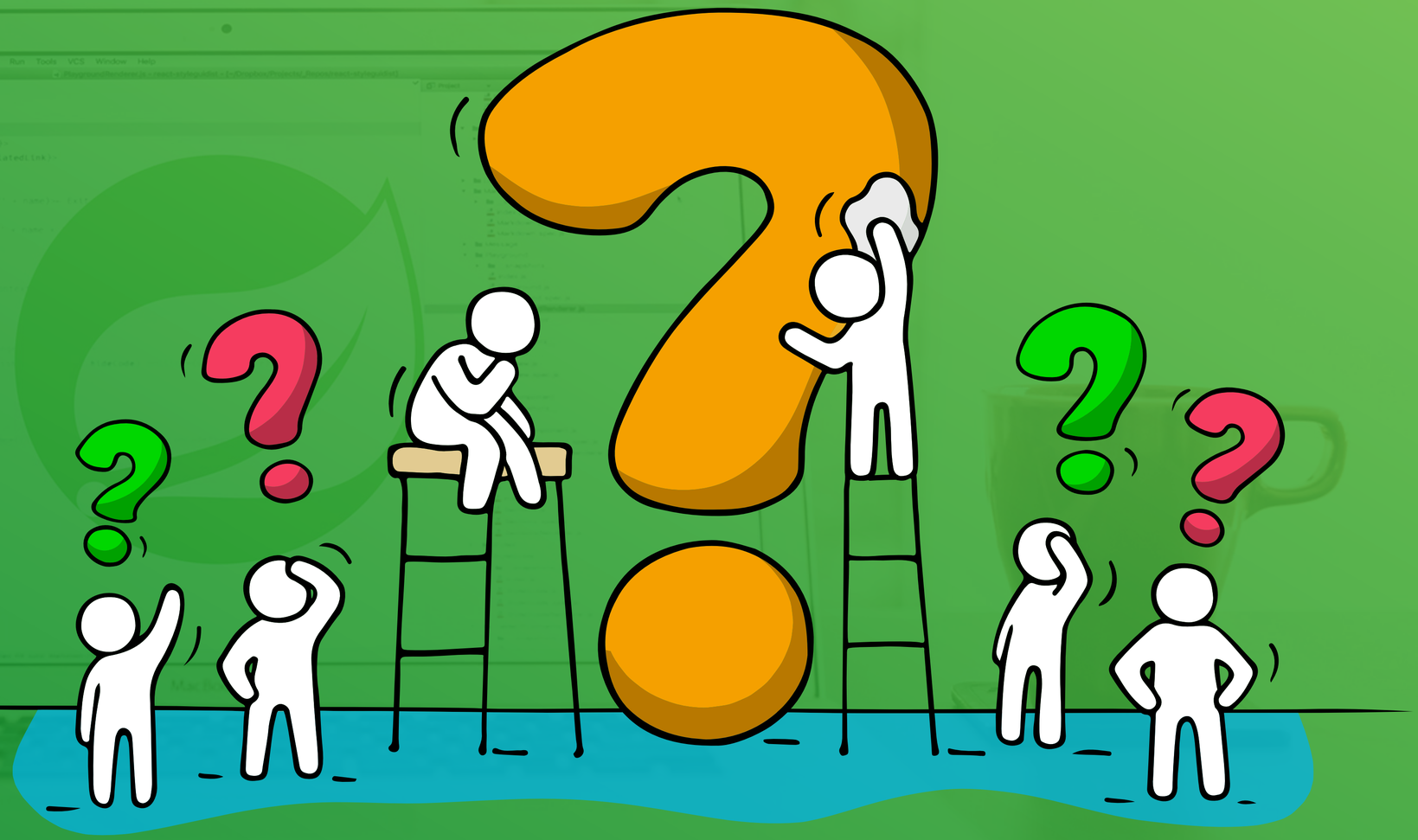
Aggregation vs. Composition - II



- A tree is a composition of its root, trunk and leaves.
- But the relationship among courses and professors and students is somehow looser in terms of ownership so it is more like an aggregation.



Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr

Consequences of Dependency

Consequences of Dependency

- Dependency causes complexity.
- Dependency necessiates understanding and changing together.
- Dependency prevents understanding and changing dependent object in isolation.
- That’s because of the fact that the semantics of the dependent object is incomplete without the depended object.
- And a change to a depended object may cause changes to depending ones.

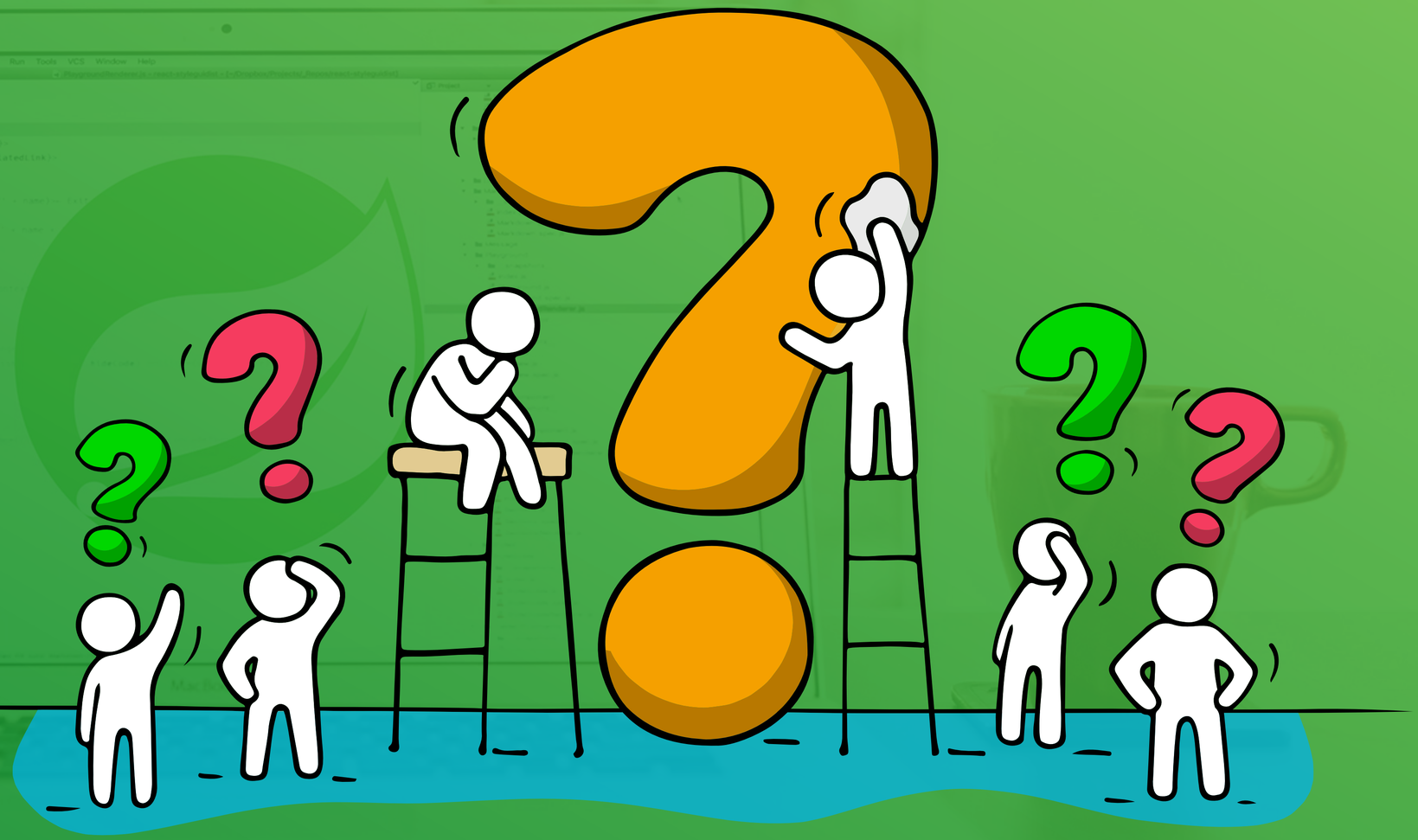
Dependency is Unavoidable

- On the other hand dependency is a must; to be part of a system objects must depend on each other in numerous ways:
  - Some objects must be part of other objects,
  - Some objects must create others,
  - Objects must call methods on each other by passing some other objects,
  - Some objects must throw exceptions while others must catch them.
  - etc.

So What?

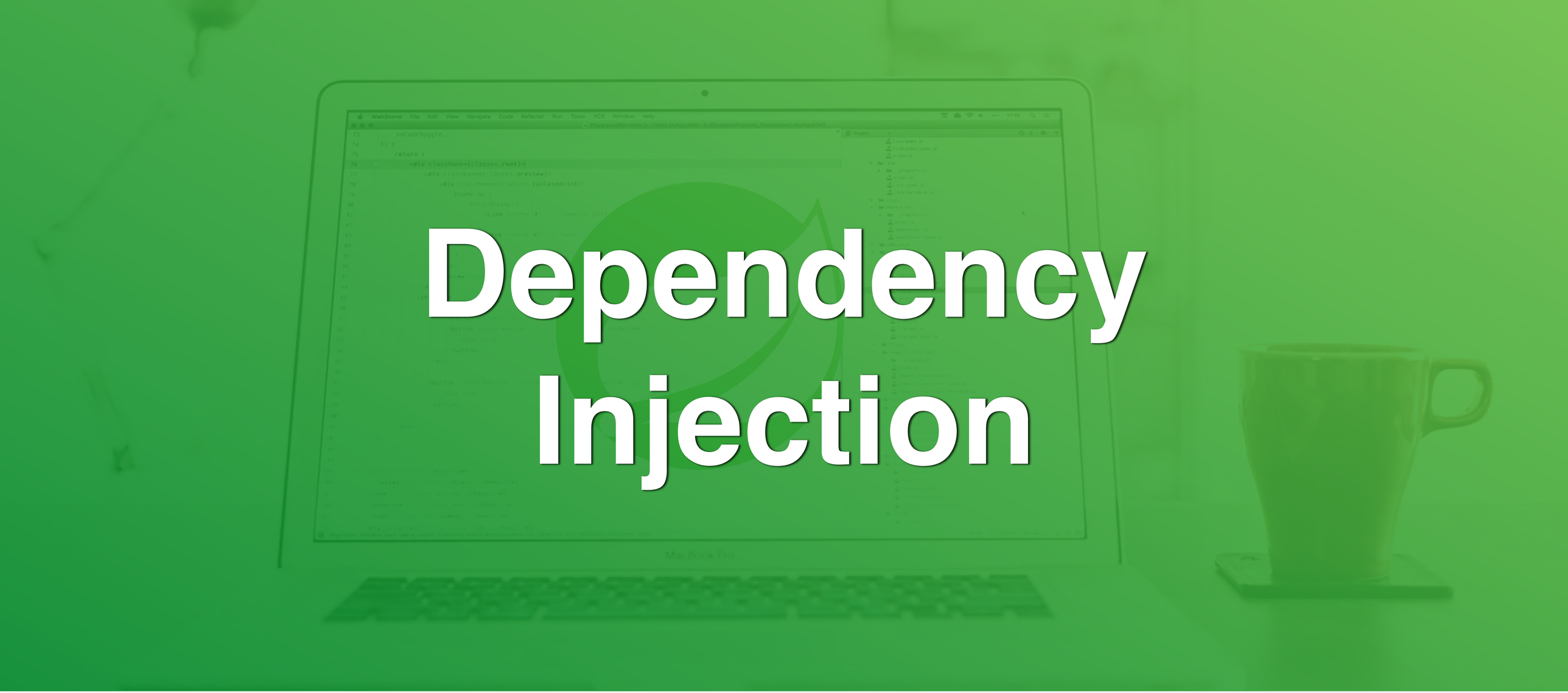
- So the important thing is about having healthy dependencies and managing and maintaining them easily.
- For a more specific discussion on coupling and dependency and how to handle them in different scenarios you can consult with our **Clean Code** and **Design Patterns** courses.

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr

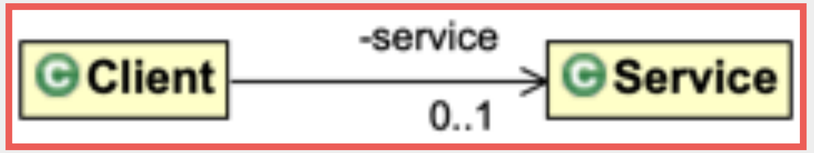


Dependency Injection

selsoft
build better, deliver faster

Managing Dependency - I

- Think about the example code where **`Client`** is dependent on **`Service`** (or **`Client`** has dependency on **`Service`**).
- There may be several different scenarios regarding how a **`Client`** object reaches a **`Service`** object.

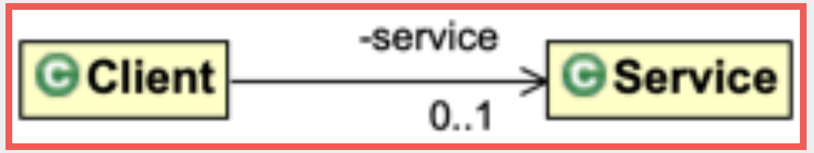



```
public class Service {

}
```

```
public class Client {

  private Service service;

}
```

Managing Dependency - II

- **`Client`** object creates **`Service`** object, which points to a highly-coupled association.
- This kind of dependency causes high-coupling beause of the fact that the existence of the **`Service`** object totally depends on the existence of the **`Client`** object.
- **`Service`** object is solely bounded to **`Client`** object in terms of its life-cycle, a **`Service`** object lives as long as its owner **`Client`** object lives.

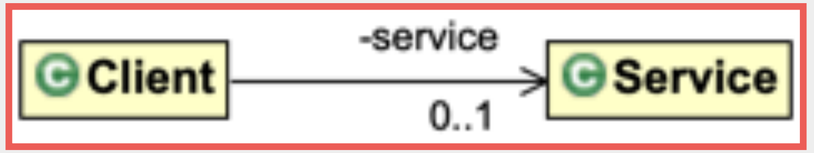



```
public class Service {

}
```

```
public class Client {

  private Service service;

  public Client(){
    service = new Service();
  }
}
```

Managing Dependency - III

- It is a composition in terms of structure.
- But creating a **`Service`** object is a burden on **`Client`** object.
- Being a composition should not put the responsibility of creating parts on the composite.
- Most of the time creating an object is much more difficult than using it!




```
public class Service {

}
```

```
public class Client {

  private Service service;

  public Client(){
    service = new Service();
  }
}
```

Managing Dependency - IV

- Solution might be creating a **`Service`** object and passing it to the **`Client`** object.


```
public class Service {

}
```

- If the **`Client`** object does not create the **`Service`** object but the **`Client`** object receives the **`Service`** object, it points to a looser coupling in association.

```
public class Client {

  private Service service;

  public Client(Service service){
    this.service = service;
  }

  public void setService(Service service){
    this.service = service;
  }
}
```

Managing Dependency - V

- In this case the **`Service`** object may receive the **`Client`** object in different ways:
  - The **`Client`** object can be passed to the **`Service`** object through its corresponding field (which is discouraged due to information hiding), constructor and method, most probaby a setter method.


```
public class Service {

}
```

```
public class Client {

  private Service service;

  public Client(Service service){
    this.service = service;
  }

  public void setService(Service service){
    this.service = service;
  }
}
```

Managing Dependency - VI

- Creating the **`Service`** object somewhere else and passing it to the **`Client`** object is a much better solution in terms of coupling and complexity of **`Client`**.
- The **`Client`** still depends on the **`Service`** but the coupling, degree of dependency is lower.


```
public class Service {

}
```

```
public class Client {

  private Service service;

  public Client(Service service){
    this.service = service;
  }

  public void setService(Service service){
    this.service = service;
  }
}
```

Dependency Injection - I

- Creating a depended object somewhere else and then passing it to the depending object is called **dependency injection**.
- In dependency injection the client object does not have the responsibility to create its parts or its service objects.
- So dependency injection removes the responsibility of creating its parts from the client.
- The client object should only know how to use its parts.

Dependency Injection - II

- The client object should only know how to use its parts, so the coupling level remains at interface.
- But if a client knows how to create a service object it uses the coupling level is implementation.
- Creating an object requires more information than using the same object.

Program to an interface not an implementation.

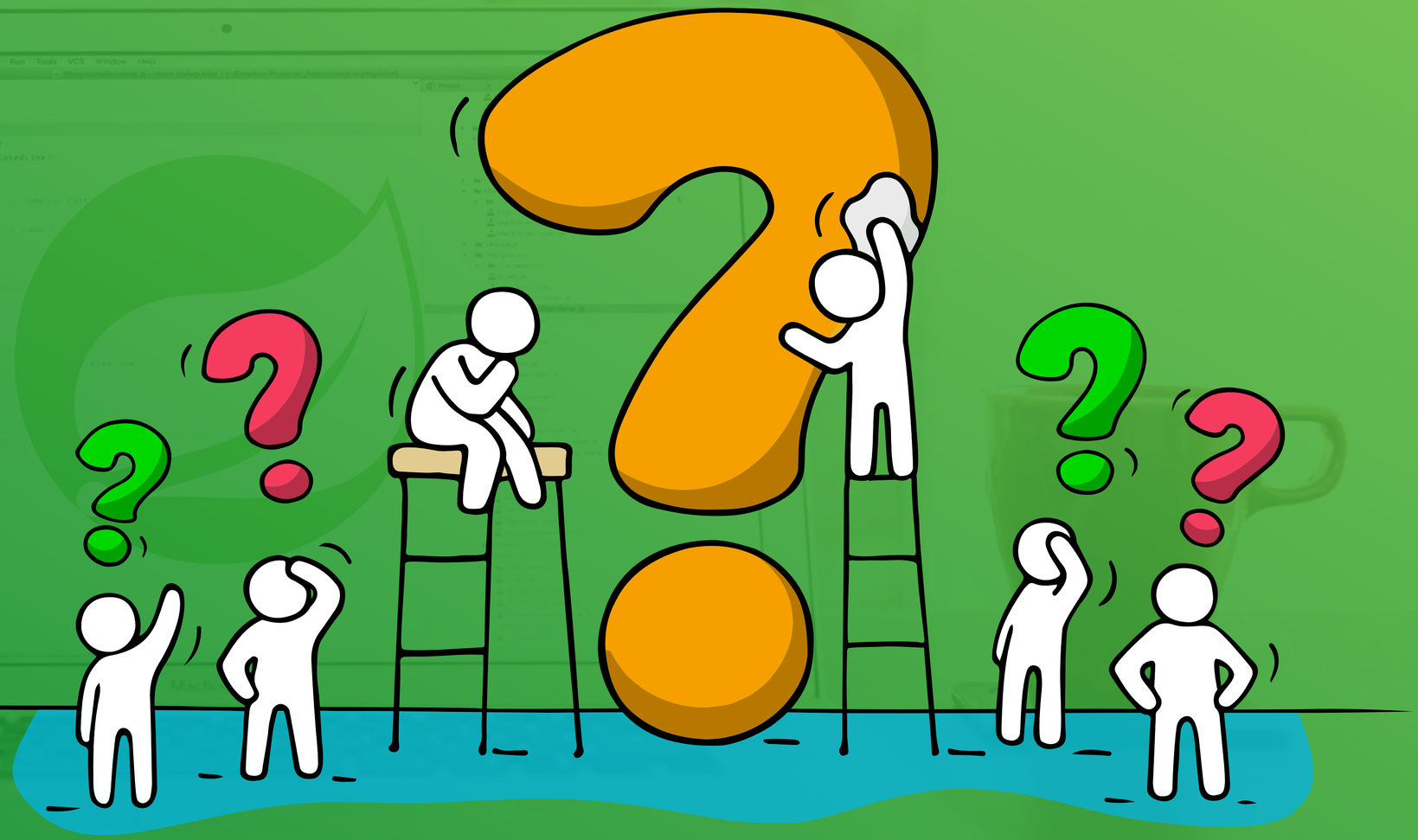
Dependency Injection - III

- By removing the responsibility of the creation of the service objects dependency injection allows the client objects to be highly-cohesive and lowly-coupled.
- Dependency injection allows the client only know the interface of the service objects it uses not their implementation.

Dependency Injection - IV

- Now the question becomes “who creates the parts?”

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr

Examples

selsoft

build better, deliver faster

Direction

- In this section we deal with a very basic and well-known program, **`Hello World`**, and create its different versions with the purpose of having highly cohesive and lowly-coupled programs.
- Throughout these versions we observe the need for dependency injection and how a framework handling all dependency issues will help us have such a program.

Classical Hello World :)

greeting01- I

- What is wrong with this code?
- Think about dependencies **`Application`** has.
- What kinds of changes force **`Application`** to change?

```
public class Application {

    public static void main(String[] args) {
        System.out.println(“Hello world :)”);
    }
}
```

greeting01- II

- First and foremost factor that would force **`Application`** to change is the message itself, which is a **`String`** object.
  - What if we want to say greeting in another language like “Selam”?
  - **`Application`** must be changed for each different greeting.
  - So let’s refactor it to accomodate such a need.

```
public class Application {

    public static void main(String[] args) {
        System.out.println(“Hello world :)”);
    }
}
```

Better
Hello World :)

greeting02 - I

- **`Application`** is better off now in terms of allowing different messages without a change.
- This is achieved by providing a **`String`** message object to the **`main`** method.

```
public class Application {

    public static void main(String[] args) {
        // If an argument is provided, use it, otherwise, display “Selam”
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello world :)");
        }
    }
}
```

greeting02 - II

- How about other dependencies **`Application`** has.
- What kinds of changes force **`Application`** to change?

```
public class Application {

    public static void main(String[] args) {
        // If an argument is provided, use it, otherwise, display “Selam”
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello world :)");
        }
    }
}
```

greeting02 - III

- `System.out.println` is the only way for `Application` to greet.
- What if `Application` wants to write greeting somewhere else such as a file or a web service?

```
public class Application {

    public static void main(String[] args) {
        // If an argument is provided, use it, otherwise, display “Selam”
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello world :)");
        }
    }
}
```

greeting02 - IV

- In fact `System.out.println` does two things:
  - Providing the message and
  - Rendering the message.

```
public class Application {

    public static void main(String[] args) {
        // If an argument is provided, use it, otherwise, display “Selam”
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello world :)");
        }
    }
}
```

greeting02 - V

- Providing and rendering the message are two different responsibilities and putting them together makes **`Application`**'s dependency on **`System.out.println`** two-fold, so let's separate them.

```
public class Application {

    public static void main(String[] args) {
        // If an argument is provided, use it, otherwise, display “Selam”
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello world :)");
        }
    }
}
```

Seperating Responsibilities

greeting03 - I

- Responsibilities are separated: **`HelloWorldGreetingProvider`** is responsible for providing the greeting while **`StandardOutputRenderer`** is responsible for rendering the greeting.

```
public class Application {
  public static void main(String[] args) {
    // Create renderer
    StandardOutputRenderer renderer = new StandardOutputRenderer();
    // Create provider
    HelloWorldGreetingProvider provider = new HelloWorldGreetingProvider();
    // Set the provider to the renderer
    renderer.setGreetingProvider(provider);
    // Call renderer
    renderer.render();
  }
}
```

greeting03 - II

- The object of **`HelloWorldGreetingProvider`** is passed to the object of **`StandardOutputRenderer`**.

- **`StandardOutputRenderer`** renders what is provided by **`HelloWorldGreetingProvider`**.

```
public class HelloWorldGreetingProvider{

  public String getGreeting() {
     return "Hello World :)";
  }
}
```

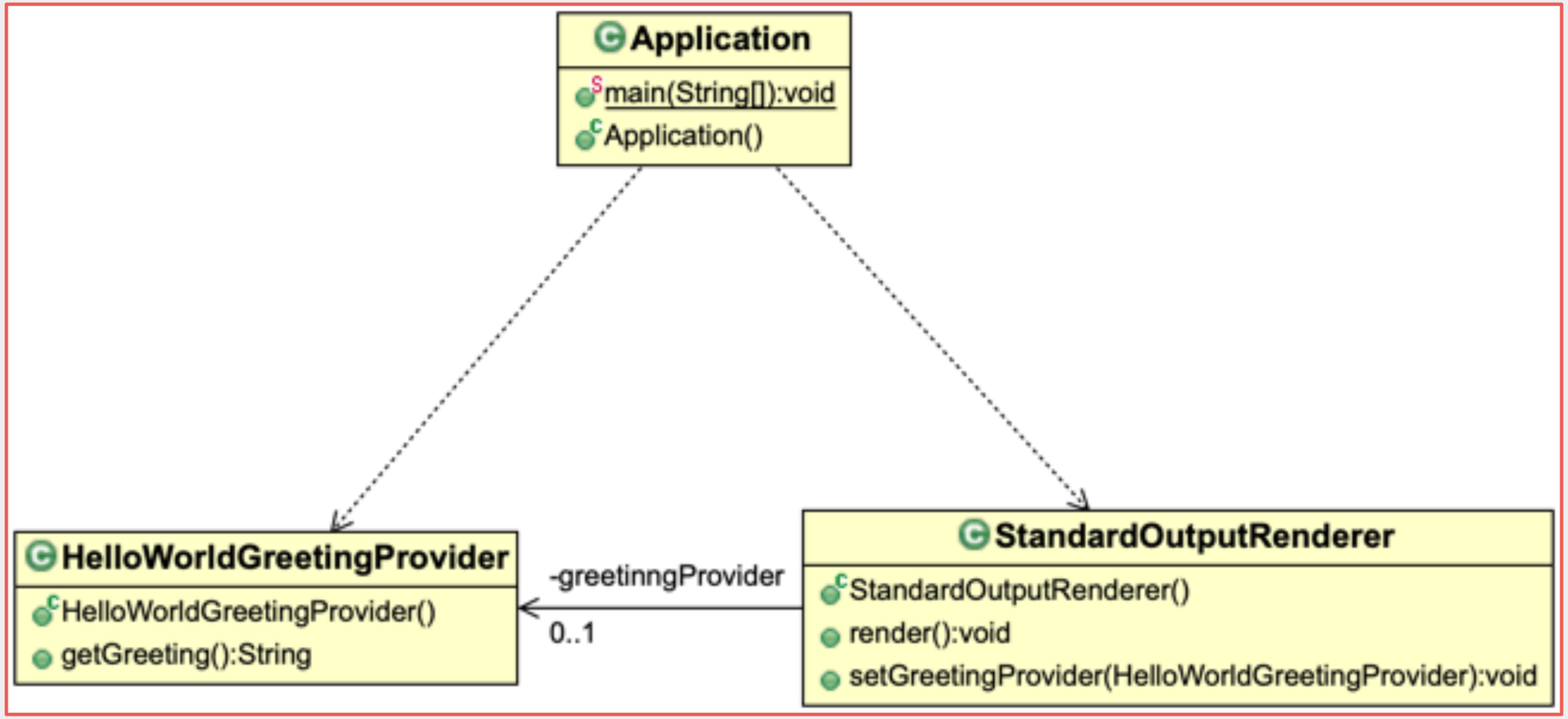
```
public class StandardOutputRenderer {
  private HelloWorldGreetingProvider greetingProvider = null;

  public void setGreetingProvider(HelloWorldGreetingProvider provider){
     this.greetingProvider = provider;
  }

  public void render() {
     String greeting = greetingProvider.getGreeting();
     System.out.println(greeting);
  }
}
```

greeting03 - III

- What is wrong with this code? Or what else can be improved in this code?
- Think about the dependencies again!

```
public class Application {
  public static void main(String[] args) {
    // Create renderer
    StandardOutputRenderer renderer = new StandardOutputRenderer();
    // Create provider
    HelloWorldGreetingProvider provider = new HelloWorldGreetingProvider();
    // Set the provider to the renderer
    renderer.setGreetingProvider(provider);
    // Call renderer
    renderer.render();
  }
}
```

greeting03 - IV



greeting03 - V

- Dependencies are on concrete classes.

  Do not depend on concretions, depend on abstractions.

- So all dependencies should be inverted (**Dependency Inversion**).
  - All classes should be an implementation of interfaces and
  - **`Application`** should only know interfaces not classes.

Depend on Abstractions

greeting04 - I

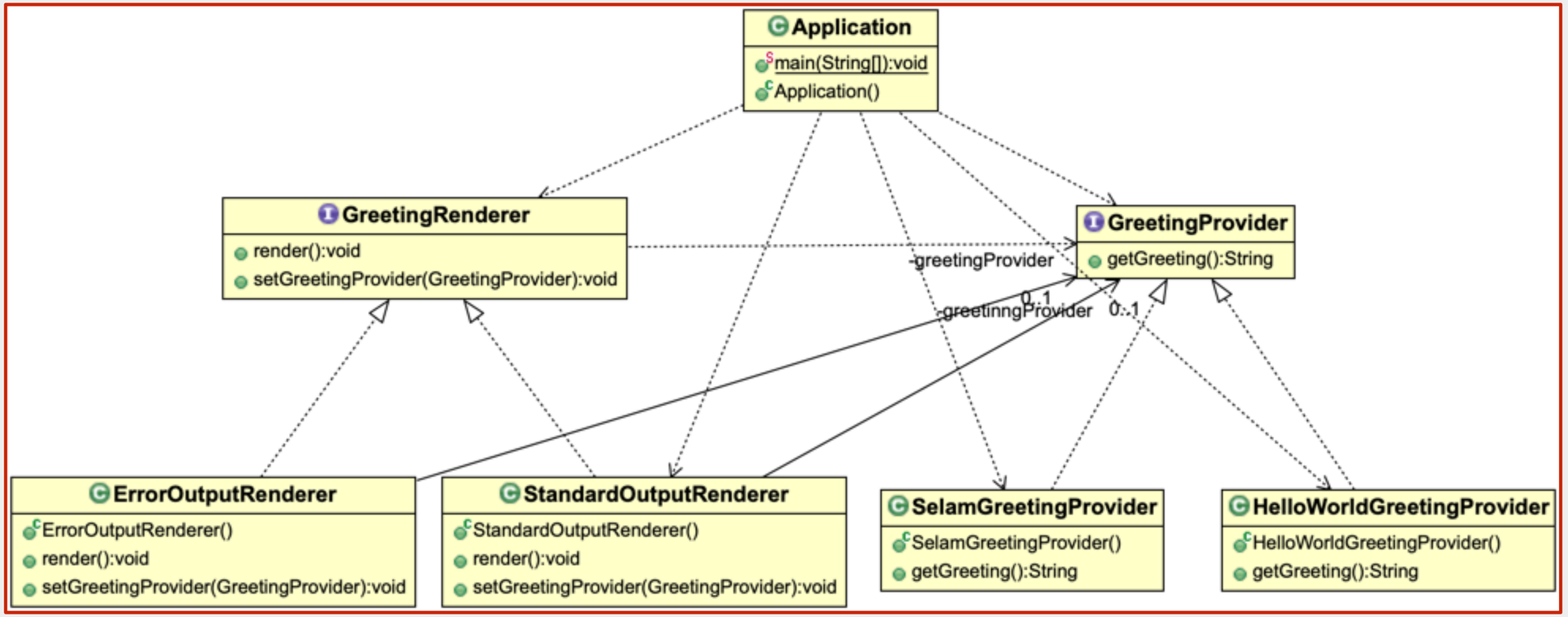
- All dependencies are inverted so all classes only depend on abstractions.
  - Well, not exactly!

```
public interface GreetingRenderer {
 public void render();
 public void setGreetingProvider(GreetingProvider provider);
}
```

```
public interface GreetingProvider {
  public String getGreeting();
}
```

```
public class Application {

  public static void main(String[] args) {
    GreetingRenderer renderer = new StandardOutputRenderer();
    GreetingProvider helloGreetingProvider = new HelloWorldGreetingProvider();
    renderer.setGreetingProvider(helloGreetingProvider);
    renderer.render();
  }
}
```

greeting04 - II



greeting04 - III

```
public class Application {
  public static void main(String[] args) {
     GreetingRenderer renderer = new StandardOutputRenderer();
     GreetingProvider helloGreetingProvider = new HelloWorldGreetingProvider();
     renderer.setGreetingProvider(helloGreetingProvider);
     renderer.render();
  }
}
```

```
public interface GreetingProvider {
  public String getGreeting();
}
```

```
public interface GreetingRenderer {

 public void render();
 public void setGreetingProvider(GreetingProvider provider);
}
```

```
public class HelloWorldGreetingProvider
         implements GreetingProvider{

  public String getGreeting() {
     return "Hello World :)";
  }
}
```

```
public class StandardOutputRenderer implements GreetingRenderer{
  private GreetingProvider greetingProvider;

  public void setGreetingProvider(GreetingProvider provider) {
     this.greetingProvider = provider;
  }

  public void render() {
     String greeting = greetingProvider.getGreeting();
     System.out.println(greeting);
  }
}
```

greeting04 - IV

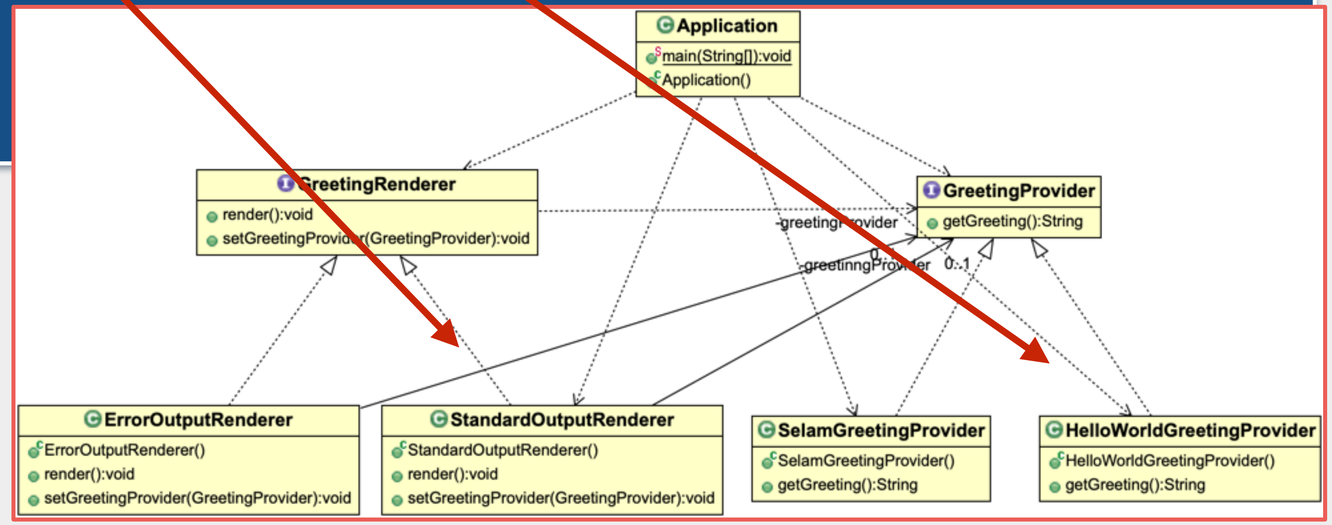
- What is wrong with this code?
- Think about the dependencies again!
- Who creates objects?

```
public class Application {

    public static void main(String[] args) {
        GreetingRenderer renderer = new StandardOutputRenderer();

        GreetingProvider helloGreetingProvider = new HelloWorldGreetingProvider();
        renderer.setGreetingProvider(helloGreetingProvider);
        renderer.render();
    }
}
```

greeting04 - V

- Objects are still created by **`Application`** so it is not true that all classes depend on abstractions.

```
public class Application {
  public static void main(String[] args) {
    GreetingRenderer renderer = new StandardOutputRenderer();

    GreetingProvider helloGreetingProvider = new HelloWorldGreetingProvider();
    renderer.setGreetingProvider(helloGreetingProvider);
    renderer.render();
  }
}
```



Use of Factories

greeting05 - I

- Objects should be created outside of **`Application`** so it only references their parent types.
- For this purpose factories can be used.

```
public class Application {
  public static void main(String[] args) {
    Factory factory = GreetingFactory.getInstance();

    GreetingRenderer renderer = factory.getGreetingRenderer();
    GreetingProvider provider = factory.getGreetingProvider();
    renderer.setGreetingProvider(provider);
    renderer.render();
  }
}
```

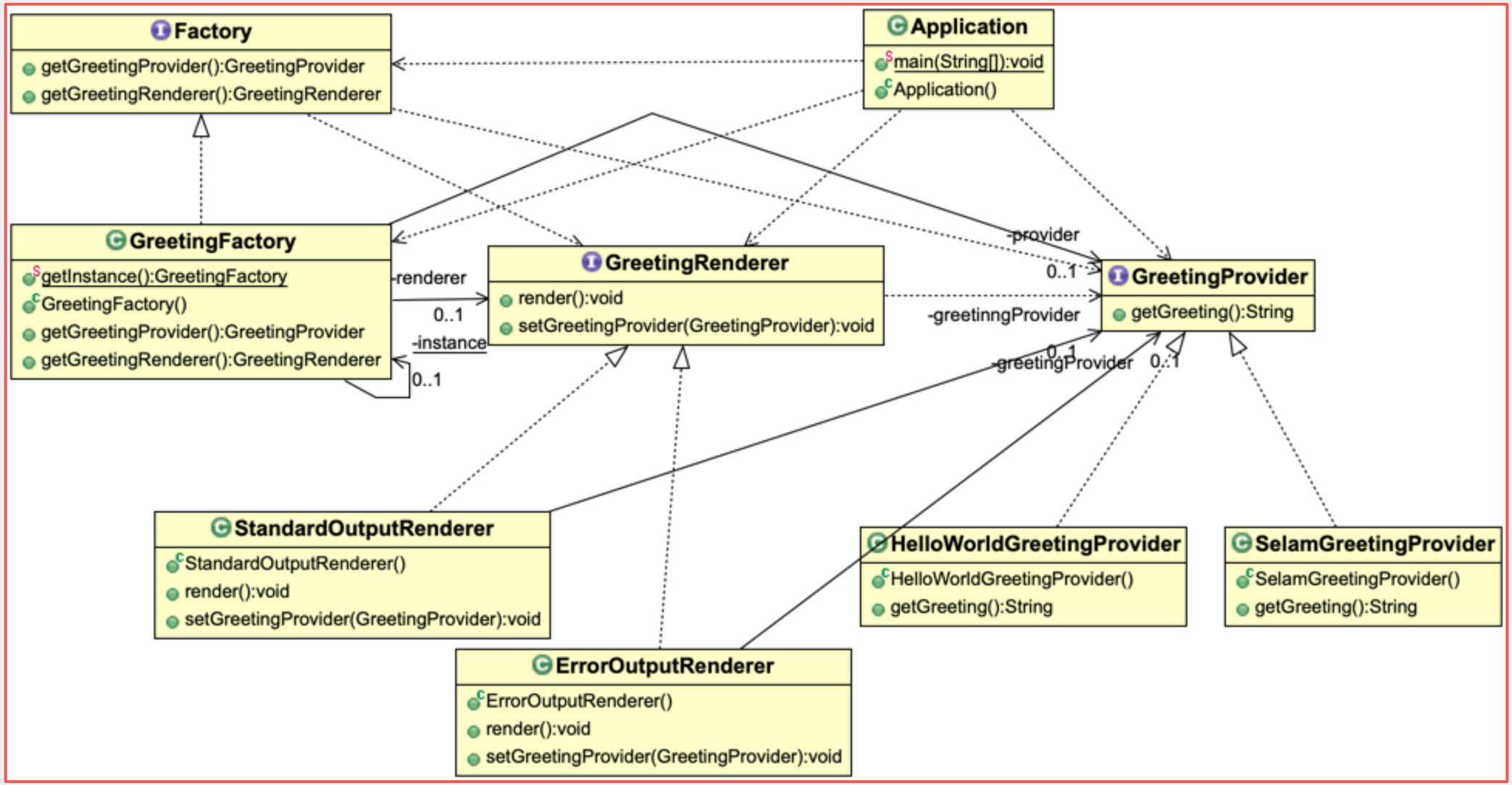
greeting05 - II

- This is an example of **Abstract Factory**.
- You can consult with design patterns book for more information.

```
public interface Factory {

  public GreetingRenderer getGreetingRenderer();
  public GreetingProvider getGreetingProvider();

}
```

```
public class GreetingFactory implements Factory{
     private static GreetingFactory instance;
     private Properties props;

     private GreetingRenderer renderer;
     private GreetingProvider provider;

     { … }

     static { instance = new GreetingFactory(); }

     public static GreetingFactory getInstance(){
          return instance;
     }

     @Override
     public GreetingRenderer getGreetingRenderer(){
          return renderer;
     }

     @Override
     public GreetingProvider getGreetingProvider(){
          return provider;
     }
}
```

greeting05 - III



greeting05 - IV

- What can be improved in this code?
- Think about the dependencies again!
- Who develops and maintains factories?

```
public class Application {
  public static void main(String[] args) {
    Factory factory = GreetingFactory.getInstance();

    GreetingRenderer renderer = factory.getGreetingRenderer();
    GreetingProvider provider = factory.getGreetingProvider();
    renderer.setGreetingProvider(provider);
    renderer.render();
  }
}
```

Framework of Factories

greeting06 - I

- How about if a framework that is responsible for creating all objects is used?
- In this case we don’t have to develop and maintain factories.
  - All objects are created by the framework.

```
public class Application {
  public static void main(String[] args) {
   ObjectProviderFramework framework = new ObjectProviderFramework();

   GreetingRenderer renderer = (GreetingRenderer) framework.getObject(“org….StandardOutputRenderer”);
   GreetingProvider provider = (GreetingProvider) framework.getObject(“org….HelloWorldGreetingProvider”);

   renderer.setGreetingProvider(provider);
   renderer.render();
  }
}
```

greeting06 - II

- What else can be improved in this code?
- Passing objects to each other to fulfill the dependencies is alse called **wiring objects**.
- Who wires the objects?

```
public class Application {
  public static void main(String[] args) {
   ObjectProviderFramework framework = new ObjectProviderFramework();

   GreetingRenderer renderer = (GreetingRenderer) framework.getObject("org….StandardOutputRenderer");
   GreetingProvider provider = (GreetingProvider) framework.getObject("org….HelloWorldGreetingProvider");

   renderer.setGreetingProvider(provider);
   renderer.render();
  }
}
```

greeting06 - III

- `Application` is responsible for wiring the objects.
- Let’s get rid of the responsibility of wiring too!

```
public class Application {
  public static void main(String[] args) {
   ObjectProviderFramework framework = new ObjectProviderFramework();

   GreetingRenderer renderer = (GreetingRenderer) framework.getObject(“org….StandardOutputRenderer”);
   GreetingProvider provider = (GreetingProvider) framework.getObject(“org….HelloWorldGreetingProvider”);

   renderer.setGreetingProvider(provider);
   renderer.render();
  }
}
```

Framework of Objects

greeting07

- How about if a framework that is responsible for creating and wiring objects is used?

- In this case we don’t have to develop and maintain factories and won’t be responsible for wiring objects, all these are managed by the framework itself.

```
public class Application {
  public static void main(String[] args) {
   ObjectProviderFramework framework = new ObjectProviderFramework();

    GreetingRenderer renderer = (GreetingRenderer) framework.getObject(“org.javaturk.spring.ch02.greeting07.StandardOutputRenderer",
    "org.javaturk.spring.ch02.greeting07.HelloWorldGreetingProvider");

    // No need to wiring, the renderer aleady has the provider.
   renderer.render();
  }
}
```

Spring Framework for Objects

greeting08 - 1

- **Spring** does exactly what `ObjectProviderFramework` does but in a different way.

```
public class Application {

  public static void main(String[] args) throws Exception {
      BeanFactory factory = new ClassPathXmlApplicationContext("org/javaturk/spring/di/ch01/greeting08/resources/beans1.xml");
      GreetingRenderer renderer = (GreetingRenderer) factory.getBean("renderer");
      // No need to wiring, the renderer aleady has the provider.
      renderer.render();

      GreetingProvider provider = (GreetingProvider) factory.getBean("provider");
      System.out.println(provider.getGreeting());
  }
}
```

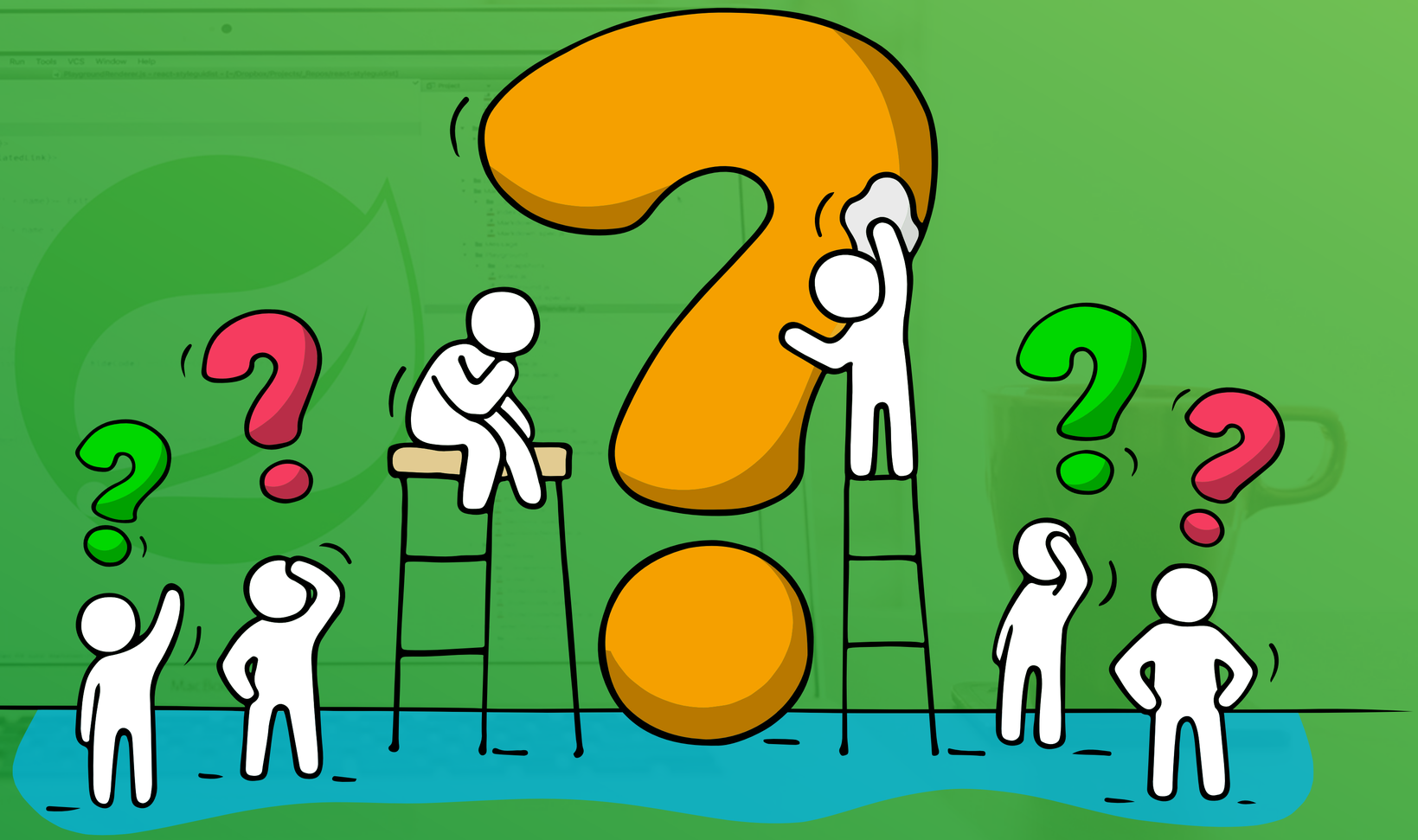
```
public class Application {
  public static void main(String[] args) {
   ObjectProviderFramework framework = new ObjectProviderFramework();

   GreetingRenderer renderer = (GreetingRenderer) framework.getObject("org.javaturk.spring.ch02.greeting07.StandardOutputRenderer",
    "org.javaturk.spring.ch02.greeting07.HelloWorldGreetingProvider");
   // No need to wiring, the renderer aleady has the provider.
   renderer.render();
  }
}
```

Spring As Biggest Factory

- **Spring** is the biggest factory or more correctly a framework to create factories that create and wire objects.
- So **Spring** first creates objects and wires them by passing objects it created to each other to satisfy dependencies among them.

```
public class Application {

  public static void main(String[] args) throws Exception {
     BeanFactory factory = new ClassPathXmlApplicationContext("org/javaturk/spring/di/ch01/greeting08/resources/beans1.xml");
     GreetingRenderer renderer = (GreetingRenderer) factory.getBean(“renderer");
     // No need to wiring, the renderer aleady has the provider.
     renderer.render();

     GreetingProvider provider = (GreetingProvider) factory.getBean("provider");
     System.out.println(provider.getGreeting());
  }
}
```

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr

Homeworks

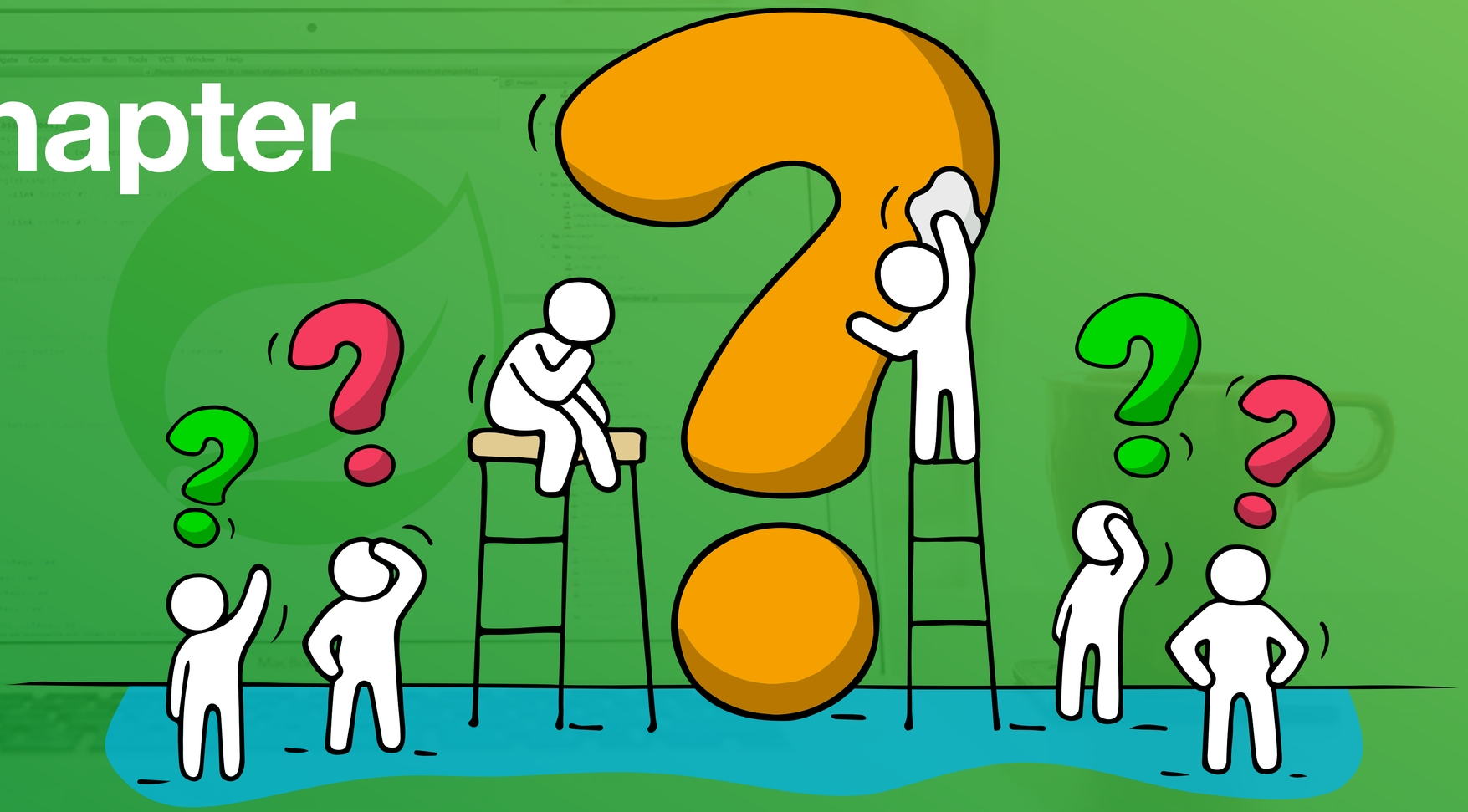
Homework

1. Start reading IoC Container from **Spring Reference Documentation** especially Dependency Injection part: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#spring-core

2. Start reading Chapter 3 from **Pro Spring 5** book.

End of Chapter

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr