Dependency Injection

Containers

We have heard about Java EE containers.
What are these containers?

Definition

--an object that can be used to hold or transport something.

So, What are Java and Java EE Containers?

Normally, thin-client multi-tiered applications are hard to write because they involve many lines of intricate code to handle

- Transaction Management
- State management
- Multithreading
- Resource pooling (Object and DB Connection)
- Security
- and other complex low-level details.

Java EE Component Model

- The component-based and platform-independent Java EE architecture makes Java EE applications easy to write because business logic is organized into reusable components.

- In addition, the Java EE server provides underlying services in the form of a container for every component type.

- Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Non Java EE Application

Business Code

Transaction Code

Security Code

Thread Management

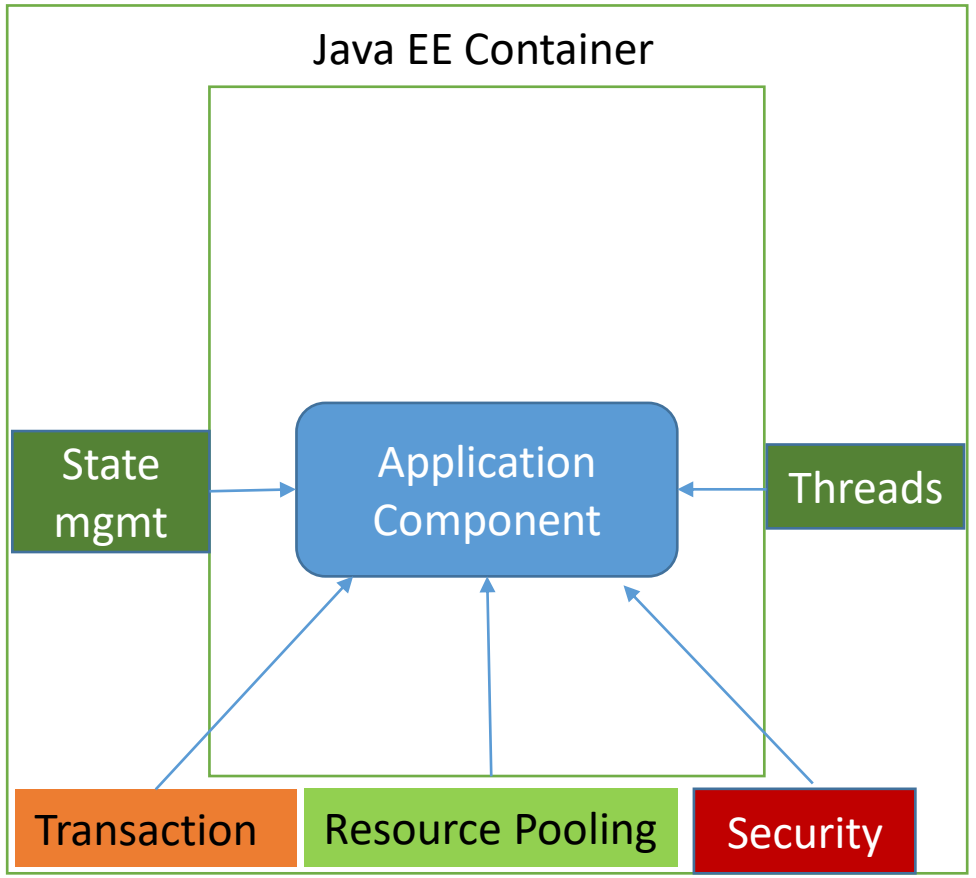
Resource Management

……

……

Not Reusable

Java EE Application

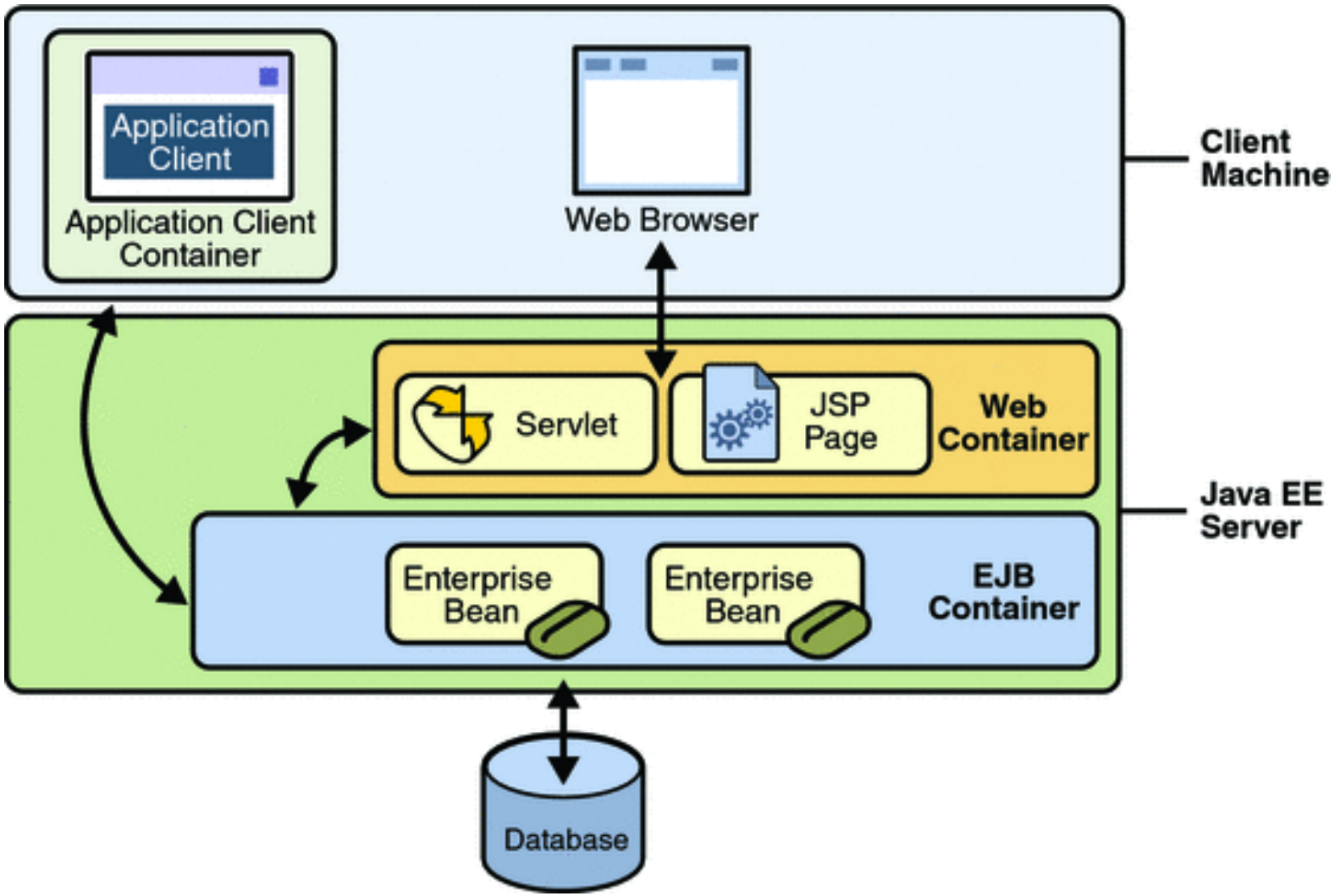


The Component is re usable as well Services are also reusable

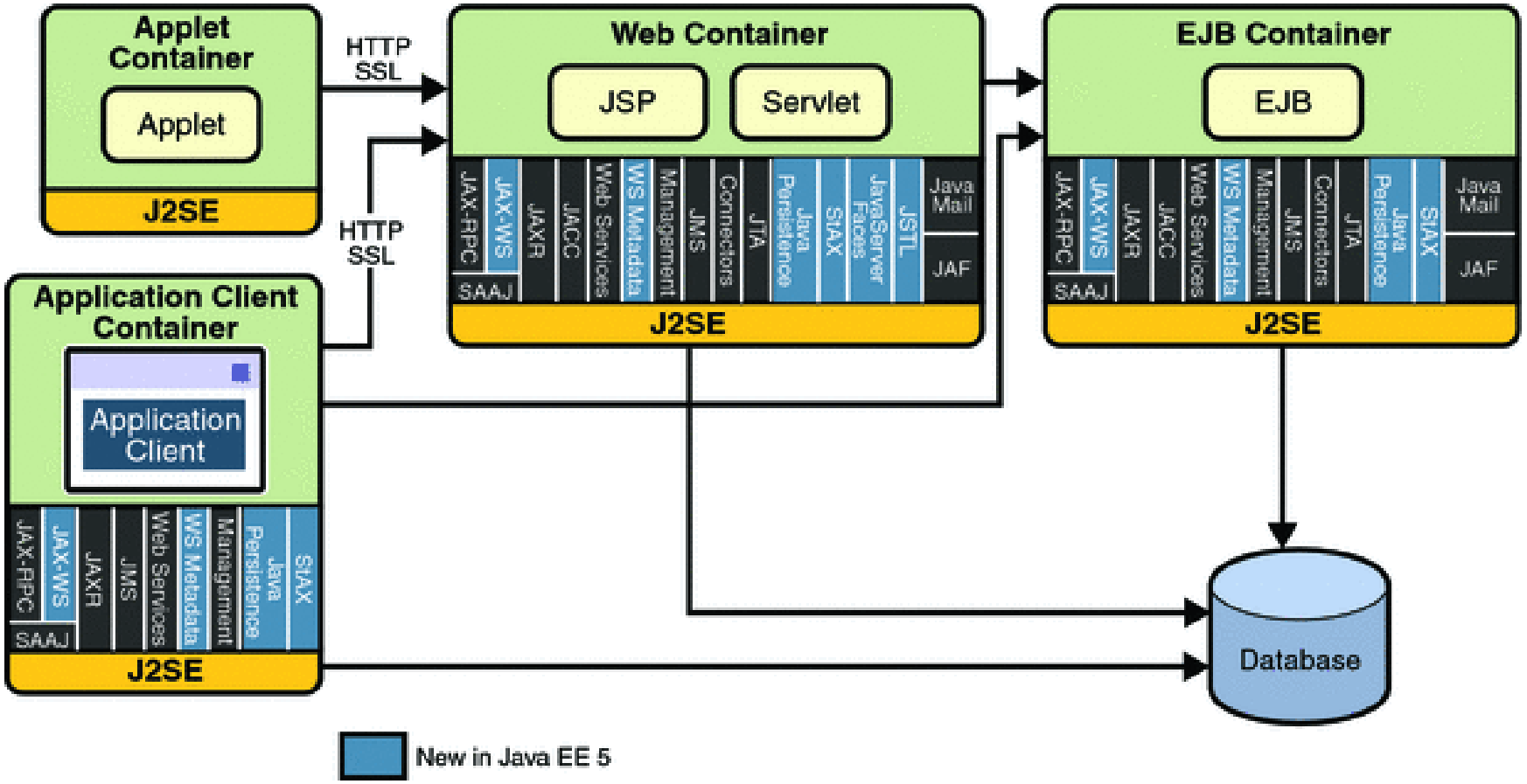
Containers As Defined in Java EE

- **Containers** are the interface between a component and the low-level platform-specific functionality that supports the component.
- **Java EE server**: The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.
- **Enterprise JavaBeans (EJB) container**: Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.
- **Web container**: Manages the execution of web pages, servlets, and some EJB components for Java EE applications. Web components and their container run on the Java EE server.
- **Application client container**: Manages the execution of application client components. Application clients and their container run on the client.

Containers As Defined in Java EE



How does it look in API Mode?



Dependency Injection

Suppose You Need to Travel outstation for your job….

What would be your dependencies?

You have to book them all!!

Hence You are satisfying your own dependency yourself

Think about getting it done by your travel desk…

1. You inform your travel desk.
2. Give the details to them
3. You are done.

4. In this case you are not worried about the following
    1. Who did the reservations and how did they do it!
    2. Even if the travel partner changes your admin will take care of linking the new partner, not you!
    3. You are totally decoupled from the hassle of booking.

    4. This is the type work a dependency injection container does for You….

Dependency Injection

- Consider the following code snippet

```
public class MyDao {
protected DataSource dataSource =
    new DataSourceImpl("driver", "url", "user", "password");
//data access methods...
    public Person readPerson(int primaryKey) {...}
}
```

- It cannot carry out its work without a DataSource implementation.
- Therefore, MyDao has a "dependency" on the DataSource interface and on some implementation of it.
- MyDao is responsible for creating the object of DataSource.
- It Means MyDao is responsible for satisfying its own dependency.

Let's change the design a little and see….

```
public class MyDao {
protected DataSource dataSource = null;
public MyDao( String driver, String url, String user, String password ){
    this.dataSource = new DataSourceImpl(driver, url, user, password);
  }
//data access methods...
  public Person readPerson(int primaryKey) {...}
}
```

Notice how the DataSourceImpl instantiation is moved into a constructor.

The constructor takes four parameters which are the four values needed by the DataSourceImpl

- MyDao still depends on a DataSource Object, but the four Constructor parameters are provided by the class which instantiates MyDao.
- The values for constructor parameters are said to be “injected”.
- Hence the term Dependency Injection

The MyDao class can still be made more independent..

```
public class MyDao {
protected DataSource dataSource = null;

    public MyDao(DataSource dataSource){
      this.dataSource = dataSource;
    }
//data access methods...
    public Person readPerson(int primaryKey) {...}
}
```

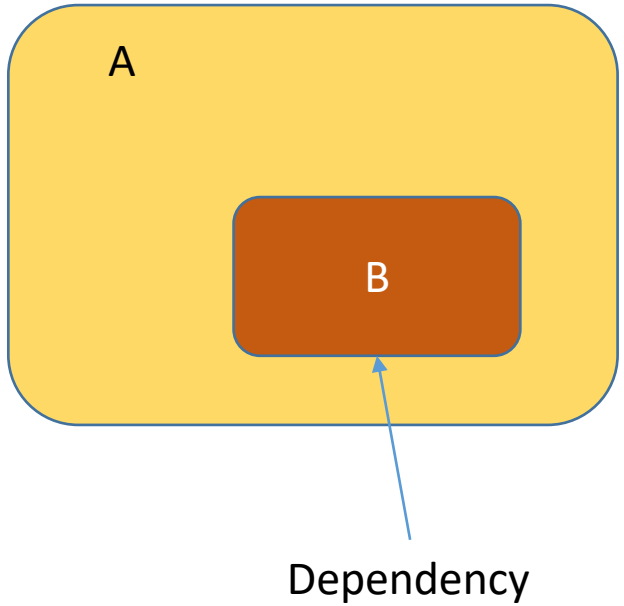
There is no need for it to depend on more than the DataSource interface.

This can be achieved by injecting a DataSource into the constructor instead of the four string parameters

- You can now inject any DataSource implementation into the MyDao constructor

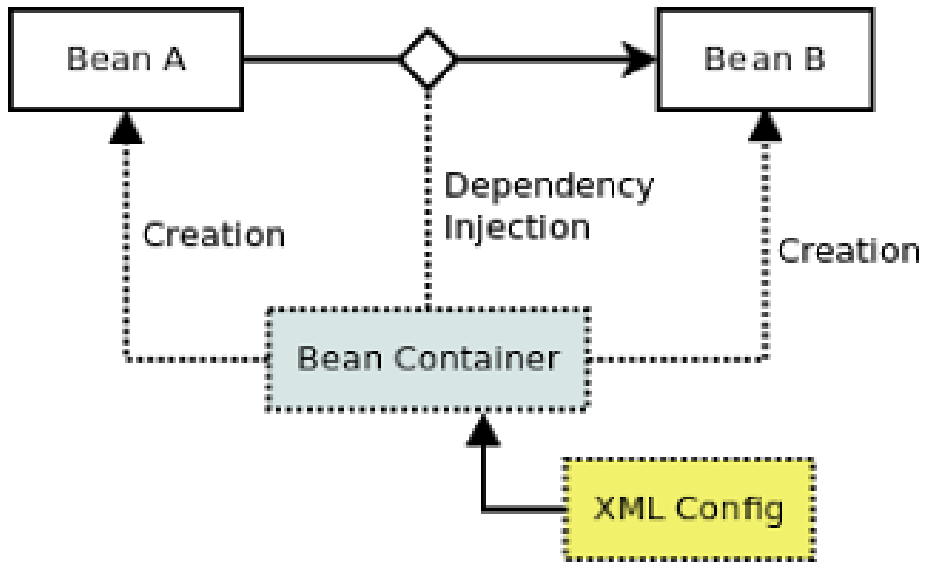
So we define **dependency injection as follows**

- **Dependency injection** is a technique whereby one object supplies the dependencies of another object.
- A dependency is an object that can be used (a service).
- An injection is the passing of a dependency to a dependent object (a client) that would use it.
- The service is made part of the client's state.
- Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.



Dependency Injection Containers

- Dependency Injection Containers inject dependency of objects.
- They read the dependencies from configuration metadata (XML or Annotation)
- Create Objects and injects them into other objects.
- Maintain lifecycle of Objects.



Types of Dependency Injection

- There are at least three ways an object can receive a reference to an external module
- ***constructor injection***
  - the dependencies are provided through a class constructor.
- ***setter injection***
  - the client exposes a setter method that the injector uses to inject the dependency.
- ***interface injection***
  - the dependency provides an injector method that will inject the dependency into any client passed to it.
  - Clients must implement an interface that exposes a setter method that accepts the dependency.

Code Without Dependency Injection

```java
// An example without dependency injection
public class Client {
    // Internal reference to the service used by this client
    private ServiceExample service;

    // Constructor
    Client() {
        // Specify a specific implementation in the constructor instead of using dependency injection
        service = new ServiceExample();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

The client controls which implementation of service is used and controls its construction.

In this situation, the client is said to have a hard-coded dependency on ServiceExample.

Constructor injection

- This method requires the client to provide a parameter in a constructor for the dependency.

```
// Constructor
Client(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Setter injection

- This method requires the client to provide a setter method for the dependency.

```
// Setter method
public void setService(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

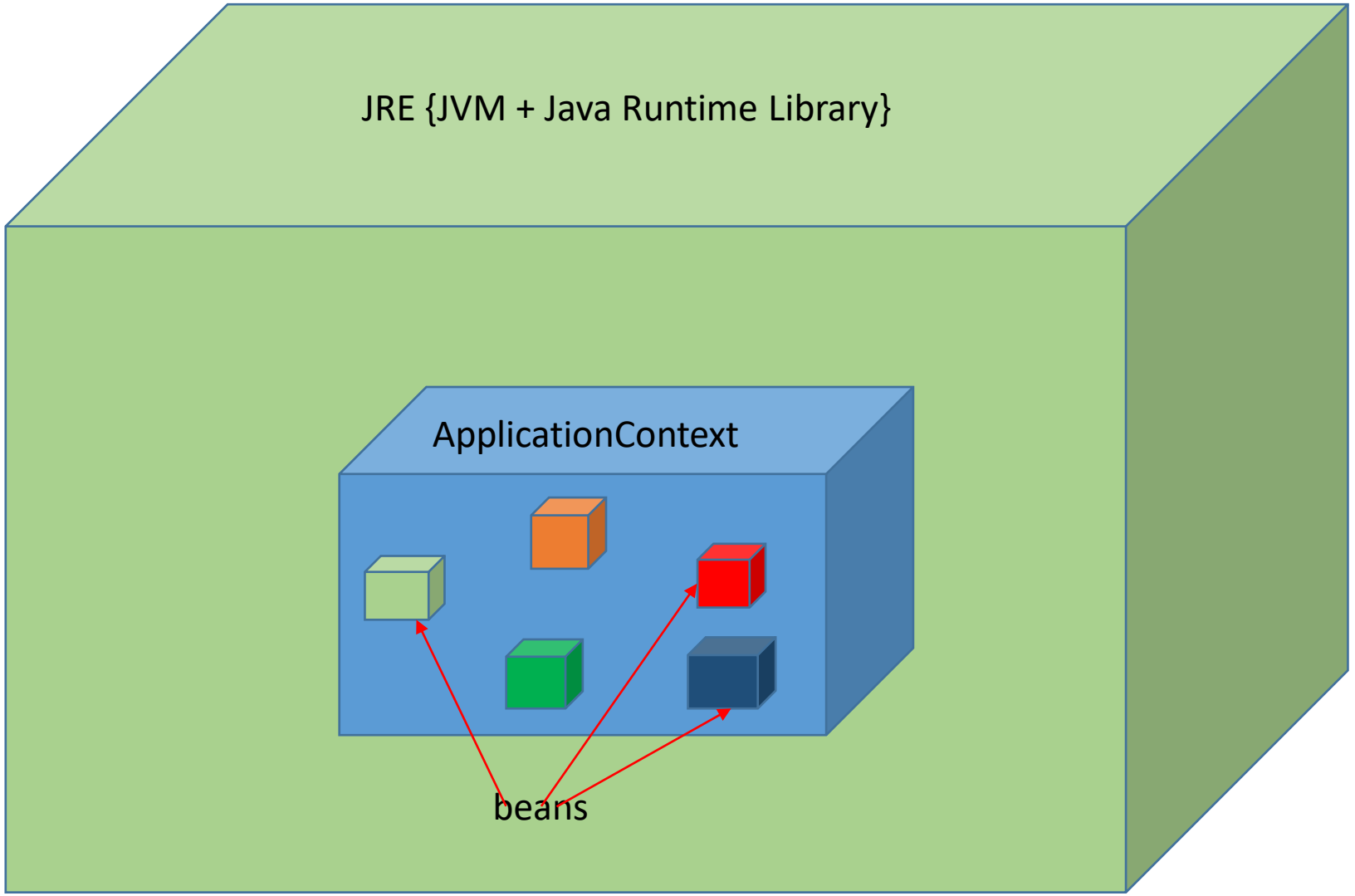
Interface injection

- This is simply the client publishing a role interface to the setter methods of the client's dependencies.
- It can be used to establish how the injector should talk to the client when injecting dependencies.

```java
// Service setter interface.
public interface ServiceSetter {
    public void setService(Service service);
}

// Client class
public class Client implements ServiceSetter {
    // Internal reference to the service used by this client.
    private Service service;

    // Set the service that this client is to use.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```


JRE {JVM + Java Runtime Library}
ApplicationContext
beans