1

POJO Programming Model, Lightweight Containers, and Inversion of Control

COPYRIGHTED MATERIAL

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Problems of the old EJB programming model that triggered the birth of POJO movement
- Advantages of the POJO programming model
- What a container is and what services it provides to its deployed applications
- Lightweight containers and what makes a container lightweight
- What Inversion of Control (IoC) means and its importance for applications
- Relationship between IoC and dependency injection
- Dependency injection methods, setter and constructor injection
- Advantages and disadvantages of those different dependency injection methods

The *Plain Old Java Object* (POJO) movement started around the beginning of the 2000s and quickly became mainstream in the enterprise Java world. This quick popularity is certainly closely related with the open source movement during that time. Lots of projects appeared, and most of them helped the POJO programming model become mature over time. This chapter first closely examines how things were before the POJO programming model existed in the enterprise Java community and discusses the problems of the old Enterprise JavaBeans (EJB) programming model. It's important that you understand the characteristics of the POJO programming model and what it provides to developers.

The second half of the chapter focuses on containers and the inversion of control patterns that are at the heart of the lightweight containers we use today. You learn what a container is, what services it offers, and what makes a container lightweight. You also learn how the inversion of control pattern arises and its close relationship with dependency injection terms. The chapter concludes with an examination of two different dependency injection methods and their pros and cons.

POJO PROGRAMMING MODEL

POJO means Plain Old Java Objects. The name was first coined by Martin Fowler, Rebecca Parsons, and Josh MacKenzie to give regular Java objects an exciting-sounding name. It represents a programming trend that aims to simplify the coding, testing, and deployment phases of Java applications—especially enterprise Java applications.

You'll have a better understanding of what problems the POJO programming model solves if you first understand what problems the old EJB programming model had.

Problems of the Old EJB Programming Model

The Enterprise JavaBeans (EJB) technology was first announced around 1997. It offered a distributed business component model combined with a runtime platform that provided all the necessary middleware services those EJB components needed for their execution. It was a main specification under the J2EE specification umbrella at the time.

Many people were really excited by the promise of the EJB technology and J2EE platform. EJBs were offering a component model that would let developers focus only on the business side of the system while ignoring the middleware requirements, such as wiring of components, transaction management, persistence operations, security, resource pooling, threading, distribution, remoting, and so on. Developers were told that services for middleware requirements could be easily added into the system whenever there was any need of them. Everything seemed good and very promising on paper, but things didn't go well in practice.

The EJB 2.x specification required that the component interface and business logic implementation class extend interfaces from the EJB framework package. These requirements created a tight coupling between the developer-written code and the interface classes from the EJB framework package. It also required the implementation of several unnecessary callback methods, such as `ejbCreate, ejbPassivate`, and `ejbActivate`, which are not directly related to the main design goal of EJB.

To develop an EJB component, developers had to write at least three different classes—one for home, one for remote interfaces, and one for business objects, as shown here:

```
public interface PetClinicService extends EJBObject {
  public void saveOwner(Owner owner) throws RemoteException;
}


public interface PetClinicServiceHome extends EJBHome {
  public PetClinicService create() throws RemoteException, CreateException;
```

```
}


public class PetClinicServiceBean implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
  }
  public void saveOwner() throws java.rmi.RemoteException {
    //implementation of saving owner instance...
  }
}
```

The preceding code snippet shows the minimum amount of code that needs to be written in order to create an EJB component with only one method using the EJB2 application programming interface (API). Although the remote interface defined the public API of the business object class to the outside world, a non-mandatory requirement in the specification asked that the business object class implementation not depend on the remote interface directly. When developers obeyed this warning, however, they were opening up a possibility that business object class implementation and its public API remote interface would become unsynchronized whenever the method declarations were modified in one of those classes. The solution was to introduce a fourth interface, which was implemented by the business object class and extended by the remote interface to keep the remote interface and the business object class implementation synchronized while not violating this non-mandatory requirement.

There were actually two interfaces that defined the public API of the business object class: the remote and local interfaces. Local interfaces were introduced to the EJB specification when people realized that remote interfaces were causing unnecessary performance overheads in systems in which there were no physically separated layers, and there was no direct access to the EJB layer from another client in the architecture, except through servlets. However, when developers needed to make EJB components remotely available they had to create a remote interface for them. Although there was no direct dependency between the business object class and its remote interface, all public methods of the business object implementation class had to throw `RemoteException`, causing the business object implementation class to depend on EJB and remoting technologies.

Testability was one of the biggest problems of the old EJB programming model. It was almost impossible to test session and entity beans outside the EJB container; for example, inside an integrated development environment (IDE) using JUnit. This is because dependencies of those session beans were satisfied through local or remote interfaces, and it was very hard—but not impossible—to test session beans in a standalone environment. When it came time to run or test entity beans outside the container, things were more difficult because the entity bean classes had to be abstract and their concrete implementations were provided by the EJB container at deployment time. Because of such difficulties, people tried to access the EJBs deployed in the container and test them using in-container test

frameworks, such as Cactus. Nevertheless, such solutions were far from the simplicity and speed of running tests within a standalone environment by right-clicking and selecting Run As JUnit Test.

The deployment process was another time-consuming and error-prone phase of the EJB programming model. Developers used deployment descriptor files in XML format to deploy developed EJB components, but configuring their middleware requirements, such as transaction semantics, security requirements, and so on, caused those files to become several hundred lines long. Developers usually were trying to maintain the files by hand, and it was quite easy to make simple typos in package or class names, and those errors wouldn't be noticed until deployment time.

The following code snippet contains two EJB definitions, one depending on the other, and it includes a container-managed transaction configuration as well. Imagine how things can go wrong when you have dozens of other EJB definitions, each having its own dependencies, transaction management, security configurations, and so on:

```
<ejb-jar>
    <display-name>PetClinicEJB2</display-name>
    <enterprise-beans>
        <session>
            <ejb-name>PetClinicService</ejb-name>
            <home>com.example.PetClinicServiceHome</home>
            <remote>com.example.PetClinicService</remote>
            <ejb-class>com.example.PetClinicServiceImpl</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <resource-ref>
                <res-ref-name>jdbc/ds</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </session>
        <message-driven>
            <ejb-name>MessageSubscriber</ejb-name>
            <ejb-class>com.example.MessageSubscriber</ejb-class>
            <transaction-type>Container</transaction-type>
            <message-destination-type>javax.jms.Topic</message-destination-type>
            <ejb-ref>
                <ejb-ref-name>ejb/PetClinicService</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
                <home>com.example.PetClinicServiceHome</home>
                <remote>com.example.PetClinicService</remote>
                <ejb-link>PetClinicService</ejb-link>
            </ejb-ref>
        </message-driven>
    </enterprise-beans>

    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>PetClinicService</ejb-name>
                <method-name>saveOwner</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
```

```
            </container-transaction>
        </assembly-descriptor>
    </ejb-jar>
```

One very common task while coding EJBs was to access the Java Naming and Directory Interface (JNDI) context in the J2EE environment and perform object lookups so that necessary dependencies to other EJBs and `DataSource` instances could be satisfied. However, this was causing the EJB component to become tightly coupled with the container, and unit testing was hard to perform because of this environmental dependency. The following code snippets show how an EJB home object and `javax.sql.DataSource` are looked up from a JNDI repository:

```
try {
    InitialContext context = new InitialContext();
    PetClinicServiceHome petClinicServiceHome = (PetClinicServiceHome)
        context.lookup("java:/comp/env/ejb/PetClinicService");
    PetClinicService petClinicService = petClinicServiceHome.create();
    //you can now access business methods of the component...
} catch (NamingException e) {
    throw new RuntimeException(e);
}


try {
    InitialContext context = new InitialContext();
    DataSource ds = (DataSource)context. lookup("java:/comp/env/jdbc/ds");
    //you can now obtain JDBC Connections via DataSource object...
} catch (NamingException e) {
    throw new RuntimeException(e);
}
```

Actually, JNDI lookup can be considered an early form of dependency injection, but, due to its pull-based nature, it was difficult to isolate components during unit testing because of the dependency to the JNDI context.

Another problem of the old EJB programming model was that it diverted developers toward the procedural programming style. Application behavior in this style of programming is mainly handled within some methods, while data from and to those methods is carried with dumb domain model objects. Unfortunately, data and behavior are separated from each other and are not in a cohesive form in such a case. This is definitely a divergence from the object-oriented programming perspective in which one of the important characteristics is encapsulation of data together with the related behavior. After all, you are using an object-oriented programming language called Java, and you want to take advantage of all its abilities, don't you?

The main reason for such a paradigm shift, while using an object-oriented language, was the EJB programming model. People usually were developing session- and message-driven beans that were stateless, monolithic, and heavyweight components in which all the business logic was implemented with data access operations inside them. Entity EJBs were expected to represent the domain model, but they had some subtle deficiencies that prevented them from being used at all. For example, inheritance support was too limited, and recursive calls within entity beans were not supported; it was not possible to transfer the entity bean instances as session and message-driven bean method inputs and return values, and so on.

People might think that procedural style is not a big problem for scenarios in which business logic is simple. However, things don't stay simple in real-life enterprise application projects. As new requirements come along, things become more complex and written code grows to be more and more of a maintenance headache. The procedural style of programming that was promoted by the old EJB programming model caused the creation and use of dumb domain objects, which were acting purely as data transfer objects between the application layers and the network. Martin Fowler coined the term *anemic domain model* for such problematic domain objects. Anemic blood is missing vital ingredients; similarly, an anemic domain model is also limited to only data transfer and persistence-related operations, and it contains hardly any behavioral code. Unfortunately, the old EJB programming model was not able to enforce operating on a fine-grained and rich object model behind a coarse-grained component model.

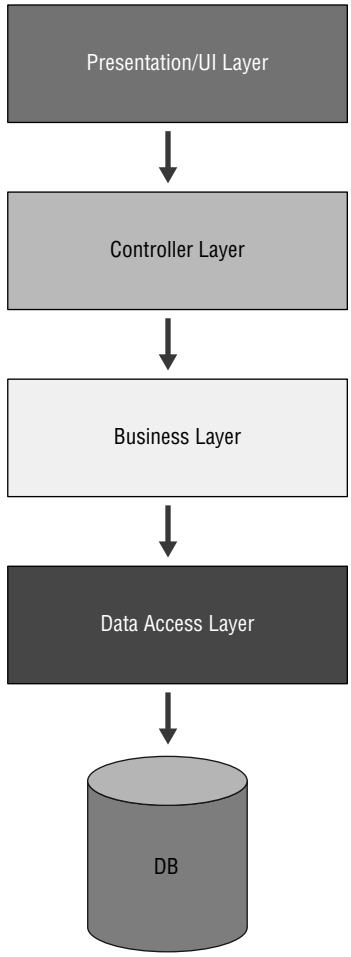

FIGURE 1-1

Enterprise applications usually have layered architectures. They are mainly composed of the web, service, and data access layers. Figure 1-1 shows those logical layers and the relationships between each.

Each layer should only know and interact with the layer just beneath it. That way, upper layers aren't affected by changes made within other layers upon which they don't directly depend. It also becomes possible to easily replace layers because only one layer depends on another, and only that dependent layer will have to be changed if there is a need.

It is a desirable and correct approach to divide the system into several logical layers. However, this doesn't mean that there should always be a one-to-one correspondence between physical layers. Unfortunately, having an EJB container caused those web and service layers to work using remote method invocation (RMI), which is practically equivalent to having separate physical layers. Hence, servlet and JavaServer Pages (JSP) components in the web layer have complex and performance-degrading interactions with the EJB components in the service layers. Apart from inefficient network interaction, developers also experienced class- and resource-loading issues. The reason for these issues were that the EJB container used a different `ClassLoader` instance than the web container.

Figure 1-2 shows a typical physical layering of a J2EE application. The application server has separate web and EJB containers. Therefore, although they are located in the same server instance, web components have to interact with EJB components as if they are in different physical servers using RMI. It is observed in many enterprise Java applications that RMI calls from the web to the service layers create an unnecessary performance cost over time when the web and EJB layers are located in the same physical machine, and the EJB layer is only accessed from the web layers. As a result, local interfaces were introduced to get rid of RMI between those layers.

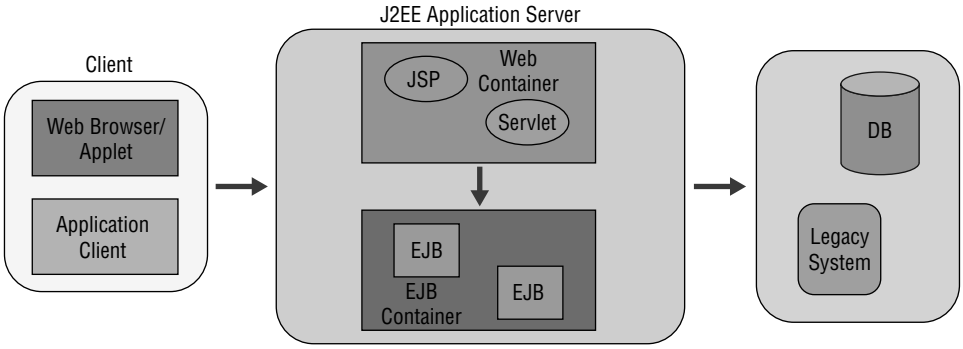

FIGURE 1-2

The "write once and run everywhere" slogan was very popular at those times, and people expected it to be true among J2EE environments as well. However, there were lots of missing and open issues in EJB and J2EE specifications, so many enterprise projects had to develop solutions specific to their application servers. Every application server had its own legacy set of features, and you had to perform server-specific configurations, or code against a server-specific API to make your application run in the target environment. Actually, the slogan had turned into "write once and debug everywhere," and this was a common joke among J2EE developers.

Most of the aforementioned problems were addressed in the EJB 3 and EJB 3.1 specifications. The most important point during those improvements is that the POJO programming model was taken as a reference by those newer EJB specifications. Session and message-driven beans are still available but much simpler now, and entity beans are transformed into POJO-based domain objects with the Java Persistence API (JPA). It is now much easier to implement, test, and deploy them. The EJB programming model has become more and like the POJO programming model over time.

Certainly, the biggest contribution to improve the EJB component model and J2EE environment has come from POJO-based, lightweight frameworks, such as Hibernate and Spring. We can safely say that the EJB programming model mostly was inspired by those frameworks, especially Spring.

Benefits of the POJO Programming Model

The most important advantage of the POJO programming model is that coding application classes is very fast and simple. This is because classes don't need to depend on any particular API, implement any special interface, or extend from a particular framework class. You do not have to create any special callback methods until you really need them.

Because the POJO-based classes don't depend on any particular API or framework code, they can easily be transferred over the network and used between layers. Therefore, you don't need to create separate data transfer object classes in order to carry data over the network.

You don't need to deploy your classes into any container or wait for long deployment cycles so that you can run and test them. You can easily test your classes within your favorite IDE using JUnit. You don't need to employ in-container testing frameworks like Cactus to perform integration unit tests.

The POJO programming model lets you code with an object-oriented perspective instead of a procedural style. It becomes possible to reflect the problem domain exactly to the solution domain. Business logic can be handled over a more fine-grained model, which is also richer in terms of behavioral aspects.

LIGHTWEIGHT CONTAINERS AND INVERSION OF CONTROL (IOC)

Despite all the difficulties and disadvantages of the old EJB programming model, there were still some attractive points in the platform that caused many people to develop enterprise Java applications and deploy them into J2EE application servers. It was very important that several middleware services crucial for applications to work were readily provided by the J2EE environment, and developers were able to utilize them in their applications. For example, the following actions are independent from business logic, and it's important that they are provided by a J2EE platform:

- Handling database connections outside the application codebase
- Enabling pooling capabilities, if necessary
- Performing transaction management with declarative means
- Working with a ready-to-use transaction management infrastructure
- Creating and wiring of components in the application
- Applying security constraints on the system
- Dealing with thread and scheduling issues

Lightweight Containers

Some people were developing their applications without using EJBs while still leveraging many of those middleware features mentioned earlier. On the other hand, they usually perceived that they had to deploy their application to a full-featured J2EE application server only so that they could leverage those middleware services. This was quite a wrong opinion at the time. It is technically possible to develop an enterprise application without using a container at all. In that case, however, you need to handle the creating and wiring of components and implement required middleware services yourself. These tasks will definitely distract you from dealing solely with business requirements of the system, and delay the completion time of it.

Therefore, in practice it is much better to have an environment by which all those components will be created and wired and those required middleware services will be provided. Such an environment is called a *container*. The Java EE platform provides several such containers, each specialized with services required by a particular layer in the application. For example, the Servlet container creates and manages components of the web layer of an application, such as Servlets, JSPs, Filters, and so on. The EJB container, on the other hand, focuses on the business layer of the application and manages the EJB components of it. Similar to the Java EE platform, the Spring Container is also a container in which components of an application are created, wired with each other, and the middleware services are provided in a lightweight manner.

When we talk about containers, it is expected that any container should be capable of providing several basic services to components managed in its environment. According to the seminal book *Expert One-on-One J2EE Development Without EJB* by Rod Johnson and Jürgen Höller (Wrox, 2004), those expected services can be listed as follows:

- Life-cycle management
- Dependency resolution
- Component lookup
- Application configuration

In addition to those features, it will be very useful if the container is able to provide following middleware services:

- Transaction management
- Security
- Thread management
- Object and resource pooling
- Remote access for components
- Management of components through a JMX-like API
- Extendibility and customizability of container

A *lightweight container* includes all of these features, but doesn't require application code to depend on its own API. That is, it doesn't have invasive character, its startup time is very fast, it doesn't need to be deployed into a full-featured Java EE application server to be able to provide those services, and deploying components into it is a trivial process. The Spring Application Framework is one of the most prominent lightweight containers in the enterprise world.

Inversion of Control (IoC)

One of the most important benefits containers that provide with components they manage is pluggable architecture. Components implement some interfaces, and they also access services provided by other components they need through similar interfaces. They never know concrete implementation classes of their services. Therefore, it becomes very easy to replace any component in the system with a different implementation. The job of a container is to create those components and their dependent services and wire them together.

Dependent components are never instantiated using a new operator within component classes. They are injected into the component by the container instance at run time. Hence, control of dependencies is moved out of components to the container. This pattern, therefore, is called *Inversion of Control*, or IoC for short. IoC is an important concept in frameworks generally, and is best understood through the Hollywood principle of "Don't call us; we'll call you."

IoC is one of the fundamental features that is expected to be provided by any container. It has basically two forms: dependency lookup and dependency injection.

In *dependency lookup*, the container provides callback methods to the components it manages, and the components interact with the container and acquire their dependencies explicitly within those callback methods. In such a scenario, there is usually a lookup context that is used to access dependent components and other resources managed by the container.

In *dependency injection*, components are provided with suitable constructors or setter methods so that the container can inject dependent components. There is hardly ever an explicit lookup performed within components. Most of the time dependencies are injected during creation of components through those methods.

The method used during the early years of J2EE corresponds to dependency lookup. The lookup context mentioned earlier was also called the JNDI context in this environment. EJB components and other resources such as JDBC `DataSource` and JMS `ConnectionFactory` were accessed through that JNDI context. Figure 1-3 depicts explicit interaction of various parts with the JNDI repository in the J2EE platform via JNDI API.

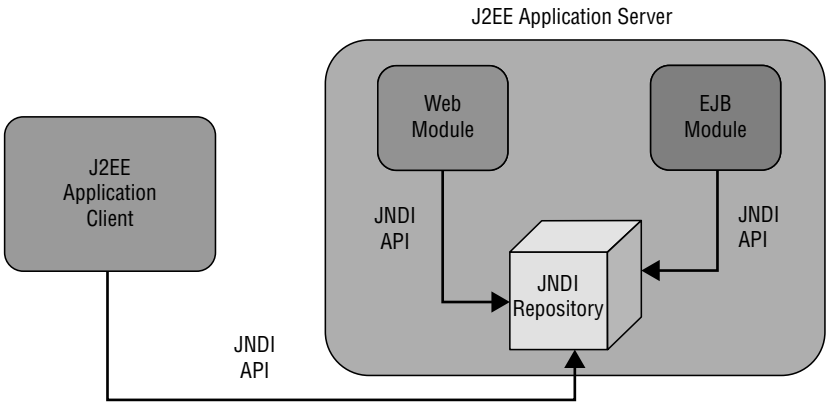

FIGURE 1-3

With the advent of the Spring Application Framework and other lightweight IoC frameworks, the dependency injection method has become popular. In this scenario, how components are instantiated and what dependent components they need are defined using a container's own configuration mechanism. It is the job of the container to process this configuration information to instantiate necessary components and wire up their dependencies at run time. During the evolution process of J2EE toward Java EE, explicit dependency lookup using JNDI has been transformed into the implicit dependency injection method. Today, when IoC is mentioned, it is usually understood as dependency injection among developers.

DEPENDENCY INJECTION

The fundamental principle of dependency injection is that application objects should not be responsible for looking up the resources or collaborators on which they depend. Instead, an IoC container should handle object creation and dependency injection, resulting in the externalization of resource lookup from application code to the container.

Dependency injection has several benefits to the overall system. First of all, lookup logic is completely removed from application code, and dependencies can be injected into the target component in a pluggable manner. Components don't know the location or class of their dependencies. Therefore, unit testing of such components becomes very easy because there is no environmental dependency like the JNDI context, and dependent components can easily be mocked and wired up to the component in the test case. Configuration of the application for different environments also becomes very easy and achievable without code modification because no concrete class dependencies exist within components. There is no dependence on the container API. Code can be moved from one container to another, and it should still work without any modification in the codebase. There is no requirement to implement any special interfaces at all. Written classes are just plain Java objects, and it is not necessary to deploy those components to make them run.

Two dependency injection methods can be used. One is constructor injection, and the other is setter injection. A good container should be able to support both at the same time, and should allow mixing them.

Setter Injection

The setter methods are invoked immediately after the object is instantiated by the container. The injection occurs during the component creation or initialization phase, which is performed much earlier in the process than handling business method calls. Thus, there are no threading issues related with calling those setter methods. Setter methods are part of the JavaBean specification, so that they allow the outside world to change collaborators and property values of components. Those JavaBean properties are also used to externalize simple properties such as `int` or `boolean` values. This simplifies the code and makes it reusable in a variety of environments.

The most important advantage of setter injection is that it allows re-configurability of the component after its creation. The component's dependencies can be changed at run time. Many existing classes can already be used with standard JavaBean-style programming. In other words, they offer getter and setter methods to access their properties. For example, Jakarta Commons DBCP `DataSource` provides a commonly used `DataSource` implementation, and it can be managed via its JavaBean properties within the container. It's possible to use the standard JavaBeans property-editor mechanism for type conversions whenever necessary. For example, a `String` value given in configuration can easily be converted into a necessary typed value, or a location can be resolved into a resource instance, and so on. If there is a corresponding getter for each setter, it becomes possible to obtain the current state of the component and save it to restore for a later time. If the component has default values for some or all of its properties, it can be configured more easily using setter injection. You can still optionally provide some dependencies of it as well.

The biggest disadvantage of setter injection is that not all necessary dependencies may be injected before use, which leaves the component in a partially configured state. In some cases, the order of invocation of setter methods might be important, and this is not expressed in the component's contract. Containers provide mechanisms to detect and prevent such inconsistencies in component states during their creation phase.

Constructor Injection

With constructor injection, beans express their dependencies via constructor arguments. In this method, dependencies are injected during component creation. The same thread safety applies for

constructor injection as well. You can also inject simple properties such as `int` or `boolean` values as constructor arguments.

The biggest advantage of constructor injection is that each managed component in the container is guaranteed to be in a consistent state and ready to use after it is created. Another good point is that the amount of code written with constructor injection will be slightly less compared to the code written when setter injection is used.

The biggest disadvantage of constructor injection is that it won't be possible to reconfigure components after their creation unless they provide a setter for those properties given as constructor arguments. Having several overloaded constructors for different configuration options might be confusing or even unavailable most of the time. Concrete inheritance can also be problematic unless you are careful about overriding all of the constructors in the superclass.

Setter or Constructor Injection

Both methods have advantages as well as disadvantages, and it is not possible to use only one method for any application. You might have classes especially written by third parties that don't have constructors that accept suitable arguments for your configuration case. Therefore, you might first create a component with an available constructor that accepts arguments close to your needs, and then inject other dependencies with setter methods. If the components need to be reconfigurable at run time, having setters for their specific properties will be mandatory in that case. IoC containers are expected to allow developers to mix the two types of dependency injection methods for the same component within the application configuration.

SUMMARY

In this chapter, you first learned the problems of the old-school EJB programming model that caused many enterprise Java projects to fail completely—or at least fail to satisfy their promises to some degree. The main problems of the old EJB programming model was that developers had to write several interfaces to create a business component, tight coupling between EJB and J2EE technologies was necessary, you couldn't run components outside the J2EE platform, there was difficulty in unit testing outside the container, long and complex develop-package-deploy-test cycles were required, and the characteristics and limitations of J2EE technologies required promotion of the procedural style of programming. Then you found out how those problems led to the creation of the POJO programming model, how the POJO programming model solves the problems of the EJB programming model, and how the POJO programming model helped J2EE to evolve into the new Java EE environment.

This chapter discussed why so many people insisted on using J2EE technologies and tried to deploy their enterprise applications despite all those obstacles in the J2EE environment. After identifying the attractive points of the J2EE platform, we defined what a container is, listed fundamental features a container should offer to its applications, and identified what makes a container lightweight by looking at its characteristics.

The last part of the chapter focused on what IoC is, and what any container should offer as its core services. We discussed how IoC helps make applications more modular and pluggable. The chapter

wrapped up with an explanation of dependency injection, which is a form of IoC, and its two different types: setter injection and constructor injection.

EXERCISES

You can find possible solutions to the following exercises in Appendix A.

1. Investigate the in-container test frameworks available today. What are their biggest advantages and disadvantages compared to testing outside the container?
2. What IoC method is used by the new EJB programming model today?
3. Which dependency injection method can handle "circular dependencies" and which cannot?

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY POINTS
POJO	Plain Old Java Objects, a term devised to infer Java classes that don't depend on any environment-specific classes or interfaces, and don't need any special environment to run in.
EJB	Enterprise JavaBeans, the distributed business component model of the J2EE platform.
J2EE, Java EE	Java 2 Enterprise Edition, an umbrella specification that brings several different technologies together and forms the enterprise Java environment. Java Enterprise Edition (Java EE) is its newer name after Java release 5.
Container, EJB Container, Web Container	An environment in which components are created and wired together in addition to utilizing middleware services offered by the container.
Middleware services	Requirements that appear in every application, independent of business requirements such as transaction, persistence, security, remoting, threading, connection and resource pooling, caching, validation, and clustering.
Home interface	Special interface that needs to be implemented in the old EJB programming model so that clients can obtain a handle of an EJB component remotely.
Remote interface	An interface that needs to be provided in the EJB programming model so that clients can invoke business functions of an EJB component remotely.
Local interface	Similar to the remote interface but derived for efficient interaction between the web layer and the EJB layer, which sit together in the same application server and JVM.
Callback methods	Methods that are implemented in the business implementation class of the EJB component and invoked by the container to let the component interact with the environment.
JNDI context	Context available in every Java EE environment in which objects are managed with their names and attributes and are accessible using JNDI.
Inversion of Control (IoC)	Pattern that represents control of managing dependencies in a component whose dependency management is taken out of it and given to the environment—in other words, the container.
Dependency lookup	A form of IoC that is based on callback methods, invoked by a container at specific phases, and lets a component look up its dependencies using a lookup context, like the JNDI context in the J2EE environment.

TOPIC	KEY POINTS
Dependency injection	A second and more popular form of IoC in which components define their dependencies and the container wires them during component creation time.
Setter injection	Dependency injection method that uses JavaBean specification setter methods.
Constructor injection	Dependency injection method that uses constructors.